Empirical Software Engineering (2026) 31:76
https://doi.org/10.1007/s10664-026-10816-4

VLM-Fuzz: Vision language model assisted recursive depth-first search exploration for effective GUI testing of android apps

Biniam Fisseha Demissie[1] · Yan Naing Tun[2] · Lwin Khin Shar[2] · Mariano Ceccato[3]

Received: 6 March 2025 / Accepted: 28 January 2026
© The Author(s) 2026

Abstract
Testing Android apps effectively requires a systematic exploration of the app's possible states by simulating user interactions and system events. While existing approaches have proposed several fuzzing techniques to generate various text inputs and trigger user and system events for GUI state exploration, achieving high code coverage remains a significant challenge in Android app testing. The main challenges are (1) reasoning about the complex and dynamic layout of GUI screens; (2) generating required inputs/events to deal with certain widgets like pop-ups; and (3) coordination between current test inputs and previous inputs to avoid getting stuck in the same GUI screen without improving test coverage. To address these problems, we propose VLM-Fuzz, a novel automated approach for Android GUI testing. At its foundation, VLM-Fuzz utilizes a heuristic-based, recursive depth-first search (DFS) strategy that is intelligently guided by a Vision Language Model (VLM) to effectively explore the app's complex GUI states. The core innovation of VLM-Fuzz is not simply the use of a VLM, but its strategic, on-demand integration within a hybrid exploration framework. Our approach combines a fast, heuristic-based DFS for standard GUI interactions with targeted, VLM-assisted analysis for visually complex screens. We use static analysis to analyze the Android Manifest file and the runtime GUI hierarchy XML to extract the list of components, intent-filters and interactive GUI widgets. VLM is used to reason about complex GUI layout and widgets on an on-demand basis. Based on the inputs from static analysis, VLM, and the current GUI state, we use some heuristics to deal with the above-mentioned challenges. We evaluated VLM-Fuzz based on a benchmark containing 59 apps obtained from a recent work and compared it against two state-of-the-art approaches: *APE* and *DeepGUI*. VLM-Fuzz outperforms the best baseline by 9.0% , 3.7% , and 2.1% in terms of class coverage, method coverage, and line coverage, respectively. We also ran VLM-Fuzz on 80 recent Google Play apps (i.e., updated in 2024). VLM-Fuzz detected 52 unique crashes in 12 apps, which have been reported to respective developers.

Communicated by: Markus Borg.

Extended author information available on the last page of the article

Published online: 13 February 2026

Keywords Android · Fuzzing · GUI testing · Large Language Model · Vision Language Model

1 Introduction

Graphical User Interface (GUI) is important in modern software development for guaranteeing a good user experience (Kong et al. 2018. Therefore, it is important that adequate GUI testing is conducted before the software is released. However, in the context of mobile app development, as developers are often under time-to-market pressure, they primarily focus their testing on core functionalities and common usage scenarios. This often leads to released apps containing bugs or vulnerabilities which may severely impact user experiences and thus, the app's popularity. When testing the app, test engineers may manually write and maintain test scripts using frameworks like Espresso (Google 2025). This method leverages human cognition to its fullest extent, allowing for the creation of sophisticated test cases that can navigate complex app logic. While this approach represents the gold standard for test case design, significant practical limitations fundamentally constrain it (Kochhar et al. 2015). The cost of human effort in creating and maintaining these test scripts is substantial. Furthermore, manual scripting is a slow process that cannot scale to provide the rapid, comprehensive feedback needed for regression testing in fast-paced, continuous integration environments. Cruz et al. (2019) reported that mobile app developers are becoming more aware of the importance of using automated testing.

Existing automated GUI testing approaches can be categorized based on their exploration strategies such as random (Android 2024b), search-based (Amalfitano et al. 2012, 2014; Azim and Neamtiu 2013; Mao et al. 2016), model-based (Su et al. 2017; Dong et al. 2020), or program analysis based (Anand et al. 2012; Moran et al. 2017) approaches. The main limitation of search-based approaches is the mismatch between a fitness function (to guide the visit) that is statically defined at source code level, while testing is performed at GUI level, on a GUI that potentially evolves dynamically. The model-based approaches, instead, are struggling when a reliable *model* of the app to test is not available or, alternatively, to infer a model from a complex and dynamic GUI. Therefore, to deliver an effective automated test cases generation approach, the first challenge to address is:

> Challenge ❶ *Reasoning About Complex UI Layouts:* inef cient and overly complex GUI representation, which hinders effective reasoning about dynamic GUI screens and contributes to performance issues.

Additionally, these approaches tend to generate unnatural test scenarios that achieve a limited code coverage when dealing with complex interactions with the user, the system, and other apps because test inputs generated by these approaches do not resemble real users' actions (Peng et al. 2022). To cope with this limitation, other approaches have incorporated machine learning techniques such as deep learning (YazdaniBanafsheDaragh and Malek 2021) and reinforcement learning (Pan et al. 2020; Romdhana et al. 2022). However, deep learning approaches generally require large amount of labeled training data and it may not be scalable to label a representative list of GUI items. Large Language Models (LLMs) such as GPT and Gemini have emerged as a powerful tool for natural language understanding

and image recognition. Recent advances in LLM have triggered various studies examining the use of these models for software engineering tasks such as code completion (Du et al. 2024) and test generation (Ryan et al. 2024). As such, recent approaches have applied large language models for GUI testing of Android apps (Feng and Chen 2024; Liu et al. 2024). These approaches essentially formulate the GUI testing problem as a questions & answering task, i.e., asking the LLM to play the role of a human tester to test the target app, with various text prompting techniques. While these approaches may be effective, by leveraging only text-based prompting techniques, they have not explored the full capability of LLMs where they can be leveraged to reason with images such as screenshots of activity component screens. Thus, the second challenge to address is:

> Challenge ❷ *Generating Required Inputs and Events:* generation of diverse and human-like test inputs to explore critical functionalities.

While LLMs may be good at generating test inputs for a given GUI, a good state space exploration strategy is still needed to keep track of explored and unexplored states and to identify promising paths that could lead to better code coverage. So, the explored/unexplored state of GUI components should be explicitly logged to visit and test all the app methodically and exhaustively. Consequently, the theird challenge to address is:

> Challenge ❸ *Efficient and Coordinated State Exploration:* coordination between current test component and previous to avoid getting stuck in repetitive GUI states without improving test coverage.

These three challenges are not independent; they collectively contribute to the critical problem of low code coverage. (1) Failure to reason about complex layouts means the fuzzer cannot understand which inputs are required to unlock new functionality, leaving entire code paths unexplored. (2) The inability to generate specific inputs for widgets like pop-ups or custom controls means the code that handles these interactions is never executed. Finally, (3) poor coordination and getting stuck in GUI loops leads to redundant testing of the same few screens, wasting time budget that could be spent exploring undiscovered parts of the app's state space. Addressing these root causes is therefore essential to improving test coverage.

To avoid the mentioned limitations of search-based, model-based and LLM-based approaches, we investigate an alternative approach to that requires no fitness function and no model, but that it is based on the exhaustive exploration of all the app and its widgets using a combined depth-first and breadth-first visit. We propose VLM-Fuzz, a novel multiple-entry automated fuzzing approach for effective GUI testing of Android apps. The core innovation of VLM-Fuzz lies not in using a VLM in isolation, but in its strategic, on-demand integration within a hybrid, multiple-entry exploration framework. The approach first analyzes the AndroidManifest file to identify app components and relevant system events for each component, and it examines the dynamic GUI hierarchy XML to capture interactive GUI widgets (challenge ❶ *Reasoning About Complex GUI Layouts*). To optimize testing efficiency, VLM-Fuzz evaluates component complexity based on the number of interactive GUI elements and allocates testing time budget accordingly. It then generates appropriate inputs/events to different widgets leveraging both heuristics and Vision Language Model

(VLM) (challenge ❷*Generating Required Inputs and Events*). It adopts a recursive depth-first search (DFS) to explore deep interaction paths, but for each GUI state it visits, it first performs a comprehensive local exploration of all available widgets before navigating away (challenge ❸*Efficient and Coordinated State Exploration*).

The contributions of this paper are

A. *Novel Android app fuzzing algorithm*: We propose an algorithm that combines a heuristic-based recursive depth-first search exploration strategy with Vision Language Model, which addresses the challenges involved in reasoning with complex, dynamic GUI layouts and exploring the state space in an efficient manner.
B. *Comparative study with state of the arts (SOTAs)*: we compare our approach against *APE* (Gu et al. 2019) and *DeepGUI* (YazdaniBanafsheDaragh and Malek 2021) — based on a benchmark consisting of 59 apps. Our approach achieves a high code coverage (68.5 % class coverage, 53.2 % method coverage, and 46.5 % line coverage on average), outperforming the best baseline by 9.0% , 3.7% , and 2.1% in terms of class coverage, method coverage, and line coverage, respectively. We also compare our approach against GPTDroid (Liu et al. 2024), a recent and closely related tool, *analytically* (as its tool was unavailable despite our requests). We were able to replicate it following the steps given in the article, which are essentially iterative promptings of GUI information.
C. *Ablation study on the usefulness of vision LM*: We investigate the VLM's ability to reason with GUI objects and how useful it is in improving the test coverage.
D. *Bug detection capability on real world apps*: We further evaluate our approach on 80 recent official apps from Google Play. VLM-Fuzz induced crashes in 50 apps, among which we manually confirmed that 52 crashes in 12 apps are due to real bugs. The results show that VLM-Fuzz is effective in testing real-world apps.
E. *Tool availability*: We implement the proposed technique into a practical tool, VLM-Fuzz, and make it publicly available in VLM-Fuzz (2025).

The rest of the paper is organised as follows. Section 2 discusses some background on Android fuzzing and specific challenges that motivate our work. Section 3 provides an overview of our approach and presents the details of our approach. Section 4 evaluates our approach. Section 5 discusses related works. Section 6 concludes the paper and discusses future work.

2 Background

This section introduces relevant background on Android GUI fuzzing and its challenges.

2.1 GUI of Android Apps

In Android, an Activity represents a single screen with a user interface. It is the entry point for interacting with the user. The GUI is usually defined in the XML layout files located inside a specific resource folder of the APK. When an Activity is activated, e.g., when an app launch event occurs, the XML layout files are parsed and converted into a tree of

"View" objects. "View" is the base class for all GUI elements (e.g., Button, EditText). Each view object renders a rectangular area on the screen. Widgets are a specific type of View objects, which is a GUI element, used to interact with the user. Common examples of widgets include Button, EditText, CheckBox, and ImageView. It supports various GUI actions such as clicking and swiping. A widget has four categories of attributes describing its type (e.g., class), appearance (e.g., text), functionalities (e.g., clickable and scrollable), and the designated order among sibling widgets (i.e., index). Each attribute is a key-value pair.

It is important for GUI fuzzers to be able to reason about complex GUI layout and different "View" objects, and their context, to generate valid inputs. As an example, Figure 1 shows an activity that requires reasoning about the context of the app screen and the meaning of symbols such as X and ✓, which would be challenging for typical GUI fuzzers. Without any form of intelligence and reasoning capability, the fuzzer is unlikely to automatically detect that this is a command line interface and that the required inputs are a text that is potentially a valid Linux bash command followed by a tap action on the ✓ button (challenge ❷*Generating Required Inputs and Events*).

Figure 2.a shows an activity screen where a dynamic pop-up menu is overlaid on top of an initial GUI. In this case, a typical depth first search strategy may simply continue to explore on the pop-up menu while there may still be remaining states left unexplored on the initial GUI screen (challenge ❶*Reasoning About Complex GUI Layouts*).

Figure 2.b shows a GUI in which reasoning of the GUI context is required to generate valid inputs such as ISBN number (challenge ❷*Generating Required Inputs and Events*).

Figure 2.c shows a GUI where a particular sequence of events is required. This shows that a fuzzer is required to have both reasoning capability of the GUI context and a good state space exploration strategy to handle such complex GUIs, which are common in modern apps nowadays (challenge ❶*Reasoning About Complex GUI Layouts*).

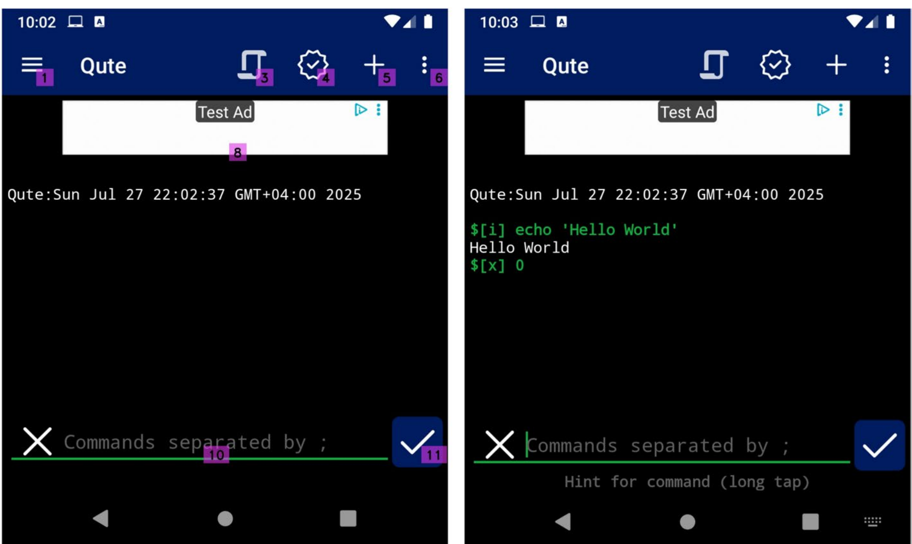


Fig. 1 An example of GUI challenging for typical GUI fuzzers (shown with VLM-Fuzz generated input)

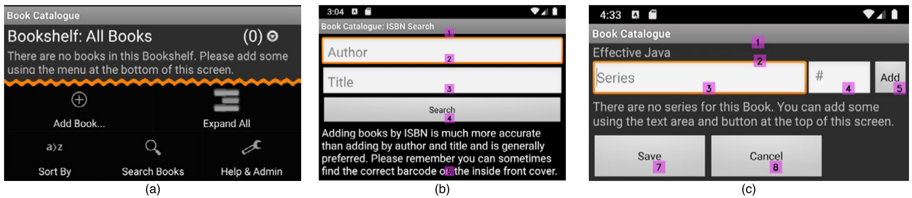


Fig. 2 Example, complex GUIs showing (a) a GUI overlaid on top of another GUI; (b) a GUI where complex, valid input is required; (c) a GUI where a particular sequence of events is required

2.2 Exploration Strategy

Random search is a simple yet often effective method where GUI events are generated randomly. It is employed by the widely-used Monkey test tool (Android 2024b). Though this strategy is useful when there is little knowledge about the input domain, often inefficient in terms of coverage, as it may repeatedly test similar or irrelevant inputs. *Depth First Search (DFS)* is a systematic method for exploring the state space by diving deep into each branch before backtracking. This strategy is used in Android fuzzers like GUIRipper (Amalfitano et al. 2012), MobiGUITAR (Amalfitano et al. 2014) and A3E (Azim and Neamtiu 2013). This strategy can also be inefficient without proper handling since DFS can get stuck in cyclic paths or in deep branches (challenge ❸ *Efficient and Coordinated State Exploration*).

Evolutionary-based search strategies, such as Genetic Algorithms (GA), simulate the process of natural evolution to optimize test generation. This strategy is used in Prev (Demissie et al. 2020). Although evolutionary search can be powerful in many contexts due to its adaptive nature in exploring the state space, it is often challenging to define appropriate fitness function for GUI testing. GUI interactions might lead to subtle changes in the app's state that are hard to capture in a fitness function (challenge ❶ *Reasoning About Complex GUI Layouts*). For instance, a GUI screen might appear to be the same as previous test outputs and hence, a fitness function may evaluate that this screen has been fully tested but it may still have subtle states that may not be captured by fitness functions.

Model-based search strategies, such as Stoat (Su et al. 2017) and *APE* (Choi et al. 2018) generate app event sequences according to models extracted from project artifacts such as source code, XML configuration files, and GUI runtime state. These approaches enable the representation of app behavior as a model, allowing for the application of various exploration strategies. However, complex widgets like dynamic pop-ups (e.g., Figure 2.a), commonly used in apps, present challenges in model construction, often resulting in an incomplete model (challenge ❶ *Reasoning About Complex GUI Layouts* and challenge ❸ *Efficient and Coordinated State Exploration*).

These observations motivate us to propose a hybrid approach that enhances a traditional DFS exploration strategy with two key innovations. To avoid getting stuck in cyclic paths, a common pitfall for DFS, our approach incorporates a novel state-visitation heuristic that goes beyond simple screen-hash comparison. It maintains a detailed record of the structure and properties of its interactive widgets along with unique their ID dynamically added to the GUI hierarchy XML via our custom dumper (discussed in Section 3). This allows it to differentiate between subtly different GUI states that might appear identical to a simpler crawler. Furthermore, our approach is augmented with vision-based reasoning AI model

to understand and interact with complex GUIs. This VLM-assisted mechanism, combined with our state-aware DFS, is designed to overcome the key challenges involved in the effective exploration of modern, complex graphical user interfaces.

3 Approach: The VLM-Fuzz Framework

Our approach introduces a novel method for fully automated Android GUI testing that leverages Vision Language Model (VLM) to achieve semantic understanding of GUIs. To explore the app, it utilizes a recursive Depth-First Search (DFS) exploration strategy. The strategy is described as follows:

- **Depth-First Traversal of GUI Components (Activities):** The algorithm follows a classic DFS pattern for navigating between different GUI components. It pursues a 'scenario' deep into the application's hierarchical Activity graph. It follows a path from a starting Activity until it can go no further, before backtracking to explore alternative paths from the last screen.
- **Recursive State-Exploration within each GUI component:** Before executing an action that would transition from the current component (Activity) to another, our algorithm recursively explores widgets and actions within the current Activity. This crawling strategy ensures comprehensive coverage of each individual Activity. While a pure DFS might miss this, our recursive state-exploration approach guarantees that we do not prematurely exit a screen before all of its interactive components have been tested. This is demonstrated in Fig. 6.

3.1 Overview

Figure 3 shows an overview of the workflow of VLM-FUZZ. Given the path of an APK file (A) as an input, VLM-FUZZ parses the AndroidManifest file and obtains the list of components (Activities, Services, and Broadcast Receivers) along with their intent-filters — a section that declares the capabilities of a given component (C). It then attempts to launch each Activity on the emulator/device (B). If the component is visible, VLM-FUZZ identifies and extracts the list of initial interactive GUI widgets from the current GUI hierarchy that is dumped using a custom-made tool that leverages the Accessibility Service. It then computes fuzzing time budgets for each Activity. Once the budget is computed, the main fuzzing loop starts. From the list of components, the GUI Analyzer (D) randomly selects a component and instantiates it on the emulator/device. Note that the component could also be a Service, a component that does not show a GUI. If it is a component with a GUI, VLM-FUZZ then captures the interactive GUI widgets along with the screenshot of the GUI. When a component is identified as complex (E), the VLM-Assisted Fuzzer (F) is instantiated to identify the right inputs for the different interactive GUI widgets on the screenshot with the help of a VLM engine. Otherwise, a Heuristic Based Fuzzer is initialized that uses heuristics to identifiy possible inputs (G). The Action Generator (H) then generates the appropriate events to the different widgets based on the VLM-assisted or heuristic based fuzzers results. Generated events are then sent to the GUI widgets by the executor (I).

Fig. 3 Approach overview

Unlike existing approaches that often rely on model-based or coverage-guided techniques, this method emulates human-like understanding of GUI interactions and emphasizes thorough state traversal, setting it apart from other approaches.

3.2 Test Budget Allocation

Algorithm 1 describes our main Executor algorithm. Our approach begins with a preliminary computation of component (i.e., Activity) budgets. To optimize testing efficiency and focus our efforts on the most feature-rich parts of an app, VLM-Fuzz employs an intelligent time budget allocation strategy. The total testing budget, B_{total}, is a user-configurable parameter specified in minutes (e.g., 60 minutes in this study). Instead of distributing the budget equally among components, we base our calculations on the number of interactive

widgets and menu items present in each GUI. Some simple GUIs, despite their apparent straightforwardness, could potentially trap the recursive testing approach in infinite loops. For instance, each interaction might add a new item to a ListView without improving overall coverage, leaving complex GUIs with insufficient testing time. This preliminary assessment is conducted by launching each component and enumerating the distinct *interactive* widgets and menu items present. The APK's AndroidManifest file is dumped using AAPT[1], and we use a custom-made parser to extract app components and their intent-filters. Then, each component is systematically launched by sending explicit Intents that satisfy the intent-filters (function *extractComponents()*). Upon successful component launch (startComponent(), Algorithm 1, line 5), the component's GUI hierarchy XML is retrieved using a custom-made GUI hierarchy XML dumper that leverages Accessibility Service[2] and the component budget is computed (function *getComponentsBudget()*) as follows: For each component *i*, we count the total number of interactive widgets w_i and the number of widgets accessible via a menu m_i. The complexity score for component *i* is then computed as a weighted sum. Based on our empirical observation, menu interactions often reveal significant new functionality, and therefore we apply a higher weight to menu-accessible widgets. The budget allocation ratio for component *i*, $BudgetRatio_i$, is then computed as follows:

$$BudgetRatio_i = \frac{w_i + 4 \times m_i}{\sum_{j \in S}(w_j + 4 \times m_j)} \tag{1}$$

where *S* is the set of all identified accessible app components. The $4\times$ multiplier for menu items is a heuristic based on the observation that a single menu action often reveals a new GUI state containing at least four new interactive elements (e.g., in a settings dialog). We chose 4 as a conservative estimate for the minimum number of interactive widgets a new state must have to be considered functionally significant. The final time budget for component *i* is then $B_i = BudgetRatio_i \times B_{total}$. This approach ensures that more complex components with several interactive GUI elements receive proportionally more testing time, while simpler components with fewer GUI elements are allocated less time, ensuring comprehensive coverage across varying GUI complexities.

Algorithm 1 Executor().

Input: Path to the APK to be tested (*apkPath*).
Allocated budget (mins) for the entire test (*budget*).
1: **function** EXECUTOR(*apkPath*, *budget*)
2: *componentsList* ← extractComponents()
3: *componentBudgetList* ← getComponentsBudget(*componentsList*, *budget*)
4: **for** *component* ∈ *componentsList* **do**
5: *startComponent*(*component*)// Could also be a Service or BroadcastReciever
6: *budget* ← *componentBudgetList*[*component*]
7: **UNTIL**(**budget**) : *UI_Analyzer*(*component*, *UIStack*, *nonIgnoreComponentList*)
8: **end for**
9: **end function**

---

[1] https://developer.android.com/tools/aapt2

[2] https://developer.android.com/reference/android/accessibilityservice/AccessibilityService

3.3 GUI State Exploration

Once the proportional budget is computed, the algorithm explores the Android app's GUI state space. It uses a heuristic-based recursive depth-first search. Similar to the preliminary assessment phase, it launches each component by sending explicit Intents and dumps the corresponding GUI hierarchy (function *startComponent()*). We start each component via explicit Intent to ensure reliable testing, as components may not be directly accessible from the main Activity. Additionally, components can exhibit different behaviors based on the Intent's data and action, allowing us to test various scenarios independently. The algorithm then parses the XML output to identify interactive widgets on the currently visible portion of the component. A mapping of widget to action is then generated for each identified interactive widget (e.g., a tap action for a button). The subsequent step involves interacting with these items while monitoring for GUI state change. The recursive process is repeated if GUI state change is detected. A GUI state change event is fired when, (i) on the same component, new widgets appear (e.g., after a scroll action or a new list item is added), (ii) a popup overlay is shown on the same component (Fig. 2.a), or (iii) a component switch occurs. The component's GUI is then explored until its budget is exhausted (function *GUI_Analizer()*). We developed a recursive GUI state explorer that leverages Vision Language Models (VLMs) on complex GUIs. VLMs integrate computer vision with natural language processing, enabling them to interpret images and text together. This dual capability allows them to handle tasks requiring both visual understanding and linguistic comprehension. In this work, we use OpenAI's GPT-4o Vision, a state-of-the-art multimodal AI model (OpenAI 2024). Before selecting GPT-4o, we did a preliminary experiment with several VLMs, including GPT-4o, Qwen2-VL-7b, Gemini-1.5, Pixtral-12b, and Claude 3.5 Sonnet, via HuggingFace's VisionArena (Hugging Face 2024). In our preliminary experiment, the VLMs were presented with 10 screenshots of GUIs randomly extracted from our benchmark apps and were prompted to generate a sequence of events to exercise the presented GUIs using the prompt shown in Fig. 4. The default configuration of the VLMs was used. To provide a quantitative basis for our comparison, we defined a metric we term Task Completion Rate. A 'task' was defined as a multi-step GUI goal (e.g., filling a form with correct data). The rate measures the percentage of tasks for which the VLM produced a syntactically (i.e., can be parsed by our tool) and semantically correct and fully executable sequence of actions as per the provided one-shot example. One of the authors evaluated each VLM on a set of 10 representative GUI tasks. We acknowledge that having a single evaluator introduces a potential for subjectivity. However, this preliminary study's goal was not to serve as a rigorous, standalone evaluation of VLMs, but rather to select the most suitable model for VLM-Fuzz based on a clear performance gap. The strong result from GPT-4o motivated its selection for our main experiments. As shown in Table 1, GPT-4o achieved the most accurate results. GPT-4o likely performed better due to its enhanced ability to process visual information from GUI screenshots and understand spatial relationships between interface elements, combined with its strong capabilities in generating sequential instructions and understanding GUI interaction patterns. As an example, for the GUI screen given in Fig. 2.c, the sequence of events generated by a few VLMs are shown in Table 2 where GPT-4o produced the most accurate sequence of events. We made all these experiment results — the sequence of events generated by various VLMs for the 10 GUI screenshots — available in our replication package (VLM-Fuzz 2025).

Prompt snippet

Given the screenshot of an app with interactive GUI elements labeled with numbers, provide an analysis and interaction strategy using the following functions to interact with the app:

1. tap(element: int): Taps the specified GUI element by its numeric label.
2. long_press(element: int): Long presses the specified GUI element by its numeric label.
3. swipe(element: int, direction: str, dist: str): Swipes on the specified GUI element in the given direction (up, down, left, right) for a given distance (short, medium, long).
4. input(element: int, text_input: str): Inputs text into the specified text box by its numeric label.
5. tap(BACK): Performs the BACK action.
6. tap(ENTER): Performs the ENTER action.
7. scroll(UP): Scrolls up the screen.
8. scroll(DOWN): Scrolls down the screen.

The response should be structured as follows each on a single line:
...

Response snippet

Process: To search for a book, I should input the author and title into the respective fields and then tap the "Search" button.
Steps: [tap(2); input(2, "J.K. Rowling"); tap(3); input(3, "Harry Potter"); tap(4);]
Summary: I entered "J.K. Rowling" into the author field, "Harry Potter" into the title field, and tapped the "Search" button to perform the search.

Fig. 4 Example prompt snippet and VLM response

Table 1 Task Completion Rate of Different VLMs

VLM Model	Task Completion Rate
GPT-4o	10/10 (100%)
Claude 3.5 Sonnet	8/10 (80%)
Gemini 1.5 Flash	6/10 (60%)
Qwen2-VL-7B	3/10 (30%)
minicpm-llama3-v	0/10 (0%)
pixtral	1/10 (10%)

3.4 GUI Analyzer

Algorithm 2 outlines our GUI Analyzer approach. The GUI analyzer is the core part of our approach that recursively analyzes the current GUI, dumping its GUI hierarchy, extracting interactive GUI widgets, performing interactions, and monitoring GUI state changes. It takes three inputs: the currently visible component's details, a stack tracking previously visited components, and a list of relevant external components (such as Android framework dialogs for permission requests) that, while not part of the app under test, are critical to restore app functionality. Moreover, the algorithm implements several heuristics to maximize new state discovery. We maintain a call stack and set a depth threshold τ to limit the number of times a GUI component can be analyzed in a *single instance* (e.g., **Activity A**$\rightarrow$

Table 2 Sample sequence of events generated by VLMs for the GUI screen in Fig. 2.c

	VLM	Sequence of Events
Commercial	GPT-4o	[tap(3); input(3, "Series 1"); tap(4); input(4, "1"); tap(5); tap(7);]
Commercial	Claude3.5 Sonnet	[input(2, "Harry Potter"); input(4, "1"); tap(5); tap(7);]
Commercial	Gemini-1.5-flash	[tap(2); input(2, "The Lord of the Rings"); tap(4); input(3, "The Hobbit"); tap(7);]
Open source	Qwen2-VL-7B	[tap(2); input(2, \"Series Name\");]
Open source	minicpm-llama3-v	[tap(3); input(3, \"The Lord of the Rings\"); tap(5);]
Open source	pixtral	[tap(3); input(3, "Harry Potter"); tap(5);]

Activity B→Activity C→**Activity A**), where our experiments showed that $\tau = 2$ achieves good test coverage while staying within budget of 60 minutes. We found that a value of $\tau = 1$ was sometimes insufficient to fully explore complex screens with nested states, while a value of $\tau \geq 3$ provided diminishing returns in code coverage while significantly increasing the risk of exceeding the time budget on simpler screens. The overall testing budget of 60 minutes per app is a standard time-box used in comparable state-of-the-art studies (Mao et al. 2016; Choi et al. 2013; Su et al. 2021; Lai and Rubin 2019). While we needed to stay in this constrained setting for the sake of comparison with the state-of-the-art, it is noteworthy that internal, unconstrained experiments indicated a higher τ values (e.g., 4) and longer testing durations (120 mins) in turn resulted in increased test coverage. Therefore, once a component has been analyzed twice, any further state changes in that GUI element will not trigger a new analysis allowing us overcome the challenge ❸*Efficient and Coordinated State Exploration*, avoiding getting stuck in repetitive GUI states. When a component is ready for analysis and contains a text editor widget (e.g., *EditText*), the algorithm first tries to use the VLM-assisted action performer (via the *performVisionActions()* function, Algorithm 2, line 21). This function begins by capturing and labeling a screenshot of the current GUI (Algorithm 3, lines 2-3) and performs a REST API request (line 4) to the configured VLM service provider (i.e., GPT-4o). It then iterates through the actions identified by the VLM. For example, text input is sent at line 17 or a tap action is performed at line 20. The VLM is prioritized because it can both generate contextually relevant text input and determine appropriate sequences of GUI events to accomplish tasks presented on the screen. If this function successfully explores all widgets, the test is considered complete. Considering that the VLM's output may sometimes be ambiguous or imperfect, we have implemented a robust fallback mechanism. This mechanism cross-checks the suggested actions against the static properties of the GUI elements. If the VLM-assisted approach fails to achieve complete GUI coverage or encounters screens without text editors, the algorithm falls back to the non-VLM assisted action performer (via the *performNonVisionActions()* function, Algorithm 2, line 22), which provides comparable exploration effectiveness for non-text interactions while being cheap costwise. It iterates through the list of widgets and performs the corresponding inferred action. For example, a text input is sent to an *EditText* widget at

11 or tab action is performed to a clickable widget at line 14. Functions *performVisionActions()* and *performNonVisionActions()* are described in the Appendix A. Our hybrid input generation strategy addresses the challenge of ❷*Generating Required Inputs and Events*. The VLM excels at reasoning about complex GUIs to produce human-like inputs, while the heuristic-based component ensures that simpler or non-visual GUI elements are also systematically tested.

Algorithm 2 GUI Analyzer.

```
Input: Current visible component (currentComponent).
       GUI component visit stack (UIStack).
       List of non-AUT[3] components not to ignore (nonIgnoreComponentList).
1: function UI_ANALYZER(currentComponent, UIStack, nonIgnoreComponentList)
2:    currentComponentDetails ← getCurrentComponentDetails(currentComponent)
3:    if visitCount(currentComponent) > τ then
4:       return
5:    end if
6:    if currentComponent ∈ UIStack then
7:       if not UIItemsChanged() then
8:          return
9:       end if
10:   end if
11:   if not currentComponent ∈ AUT then
12:      press_back()// AUT is not in foreground, try restoring it by pressing back
13:      return
14:   end if
15:   if not currentComponent ∈ UIStack then
16:      UIStack.push(currentComponent)
17:   else if UIStack[lastEntry] == currentComponent then
18:      return
19:   end if
20:   widgetsList ← currentComponent.getWidgets()
21:   if not performVisionActions(widgetsList) then
22:      performNonVisionActions(widgetsList))
23:   end if
24:   UIStack.pop()
25:   return
26: end function
```

3.4.1 Complex GUI: GUI with text input widgets

We define a GUI screen as *complex* if it includes one or more text input widgets. This definition is a cornerstone of our hybrid, cost-effective testing strategy, which selectively utilizes a VLM to balance performance with deep analytical capability. We identify these screens as complex because they represent a significant weakness in traditional fuzzing; they often require contextually appropriate data (e.g., a valid email address, a relevant book title) rather than random strings, and a precise sequence of operations (e.g., filling multiple fields before tapping 'submit') to successfully navigate to subsequent states. By focusing the VLM's powerful input-generation capabilities on these specific scenarios, we maximize its impact where it is most needed, avoiding the expensive and slow invocation on every screen. If the GUI contains text widgets, a screenshot is taken and each interactive widget is labelled with numbers (e.g., Figure 2.b). The labelled screenshot along with a one-shot prompt is then fed to the VLM. The structured VLM output is then parsed and

actions are mapped to their corresponding labeled widgets, and the actions are performed following the sequences suggested by the VLM. This approach helps us overcome the first challenge, ❶*Reasoning About Complex GUI Layouts*, where an on-demand invocation of VLM-assisted analysis generates reasonable text input and seqeunce of action for the given GUI. In case of a text input event, the generated text is sent to the target widget, and any resulting state changes on the widget are monitored. If no state changes occur, for example, because alphanumeric text was generated when only numeric input is accepted (e.g., widget 4 on Fig. 2.c), an alternative input (e.g., a numeric value) is generated and sent. Since datatype is unknown at runtime, this verification is performed by comparing the *text* property of the current widget with the generated text. A mismatch indicates that the generated text was rejected. Figure 4 shows a snippet of a prompt and the corresponding VLM response. The output of the VLM is sensitive to the precise wording of the prompt and its configuration. To ensure consistent and high-quality outputs, we undertook a one-time, offline prompt engineering process during the development of our tool. This process involved an iterative refinement process, starting with a basic instruction to generate event sequences on a small, representative set of GUI screenshots to standardize the prompt structure, ensuring consistent outputs. Finally, we leveraged GPT-4o to reverse-engineer and formalize the optimal prompt structure that consistently produced the desired output format and quality. The resulting optimal prompt, shown in Fig. 4, was then fixed and used without any modification for all experiments described in this paper. In this example, we used the GUI in Fig. 2.b where author name and title of a book by the author is expected to be filled in before tapping the *Search* button. The VLM generates the steps and inputs necessary to complete the tasks on that GUI.

If there are any unprocessed widgets remaining after performing the VLM suggested actions (e.g., the VLM missed an action for an interactive widget), a non-VLM based state exploration is launched. If any of the unprocessed widgets is a text input box, attributes associated to the widget (e.g., widget ID *"authorname"*, or placeholder text *"Author"*) are extracted and structured as a JSON object and fed into an LLM to predict an appropriate text input. If this fails, a random text or number is generated. The remaining actions are inferred from the GUI hierarchy and performed following the heuristic described in Section 3.4.2.

After all possible interactions are complete, the subsequent interactive operation is the Menu tap, a function that activates the menu if the GUI implements one. An approach solely based on VLM/LLM would miss the exploration of such hidden interactions as in some cases there is no indicator of existence of menus either in the visible GUI or in the GUI hierarchy XML. VLM-Fuzz taps on the Menu button to see whether the app implements menus and monitors GUI change. If a GUI change occurs, i.e., a pop-up menu is overlaid on the current screen (see Fig. 2.a), the recursive process restarts for the current GUI. While traditional Menu-based navigation is less common in modern Android apps, our testing approach must remain comprehensive to ensure fair comparisons with existing tools on the selected benchmark. The state-of-the-art testing frameworks include Menu interaction capabilities, and excluding Menu testing would compromise VLM-FUZZ's performance evaluation on benchmark applications that utilize these elements.

Figure 2.c shows an example GUI that requires a specific sequence of actions that might be difficult to discover through random exploration alone. While traditional heuristics could help determine the correct order of interactions, a VLM would be particularly well-

suited for this task. In this specific example, the VLM successfully identified the correct sequence of user actions: *[tap(3); input(3, "Java Series"); tap(4); input(4, "1"); tap(5); tap(7);]*.

Note that if a GUI change occurs before all the possible states on that GUI are explored, the events that led to the last state are replayed (except the last action that caused the GUI change), and the remaining actions are performed. If replaying does not bring the state back, the GUI is considered permanently altered and the exploration is ended for that GUI. This process is described in details in Section 3.5.

3.4.2 Simple GUI: GUI Without Text Input Widgets

In the cases where no text input is necessary (e.g., Figure 2.a), we perform the actions based on the following orders. First, clickable (tappable) actions are performed followed by scroll actions. If the component implements a menu, the interaction with menu items is followed. The following heuristics are applied in order to maximize the new state discovery.

Clickable Widgets Clickable (tappable) widgets are grouped into three based on their text sentiment, namely, (i) neutral, (ii) positive (e.g., "Save" or "Open") or (iii) negative (e.g., "Exit", "Back" or "Cancel"). The rationale behind this is to perform the actions that could potentially lead to navigation away from the current GUI as the last step, ensuring the majority of the actions are completed first. Each group is shuffled and interactions are performed in the order (i) → (ii) → (iii). Moreover, if there are tappable widgets appearing on the same Y-axis of the screen (e.g., **Yes | No | Cancel**, similar to buttons 7 & 8 in Fig. 2.c), the tap action on these widgets will be performed at the end of all other interactions. Note that if a widget is identified as being both ‘clickable’ and ‘long-clickable’, we generate both events and add them to the queue of actions to be performed on that widget. Our state exploration strategy is designed to be exhaustive. If performing the click action first triggers a state transition, the ‘long-click’ event remains marked as unexplored for that state. After the exploration of the new state is complete, our backtracking mechanism attempts to return the fuzzer to the previous state specifically to execute any remaining unexplored events, such as the long-click.

Progress Bars After performing an action, most approaches wait for an idle state before evaluating the state change. In cases of progress indicators (e.g., a progress bar widget), though an idle state is reached (e.g., no new widgets are added to the GUI within the idle period), an effective state exploration approach should wait until the progress bar finishes or a given timeout greater than idle period is reached. In fact, if a "*loading...*" indicator appears similar to the example shown in Fig. 5, a human would naturally wait for it to finish before navigating away from this GUI. Our approach works the same way. When it detects a visible progress indicator (via the XML GUI hierarchy), it pauses the current analysis and monitors the GUI for up to 60 seconds to detect any changes. If the GUI changes (e.g., new GUI or progress indicator is hidden) in this monitored period, a new instance of the analyzer is launched for the new state, otherwise the current analyzer is resumed. This approach ensures we do not miss important GUI states that appear after loading screens or progress indicators finish. In this example, once the search completes, a GUI with a form opens where all the fields are already filled in with the search results.

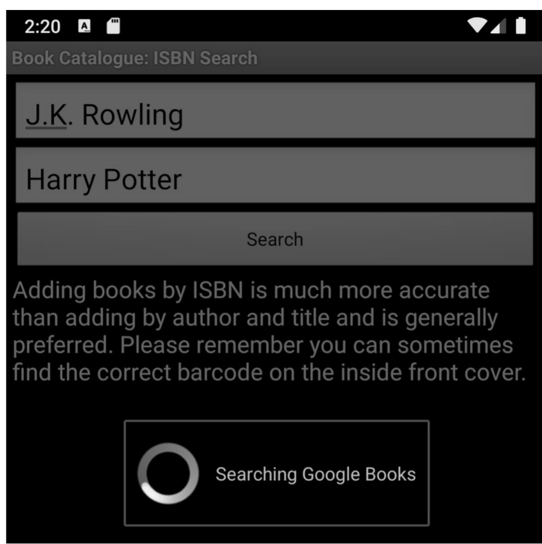


Fig. 5 A loading progress indicator

3.5 Transition Record

A sophisticated aspect of our state exploration logic is its ability to differentiate between two fundamentally different types of state transitions: inter-Activity navigation (traversing to a new screen) and intra-Activity changes (the appearance of transient GUI elements like pop-ups or drop-down menus (Spinners)). To distinguish a pop-up from a simple widget change within the same Activity, our tool employs a concrete heuristic. After performing an action, if no Activity change happens, the tool checks whether an intra-Activity GUI change occurred by performing a diff on the set of interactive GUI elements before and after the action is performed. If so, it then checks whether the new state is a pop-up using a dimensional comparison. A new GUI state is classified as a popup if its layout does not fully cover the screen. This is determined by comparing the coordinates of the new layout's bounding box against the device's screen dimensions. Specifically, a new state is considered a pop-up if the gap between any edge of its bounding box and the corresponding edge of the physical screen is greater than a threshold of 150 pixels, a value determined empirically from our experiments. While our recursive algorithm treats any new GUI hierarchy as a new state, their handling is distinct to prevent exploration loops and ensure efficiency. If a state is identified as a dialog, pop-up, or a Spinner's drop-down menu, it is marked as a 'transient state'. Most importantly, transitioning to a transient state does not increment the visitation counter (τ, discussed in Section 3.4) for the underlying host Activity (where the popup is overlayed on). This prevents the tool from prematurely terminating the exploration of an Activity that contains many pop-ups or pop-ups with multiple interactive widgets. The exploration logic then adapts based on the state type:

- **For Pop-ups and Dialogs:** The tool fully explores all actions on the pop-up. When an action closes it (e.g., tapping 'OK' or 'Cancel'), the fuzzer returns to the host Activity. It then performs a diff on the host's GUI hierarchy. If the host GUI was modified by the pop-up's action, this new layout is treated as a new state to be explored. If not, exploration

simply resumes from the widget that triggered the pop-up. Figure 7 shows an example state exploration involving several state transitions between a host Activity and popups.
- **For Drop-down Menus (Spinners):** When a Spinner is activated, its menu items are treated as the foreground GUI and the state is flagged as a 'spinner'. In this specific mode, the tool's heuristic changes: it randomly selects and clicks on only one menu item. This is because any tap on a Spinner item will both select a value and immediately close the menu, making it impossible to explore other options without re-opening the Spinner. To avoid this redundant looping, the tool performs a single selection without backtracking and then returns to the host Activity, where exploration continues from the Spinner widget without flagging a new state transition.

This state-typing mechanism allows VLM-Fuzz to robustly handle the different levels of abstraction in the GUIs without getting stuck in a loop. If a state is not transient, VLM-Fuzz performs the normal state exploration.

Typically, tap actions trigger GUI changes by either displaying a popup overlay or navigating to a different screen. For instance, tapping the Menu button on the device might reveal a popup menu that appears on top of the current GUI, as illustrated in Fig. 2.a. Despite appearing as an overlay on the current GUI, this popup maintains its own unique GUI hierarchy XML, thus qualifying as a distinct GUI state. The GUI in Fig. 2.a presents five interactive elements. Tapping on any of the items (e.g., *Add Book...*), may navigate the analyzer away from the current view, possibly even exiting the app. In such cases, interaction with the remaining four widgets would be missed. To address this issue, VLM-Fuzz records the exact sequence of state transitions, allowing it to replay these actions later to return to the same GUI state and continue testing any remaining interactive elements that were not yet explored. In this context, a state $s \in S$, where S is the set of all possible GUI states of the app under test (AUT) is a tuple defined as follows:

$$s = (W, P)$$

where:

- W is the set of GUI elements (widgets) in that GUI state
- P is the set of properties for each element

A transition T from state $s1$ to $s2$ is then defined as:

$$T : s1(W_1, P_1) \xrightarrow{a} s2(W_2, P_2)$$

where $a \in A$ is an action performed in state $s1$ and A is the set of supported actions (See Section 3.6).

An example transition record is shown in Fig. 6 for the GUI in Fig. 2.c where a component is started (S1) and a tap on widget 3 focuses the widget. A text input "JavaSeries" is inserted in widget 3 (S2) and widget 4 is tapped to focus. The text "1" is then inserted (S3) followed by taps on widget 5 and 7, respectively. If any subsequent tap action causes the app to change GUI before all the interactive widgets are explored, once the current exploration finishes, the algorithm attempts to replay the transition

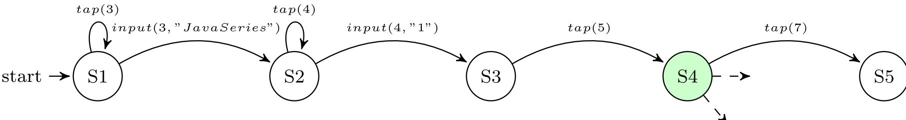


Fig. 6 Example state transition

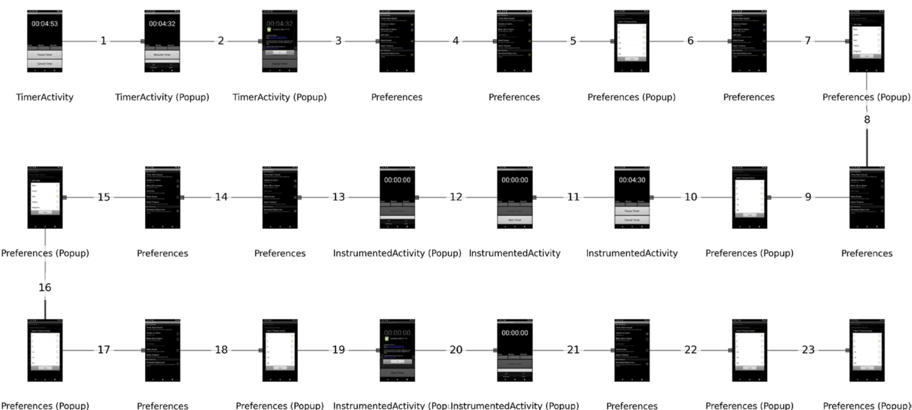


Fig. 7 Sample state space exploration of an app (`everythingandroid.timer`)

record to come back to the last state, in this case $S4$ (backtracking), skipping the last action (i.e., tap(7)) that caused the change and explores the remaining interactive widgets.

As an example, Figure 7 shows an excerpt of state space exploration of an app we experimented with. It shows that our approach was able to explore the possible states in the same activity screen despite the dynamic pop-ups (e.g., transitions 6 and 8) and escape from the current GUI once a recursion without any new discovery state is detected (e.g., transitions 3 and 10).

3.6 Implementation

VLM-Fuzz is implemented using Python while the companion server app that is used to dump the GUI hierarchy is developed in Java, and they communicate via broadcast. The server generates and dumps the GUI layout hierarchy in a format and location identical to GUIAutomator (Android 2024a), making it a suitable replacement for GUIAutomator. VLM-Fuzz runs both on emulators and real devices. It requires neither instrumentation nor modification to the app under test, Android device, or Android operating system. It only requires the device to be connected to the computer running VLM-Fuzz tool via Android Debug Bridge (ADB). Apart from Android SDK tools such as ADB and Android Asset Packaging Tool (AAPT), VLM-Fuzz does not have other dependencies.

Supported Events VLM-Fuzz supports generating the following events that can be simulated using ADB:

A. *GUI events (actions)*: click, long-click, scroll, text-input, menu-tap, app switch (switching between apps) and screen-rotate. VLM is used to generate these events.
B. *Intents*: when launching components, VLM-FUZZ uses the defined intent-filters and attempts to generate valid Intent. If there are multiple intent-filters, a random one is picked every time the component is launched. This is crucial for testing components like deep- link handlers or activities invoked by inter-app interaction, which are often unreachable through standard user navigation from the main launcher.
C. *System broadcast Intents*: valid broadcast Intents are generated based on the app's intent-filter definition. For example, if an app defines the `TIMEZONE_CHANGED` system broadcast Intent, the broadcast is sent to the app with `TIMEZONE` and `TIME_PREF` extra fields populated. VLM-FUZZ supports 187 system broadcast Intents. These system broadcast Intents are constructed once from the Android documentation[3] and stored in file, where example inputs are extracted with the help of an LLM. Whenever a BroadcastReceiver component defines system broadcast Intent, the file is consulted to generate the right Intent (i.e., action, extras, etc) for the component. The file is publicly available in our tool repository to enable researchers and developers to extend existing fuzzers with system broadcast capabilities.

Approaches developed on top of Android Monkey such as *APE* can generate more GUI events such as drag and pinch-zoom or volume level.We recognize that this engineering limitation may restrict our tool's ability to fully exercise all app functionalities, potentially impacting bug detection in scenarios involving multi-touch interactions. In future work, we will extend our approach to incorporate a wider range of actions, including drag, pinch, and other multi-touch actions, to enhance test coverage and better simulate real-world user behavior.

4 Empirical Evaluation

In this section, we present the design and results of our experiments to assess the efficacy of VLM-FUZZ. Our experiments are designed to answer the following Research Questions (RQs):

- **RQ1. Code Coverage.** How effective is VLM-FUZZ in terms of code coverage? How does VLM-FUZZ compare against the state-of-the-arts, namely *DeepGUI* and *APE* ?
- **RQ2. Ablation Study.** How useful is VLM at determining relevant test inputs? And what is the cost of using VLM? How useful is multi-entry testing in achieving a good code coverage?
- **RQ3. Bugs Detection Capability.** How is the effectiveness of VLM-FUZZ in detecting bugs in real-world apps?

4.1 Benchmark Apps

It is important to note that VLM-FUZZ operates in a black-box setting and does not require any modification or instrumentation of the app under test. However, to empirically measure

[3] https://developer.android.com/about/versions/11/reference/broadcast-intents-30

code coverage for our evaluation, we utilized the apps from the AndroTest benchmark suite (Choudhary et al. 2015), which were already pre-instrumented by the benchmark's original authors. The instrumentation was performed using the Emma (Rubtsov 2006) code coverage tool. The Emma tool works by instrumenting the app's compiled bytecode with probes before the final .apk is built. During test execution, these probes record which parts of the code are executed. It consists of 66 instrumented open-source real-world apps, including popular apps like WordPress and Wikipedia. Seven apps were excluded due to crashes and incompatibility issues with *DeepGUI* tool. This benchmark was used for comparing various Andriod testing tools in Choudhary et al. (2015), which includes Dynodroid (a random fuzzing tool) (Machiry et al. 2013), GUIRipper (a model-based depth first search tool) (Amalfitano et al. 2012), and EvoDroid (an evolutionary algorithm-based fuzzing tool) (Mahmood et al. 2014). It is also used in the experiments of Humanoid (Li et al. 2019) and *DeepGUI* (YazdaniBanafsheDaragh and Malek 2021). Figure 8 shows the statistics of the benchmark apps. It shows that the sizes of the apps range from small (100 LOCs) to large (22 kLOCs). Median size is 1060 LOCs. Number of methods range from 449 to 5329. Number of classes range from 26 to 187. Expanding the dataset is non-trivial because our current benchmarking apps were custom instrumented and built from source to enable precise measurement of class, method, and line coverage. This instrumentation requires access to the source code and modifications to the build process, making it challenging to include a broader set of apps, particularly closed-source or third-party applications that lack this level of access and control. While this instrumentation was necessary for measurement purposes, our approach does not require source code to test real-world apps.

4.2 Comparison with State-of-the-Art

For RQ1, we compare VLM-FUZZ against two state-of-the-art Android fuzzing approaches — *APE* (Gu et al. 2019) and *DeepGUI* (YazdaniBanafsheDaragh and Malek 2021). We selected these tools as our primary baselines because they represent the current state-of-the-art and have already demonstrated superior performance over standard random-based UI fuzzers such as Android Monkey (Android 2024b) in prior studies using the same AndroTest benchmark suite that we employ. For instance, the original *APE* paper reported a 17–22%

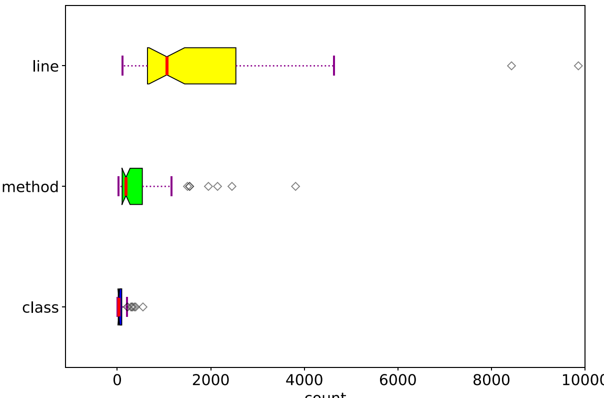


Fig. 8 Statistics of the benchmark apps

higher method coverage over other three tools including Monkey on this benchmark. Similarly, *DeepGUI* was also shown to outperform Monkey. Following previous works (Choi et al. 2013; Mao et al. 2016; Lai and Rubin 2019; Su et al. 2021), we run all the fuzzers for 60 minutes each. To ensure reliability, we repeated this entire testing process five times. We then compare the effectiveness of the approaches by analyzing their average coverage results.

Multiple-Entry Testing A key design choice in VLM-Fuzz is its ability to perform multiple-entry testing by directly launching any public component defined in the app's manifest. This contrasts with traditional single-entry fuzzers like *APE* , which begin from the main launcher activity and can only explore states reachable through user navigation. We acknowledge that this design gives VLM-Fuzz an advantage in reaching certain parts of the app's code better than the state-of-the-art. However, we argue that this is a deliberate feature designed to overcome a well-known limitation of single-entry UI fuzzers. Many Android apps contain components that are not reachable via the main entry-point but are instead triggered by external Intents, such as inter-app communication, deep-links or system broadcasts events. These activities contain crucial application logic that single-entry UI fuzzers will never test. By identifying and crafting valid Intents to launch these components, VLM-Fuzz provides a more comprehensive and realistic test that better reflects the full range of ways an app can be invoked in a real-world environment. In RQ2, we compare the single-entry version of our tool against *APE* and discuss the contribution of our algorithm innovations.

There are other approaches, namely Stoat (Su et al. 2017), and Sapienz (Mao et al. 2016), which are widely used as benchmarks in the literature. We do not compare VLM-Fuzz with them because *APE* and *DeepGUI* have already compared theirs with those approaches and outperformed them in their experiments. In addition, Choudhary et al.'s empirical study (Choudhary et al. 2015) reported that Monkey (Android 2024b) achieved the best code coverage among other fuzzing tools like Dynodroid (Machiry et al. 2013), GUIRipper (Amalfitano et al. 2012), ACTEve (Anand et al. 2012), SwiftHand (Choi et al. 2013), EvoDroid (Mahmood et al. 2014), etc. We note that *DeepGUI* requires data collection (collection of GUI images) and model training. It means that the tool may not work well when testing the app that has a GUI model different from the training data. Given that our goal is to advance the state-of-the-art, we believe that demonstrating superiority over *APE* and *DeepGUI* is a more stringent and informative evaluation. Therefore, a new direct comparison with other tools was deemed unnecessary, as outperforming *APE* and *DeepGUI* indirectly suggests a significant improvement over them.

One state-of-the-art approach most closely related to our approach is GPTDroid (Liu et al. 2024). However, the original implementation referenced in their paper was no longer available in the specified repository at the time of our evaluation. Our attempts to contact the authors for access to their implementation were unsuccessful. While we located an implementation sharing the same name on GitHub, we cannot verify if this represents the original work described in their paper. Therefore, we opted to provide a comparison of our approach and GPTDroid in related work (Section 5).

4.3 Metric

Coverage We use the standard coverage criteria — class coverage, method coverage, and line coverage — to assess the capability of the fuzzers. We shall also use the time vs the coverage as a metric for measuring the efficiency of the tools.

4.4 Setup

VLM-FUZZ runs both on emulators and real devices. It requires neither instrumentation nor modification to the app under test, Android device, or Android operating system. It only requires the device to be connected to the computer running VLM-FUZZ tool via Android Debug Bridge (ADB). Apart from Android SDK tools such as ADB and Android Asset Packaging Tool (AAPT), VLM-FUZZ does not have other dependencies.

We ran VLM-FUZZ on an Android Google Pixel 2 emulator with 4GB of RAM running Android 9 (API level 28) to ensure compatibility with *APE* in our experiments. However, VLM-FUZZ was tested on Android versions 4.1 - 12 and it operates seamlessly as a black-box testing tool out-of-the-box. We ran *DeepGUI* on Android 2.3.3 (API level 10). We monitored the class, method, and line coverage of the app under test every 60 seconds using Emma tool (Rubtsov 2006).

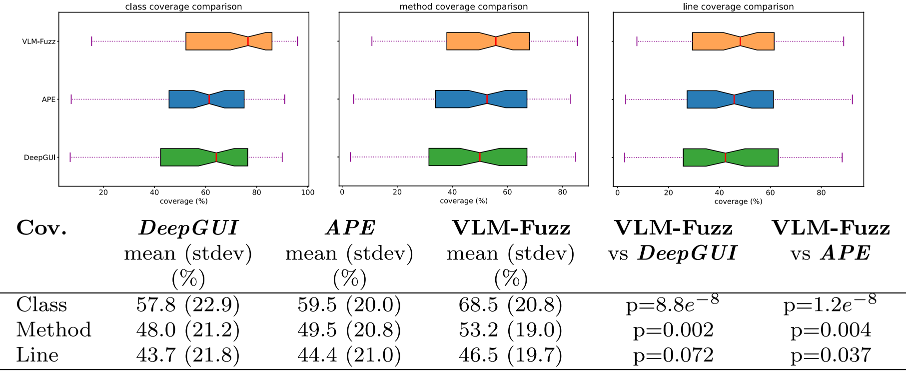
All emulators were run on a Ubuntu Desktop Intel Core i9 64GB RAM @ 3.2 GHz, 64GB. We ran all tools with their default configurations.

4.5 Results

This section discusses the experiment results addressing the research questions.

4.5.1 RQ1: Code Coverage

Figure 9 shows the boxplots of covered classes, covered methods, and covered lines of VLM-FUZZ and the baselines. The figure additionally contrasts descriptive statistics (mean and standard deviation) of these three coverage metrics for VLM-FUZZ and for the baselines. In general, VLM-FUZZ outperforms against the baselines across the three coverage



Cov.	*DeepGUI* mean (stdev) (%)	*APE* mean (stdev) (%)	VLM-Fuzz mean (stdev) (%)	VLM-Fuzz vs *DeepGUI*	VLM-Fuzz vs *APE*
Class	57.8 (22.9)	59.5 (20.0)	68.5 (20.8)	p=$8.8e^{-8}$	p=$1.2e^{-8}$
Method	48.0 (21.2)	49.5 (20.8)	53.2 (19.0)	p=0.002	p=0.004
Line	43.7 (21.8)	44.4 (21.0)	46.5 (19.7)	p=0.072	p=0.037

Fig. 9 Comparison of Class, Method, and Line Coverage, averaged across 5 runs (RQ1)

criteria. On average, VLM-Fuzz achieves class coverage of 68.5% , method coverage of 53.2% , and line coverage of 46.5% , across the 59 apps. VLM-Fuzz also achieves the lowest standard deviation in terms of method and line coverages, implying it is the most consistent across different apps. VLM-Fuzz outperforms the baselines in terms of coverage. On average, compared to *DeepGUI* and *APE* , it achieved 10.7% and 9.0% relative improvements in terms of class coverage, 5.2% and 3.7% relative improvements in terms of method coverage, and 2.8% and 2.1% relative improvements in terms of line coverage, respectively. Table 3 shows the number of apps for which a given approach achieves a better coverage. It shows that VLM-Fuzz performed better than both *DeepGUI* and *APE* on 53 and 49 out of 59 apps, respectively, in terms of class coverage; it performed better than both *DeepGUI* and *APE* on 48 and 40 out of 59 apps, respectively, in terms of method coverage; it performed better than *DeepGUI* and *APE* on 43 and 35 out of 59 apps, respectively, in terms of line coverage.

We also applied statistical tests to assess if these observed trends are statistically significant. As common practice, we assume significance at 95% confidence level ($\alpha = 0.05$), so when a statistical test has a p-value < 0.05 we reject the test null-hypothesis H_0 and formulate an alternative hypothesis H_1. First, we apply the Shapiro-Wilk test to the difference in coverage results between VLM-Fuzz and *DeepGUI* , and between VLM-Fuzz and *APE*. Considering that the two p-values are not < 0.05 (they are 0.8 and above), we cannot reject the null-hypotheses H_0 that for the Shapiro-Wilk Test claims that data are not normally distributed. So, we can assume that the data are normally distributed.

We then applied the paired t-tests comparing VLM-Fuzz vs *DeepGUI* and VLM-Fuzz vs *APE*. We used t-tests instead of non-parametric tests like Wilcoxon signed rank test, because t-test has higher statistical power, and it requires data distribution to be normal. The last two columns in Fig. 9 show the p-values computed by the t-test. When p-values are smaller than 0.05, we can reject the two-tailed null-hypothesis H_0 that there is no statistical significant difference in the mean value of the two samples, and we can formulate the alternative hypothesis H_0 that there is a statistical significant difference in the mean value of the two samples. So, we can claim that the mean class coverage and the mean method coverage of VLM-Fuzz are significantly higher than the corresponding means of both *DeepGUI* and *APE* , and the mean line coverage of VLM-Fuzz is significantly higher than mean line coverage of *APE*. Our sample size is 59, corresponding to the number of apps for which we computed class, method and line coverage values.

Our experiments show that VLM-Fuzz not only achieved a higher coverage on average but also reached to a high coverage point efficiently. As shown in Figure 10, which tracks the average coverage across all tested apps, VLM-Fuzz reached a higher (line, method, class) coverage point under 20 minutes than both *DeepGUI* and *APE*. Its coverage continues to improve over time and is yet to reach the peak at 60 minutes mark, whereas the coverage progress of *DeepGUI* and *APE* seems to have stalled by then.

Key takeaway

VLM- Fuzz achieved 46.5%, 53.2%, 68.5% line, method, and class coverages respectively on the benchmark apps, statistically outperforming the state-of-the-art approaches.

Table 3 Comparison based on #apps for which the approach achieves better coverage (RQ1)

Coverage	VLM-Fuzz vs *DeepGUI*	VLM-Fuzz vs *APE*
Class	53 vs 6	49 vs 10
Method	48 vs 11	40 vs 19
Line	43 vs 16	35 vs 23

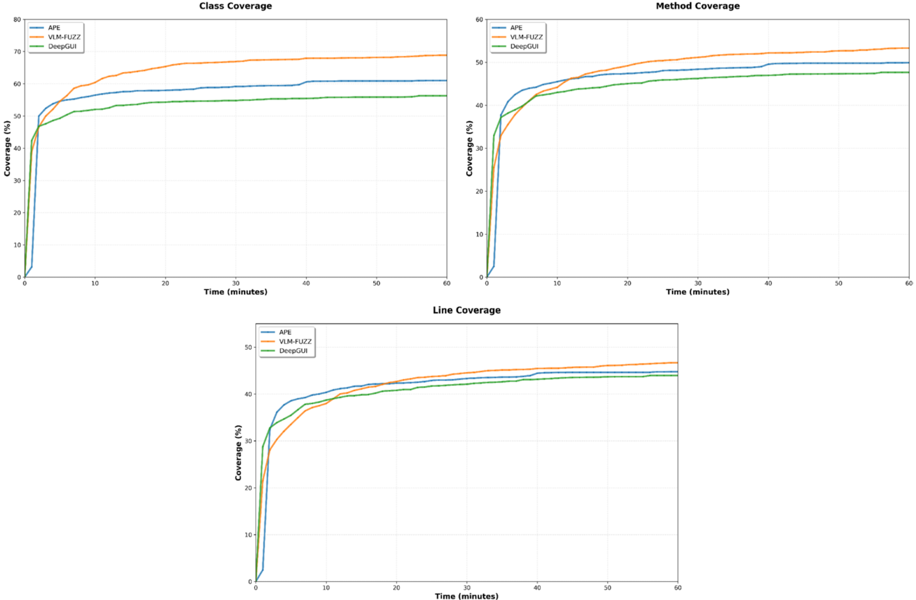


Fig. 10 Progressive average coverage metrics across all apps

4.5.2 RQ2: Ablation Study

The role of VLM Table 4 shows the number of apps that VLM-Fuzz did not utilize its VLM module. From the total dataset of 59 apps, VLM-Fuzz was able to test 13 apps without using its VLM capabilities. It also shows that, on 9 out of 13 apps, non-VLM-assisted tests achieved better line coverage compared to the baseline tool, *APE*. Note that in this ablation study, we compared with only *APE* and based on line coverage. This is because the number of apps for which VLM-Fuzz or *APE* performed better in terms of line coverage is closer (35 vs 23) than the numbers from any other comparisons, as shown in Table 3. For example, in comparison with *DeepGUI* , VLM-Fuzz performed better for 43 out of 59 apps in terms of line coverage.

To understand the impact of the VLM component, we randomly selected 5 apps for the ablation study from the remaining 46 apps that required VLM. Table 5 shows the per-

Table 4 #apps where VLM-Fuzz does not utilize VLM

No VLM	Won w/o VLM
13	9

Table 5 *APE* vs With VLM (full approach) vs Without VLM, based on average line coverage (%) (RQ2)

Selected App	*APE*	With VLM	Without VLM
fsck	3.2	21	7.2
yahtzee	10.6	53.8	45.8
weightchart	24.2	59.8	57.2
myLock	30.2	35.8	33.8
fercanet	30.6	35.2	35.2

formance comparison of the non-VLM-assisted variant of VLM-Fuzz against the full approach and the baseline, *APE*. The results on these 5 apps show that vision-based test generation plays an important role in reasoning with complex GUI components to generate valid inputs. VLM-Fuzz has a relative improvement of 5.28% line coverage compared to non-VLM-assisted variant of VLM-Fuzz .

On the other hand, we observed that the use of VLM does not always guarantee the improvement of code coverage. The code coverage for `fercanet` app is the same for our approach with and without VLM. Therefore, we conducted further (manual) analysis of the app to understand the reason. As shown in Figure 11, it is in fact a game app with a simple GUI which contains only buttons. But it requires understanding of musical notes to solve the challenges within a small time window (less than a minute in this example) and achieve better code coverage. This is an interesting case that highlights the limitation of fuzzing approaches in general. Without taking into account specific business logic, it may be hard to test the system adequately. We shall leave this challenge for future work.

Cost Efficiency While the integration of a powerful VLM such as GPT-4o is central to our approach's effectiveness, it also introduces a practical consideration: the monetary cost associated with API calls. The design of VLM-Fuzz directly addresses this through its on-demand invocation strategy. By invoking the VLM only when a GUI is considered complex and its semantic reasoning is essential (as explained in Section 3.4), our approach is engineered to be cost-efficient for typical testing scenarios, such as analyzing a single app. However, we recognize that this cost may increase significantly when scaling to large app sets or production environments. Moreover, depending on a proprietary model such as GPT-4o may affect reproducibility and long-term applicability because of LLM drift. A crucial

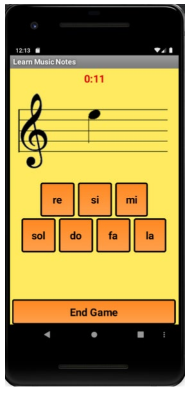


Fig. 11 A GUI of `fercanet` game app that requires understanding of musical notes to solve the given challenge within a given time

direction for future work is the exploration of open-source VLM alternatives and further cost-optimization techniques to address these scalability concerns while maintaining consistent performance.

The Role of Multi-Entry Testing Our algorithm is designed as a multi-entry testing approach. Here we aim to demonstrate that while multi-entry testing is a core feature of our approach, other algorithmic innovations such as hybrid, recursive search with per-Activity local exploration and budget-aware prioritization of complex components are also contribute to achieving a good code coverage. Therefore, we conduct another ablation study where we modify our tool to make it single-entry testing. We shall denote this modified version as VLM-Fuzz$_s$. We then ran VLM-Fuzz$_s$ against the same apps used in RQ1.

Table 6 shows the comparison between *APE* and VLM-Fuzz$_s$. Essentially, with single-entry-based testing, the class coverage of the tools becomes similar (both VLM-Fuzz$_s$ and *APE* have similar class coverage for 43 out of 59 apps). Even then, VLM-Fuzz$_s$ achieves better line coverage for 21 out of those 43 apps, with equal line coverage for 4 of them. This can be attributed to our recursive search with per-Activity local exploration approach. VLM-Fuzz$_s$ also manages to achieves better class and line coverage together for 7 apps. Overall, *APE* performs better than single-entry version of our approach only for 27 out of 59 apps. The main reason for these 27 cases, where our approach did not perform well, is due to the limited coverage of GUI events in our implementation of the tool. As acknowledged in Section 3.6, the current implementation only covers a of UI events — i.e., just 7 events, they are click, long-click, scroll, text-input, menu-tap, app switch and screen-rotate — in contrast, *APE* utilizes the full Android Monkey event suite, i.e., 33 events including GUI events (e.g., swipe/drag) and system events (e.g., lock/unlock device). Considering that different components are typically registered as system events handlers, when testing with the complete set of events APE as a matter of fact adopts a multi-entry strategy, while VLM-Fuzz$_s$ adopts a strictly single-entry because it always starts from the main launcher activity.

Due to these disadvantages, we expect that VLM-Fuzz$_s$ will be inferior to *APE* for certain apps that register different components as handlers to UI events that are not supported. For example, while *APE* can easily cover certain functionalities in `divideandconquer` app, we observed that the functionality can only be triggered by horizontal and vertical swiping events, which are not covered in our current implementation. We argue that this is the limitation of the implementation, not the approach.

Despite the inherent disadvantages of VLM-Fuzz$_s$ (strict single-entry and only 7 supported event types) compared to *APE* (partial multi-entry and 33 event types), VLM-Fuzz$_s$ remains highly competitive, outperforming or matching *APE* in 32 out of 59 apps (54%). Thus, the result demonstrates that our coverage gains are not just due to multi-entry launch-

Table 6 Comparison based on #apps for which VLM-Fuzz$_s$ (single-entry testing version) achieves better, equal, or worse coverage compared to *APE*. The table also shows the class coverage and the line coverage averaged across the 59 apps (RQ2)

Coverage	**VLM-Fuzz vs *APE***			
higher line coverage	28 vs 27			
equal line coverage	4			
	APE		VLM-Fuzz	
Coverage	Class	Line	Class	Line
(mean)	61.5	45.1	61.5	45.9

ing since the single-entry version of our approach achieves a better line coverage than *APE*. With more comprehensive coverage of UI events in our implementation (which is mainly engineering effort), it is expected that our approach will perform even better. This demonstrates that algorithmic innovations such as recursive search, per-Activity local exploration, and budget-aware prioritization of complex components along with strategic VLM invocation, which are unrelated to entry-point enumeration, are contributors to the tool's effectiveness.

Key takeaway

By selectively applying a vision language model to complex GUIs while using heuristics and hybrid search for simpler interfaces, VLM- FUZZ strikes a good balance between code coverage and cost, making it a cost-efficient solution for automated app testing. Our algorithm innovations, such as multi-entry testing, hybrid search, and budget-based test prioritization, contributes to achieving a good code coverage.

4.5.3 RQ3. Bug Detection Capability on Real-world Apps

To evaluate the effectiveness of VLM-FUZZ in detecting crash bugs, we conducted extensive testing with apps from two different sources — 80 real-world apps from Google Play Store and 55 apps from Themis benchmark (Su et al. 2021).

Evaluation with apps from Google Play Store We selected 80 official apps from Google Play Store that were updated in 2024. The selected apps represent a diverse user base, with download counts ranging from 500 to 1 billion (Table 7). Following the experimental setup detailed in Section 4.4, we executed VLM-FUZZ on this test suite. Our analysis focused specifically on fatal errors that caused app crashes, excluding errors that allowed processes to continue execution without crash. We prioritized fatal crashes because they strongly indicate the presence of software bugs and represent critical issues that directly impact user experience. Not all runtime errors indicate software bugs, particularly those triggered by security mechanisms or invalid component access. For instance, apps implementing tamper-proofing controls typically throw `App Sealing` errors when executed in an emulator environment - a legitimate security response rather than a bug. Through manual inspection, we filtered out such security-related errors from our bug analysis. We also observed that apps may throw runtime exceptions when a private activity component is invoked without the appropriate input data (such as private Intent parameters). While fuzzing tools can trigger these errors through privileged emulators and debugger mode, they would not occur during legitimate user interactions, and thus were not classified as software bugs. These considerations are different from existing approaches such as Su et al. (2017); Gu et al. (2019). Although these issues were not classified as bugs, we note that implementing proper exception handling remains a best practice regardless of how components are accessed, as it improves overall app robustness.

VLM-FUZZ induced crashes in 50 apps. Two of the authors manually investigated the crash logs (generated via ADB logcat) of these apps, together with the screenshots of the apps generated during fuzzing. The authors confirmed that crashes in 12 apps are caused

Table 7 Crashes detected in official apps (RQ3)

App	Version	#Download	#Crash by VLM-Fuzz	#Crash by *APE*	#Overlaps
`baqeyat`	3.3	100K	9	3	1
`eparking.pam`	2.0.5	1K	2	0	0
`parallel-space`	5.1.7	10M	5	0	0
`facebook.lite`	407.0.0.12.116	1B	1	0	0
`forza`	8	1K	10	4	2
`ggmgastro`	5.59.4	10K	1	0	0
`ktshow`	07.00.09	10M	6	1	1
`StudioScientifique`	1.1.3	500	1	0	0
`windeditlite`	3.0.0	1K	5	0	0
`lifebear`	4.6.7	1M	8	3	0
`obrien.dave`	4.9.1	50K	3	3	0
`passmobile`	3.7	5K	1	0	0
Total			52	14	4

by real bugs and that crashes in the other apps are due to the non-bug reasons stated above. As a comparison, we also ran *APE* on those 12 buggy apps to compare the bug detection capabilities. We did not run *DeepGUI* on those apps because *DeepGUI* only supports up to API Level 10 whereas those recent apps require API Level 28 and above.

Table 7 lists the name, version, and number of downloads of the buggy apps, the number of unique crashes by VLM-Fuzz and by *APE*, and the number of crashes overlapped between VLM-Fuzz and *APE*. VLM-Fuzz induced 52 unique crashes with 23 different types of exceptions that cause crashes. By contrast, *APE* induced 14 unique crashes with 9 different types of exceptions. `NullPointerException` is the most common type of the exceptions, which aligns to previous case studies (Gu et al. 2019; Su et al. 2017; Mao et al. 2016). Four out of the 14 crashes induced by *APE* overlap with those induced by VLM-Fuzz. We analyzed the details of these bugs. We observed that 9 of them involve multiple text inputs or compound operations to cause the crashes. Some crashes are only caused by particular sequences of operations. These findings indicate the ability of VLM-Fuzz to explore the state space effectively and the effectiveness of our approach in bug detection. For transparency, all the crash logs are provided in our repository.

We have sent the crash logs of those buggy apps to the respective developers. While direct feedback from developers can be slow, we have taken an additional step to validate

our findings by re-running the experiment 8 months later. We downloaded the latest available version of each of these apps from the Google Play Store (as of June 2025), except `baqeyat` app because we did not find any new version for this app. We then re-executed the specific test cases that had previously triggered crashes. Table 8 reports the results. Our re-validation confirmed that 10 of the 43 original crashes (excluding the 9 crashes from `baqeyat` app) are still reproducible. It also appears that the developers have fixed the bugs in 9 apps (new versions) as we were not able to reproduce 33 of the original 43 crashes. On the other hand, we observed 19 new crashes in 4 apps. The high persistence rate of bugs across subsequent app releases provides strong evidence of their validity and significance.

Industrial Relevance of Detected Bugs To address the critical question of whether the detected crashes represent practical, real-world issues versus unlikely edge cases, we performed a manual investigation of the unique crashes. For each crash log, we manually analyzed the crash stack traces and VLM-FUZZ's event logs leading up to the crash. We classify crashes caused by unnatural system-intent as "edge cases". The rest can potentially be triggered by normal user behaviour. Our findings indicate that 62/80 (77%) of the bugs could be triggered by normal user behavior. This is a direct benefit of our VLM-guided approach, which favors semantically plausible user interaction sequences over the unrealistic inputs often generated by traditional random fuzzers. We categorized the most critical bugs as follows across all tests:

- **Lifecycle and Initialization Crashes:** We found 9 fatal crashes related to NullPointerExceptions occurring within core Android lifecycle methods such as onResume(). These represent critical failures that could arise, for example, from restoring an app or rotating the screen.
- **Core Functionality Failures:** Many bugs were tied to essential app features. For instance, multiple apps crashed with NullPointerExceptions when attempting to get text from an EditText (27 instances), a clear failure in a common user task. Other examples

Table 8 Validation with new versions of the original buggy apps (RQ3)

App	New Version	#Crash total	#Crash reproduced	#Crash new
`baqeyat`	(no new ver.)	-	-	-
`eparking.pam`	2.1.4	10	0	10
`parallelspace`	5.5.6	0	0	0
`facebook.lite`	472.0.0.0.50	2	1	1
`forza`	8.1	10	8	2
`ggmgastro`	11.9.0	0	0	0
`ktshow`	08.00.07	0	0	0
`StudioScientifique`	1.1.8	0	0	0
`windeditlite`	3.1.0	1	1	0
`lifebear`	4.6.8	6	0	6
`obrien.dave`	4.9.7	0	0	0
`passmobile`	3.8	0	0	0
Total		29	10	19

include 20 instances of RuntimeExceptions/FileNotFoundExceptions caused by missing resources (e.g., icon image missing). The fuzzer also triggered 2 memory corruption within native code causing segmentation violation (SIGSEGV).
- **State Management Errors:** We discovered 6 instances of logical flaws in app navigation, such as a ClassCastException where an object was incorrectly cast to an Activity instance, indicating a fragile state management system that would fail under specific, but plausible, navigational paths.
- **Third-Party SDK Integration Bugs:** Our tool also triggered 6 instances of fatal exceptions within integrated third-party SDKs, such as advertising libraries during interactions with core Android APIs. These are also issues that can affect an app's stability and monetization.
- **Edge Cases:** Almost all the crashes can also be triggered by human interactions, provided the sequence or speed of operation matches the fuzzers (e.g., minimaize and restore app, rotate screen and restore quickly). However, some bugs were triggered because the fuzzer supports additional events that cannot be triggered by a human user (Section 3.6). For example, shortcut creation event and notification system events triggered 10 fatal crash.

The nature of these crashes strongly suggests they are not obscure edge cases but are genuine flaws in the application logic that could be encountered during normal use.

Evaluation with Apps from Themis Benchmark (Su et al. 2021). The Themis benchmark repository (Su et al. 2021) provides 79 Android apps (APKs), each with a documented bug, an expected crash signature, and a reproduction script. It also standardizes the testing process by managing app installation, execution time, and log collection. We integrated both VLM-Fuzz and our primary baseline, *APE* , into the Themis evaluation framework. We followed the same device and Android version configuration as the original work. We encountered a few issues when we attempt to evaluate our approach on the Themis benchmark. First, while we fuzzed the entire 79 apps, the provided crash verification script includes signatures for only 55 of the documented bugs. Therefore, the results in Table 9 are based on the 55 apps the script checks. In order to verify that the bugs are reproducible, we ran the the official bug reproduction scripts. However, the provided scripts were successful in triggering only 9 of the 79 documented bugs. Our manual review of several of the failing cases pointed to three main issues: (1) mismatches between the GUI element selectors in the scripts and the actual GUI hierarchies in the APKs, (2) unhandled permission related pop-ups interrupting execution, and (3) dependencies on backend services that are no longer active, leading to failures in key functions such as user login that make the documented bugs effectively unreproducible by any dynamic testing tool.

Despite these issues, our experiment provided insightful results. VLM-Fuzz successfully triggered 3 of the documented Themis bugs. Notably, one of these bugs was not triggered by its own reproduction script or by *APE*. For comparison, *APE* also triggered 3

Table 9 VLM-Fuzz vs *APE* against the Themis benchmark (RQ3)

	VLM-Fuzz	*APE*	Common
Documented crashes	3	3	2
Undocumented crashes	140	11	5

documented bugs, all of which were also triggered by the (working) reproduction scripts. On the other hand, our analysis uncovered a substantial number of previously undocumented crashes in the Themis benchmark. VLM-FUZZ induced 140 new, unique crashes across the benchmark's apps (crash exceptions available at VLM-Fuzz (2025)), whereas *APE* induced only 11. Of these, 5 crashes were induced by both tools. While we believe the algorithmic innovations also play an important role, in most of the cases, the multi-entry feature in VLM-FUZZ helped analyze public components while *APE* could not continue past the home page because of unavailable remote resource dependence. The results are summarized in Table 9.

Key takeaway

VLM- FUZZ uncovered several unique crashes corresponding to 23 different types of exceptions in 12 real-world apps that are released recently in 2024. On another benchmark containing 55 apps, VLM- FUZZ uncovered 143 bugs in which 140 were undocumented before. These results demonstrate the effectiveness and applicability of our approach over latest Android versions.

4.6 Case Studies

To analyze VLM-FUZZ's coverage patterns, we conducted detailed investigations of 6 representative apps. Our selection considers three scenarios: apps where VLM-FUZZ outperformed the best baseline, apps where it underperformed compared to the baseline, and apps where it operated without VLM assistance. This diverse sampling enables comprehensive understanding of the tool's effectiveness across different scenarios.

`bookcatalogue`: This is the running example app used in motivating and explaining our approach. For this app, VLM-FUZZ achieved 5% better line coverage than *APE*. This app required VLM since it has challenging GUIs that require reasoning about the semantic meanings of certain objects. When the app is first opened, the main activity component shows an empty bookshelf with no obvious interactive elements, except for a menu option at the bottom to add books. In another component called `BookEdit`, there is a group of input fields such as rating, notes, and dates. *APE* seems to have generated random values for those fields whereas VLM-FUZZ activated the VLM to generate inputs that have semantic meanings for each of the input fields.

`timer`: For this app, VLM-FUZZ achieved 8% better line coverage than *APE*. This app required VLM. The app has timer controls like setting minutes and pausing or canceling the timer. Similar to the above, VLM-FUZZ activated the VLM to generate inputs that have semantic meanings for those inputs.

`weightchart`: For this app, VLM-FUZZ achieved 25% better line coverage than *APE*. This app required VLM. VLM-FUZZ handled input fields, such as height and weight in `ChartActivity` and `EntryActivity` components, with more accuracy than *APE*. In addition, *APE* also missed exploring certain states in those activities whereas VLM-FUZZ managed to explore them efficiently, thereby, giving VLM-FUZZ an advantage in code coverage.

`blinkenlights`: For this app, VLM-Fuzz did not invoke Vision LM since its GUI does not have complex graphical elements. There is only one major activity component to test. Its GUI is not complex but has a setting feature that contains multiple interactive elements like checkboxes and theme selection. VLM-Fuzz achieved a significantly higher line coverage than the baseline (>25%), which shows the capability of our heuristic-based state space exploration algorithm to reach deeper codes within the app's component.

`divideandconquer`: For this app, *APE* achieved >6% line coverage than VLM-Fuzz. This was mainly because our tool does not support the "drag" user action required to test a component called `DivideAndConquerActivity`. The app requires the user to play a game by dragging the lines. *APE* managed to generate the drag action each time, thereby covering more of the app's functionality. The GUI actions currently supported in our implementation are stated in Section 3.6.

`multismssender`: For this app, *APE* achieved >15% line coverage than VLM-Fuzz. This was due to the similar reason mentioned above.

Overall, we observed that VLM-Fuzz performs better when the app requires complex interactions with the user, the system, and/or other apps and whether it requires reasoning about a group of GUI elements to determine the required inputs. We also observed that while VLMs excel at identifying visually apparent interface elements, they lack in generating non-visible interaction patterns, such as long-press gestures, scroll behaviors, and menu activations. Furthermore, the computational complexity and associated costs of iterative VLM queries following each interaction event, similar to the approach in Zhang et al. (2023); Liu et al. (2024), render this approach suboptimal for comprehensive test coverage. The experimental results suggest that an optimal testing methodology should integrate VLM capabilities with conventional heuristic-based testing algorithms, thereby establishing a more robust framework for GUI testing.

Key takeaway

While our approach builds on existing exploration strategies, the integration of VLMs introduces a novel way to enhance GUI reasoning and input generation. The improvements in code coverage are statistically significant and demonstrate the potential of VLM-assisted fuzzing. Furthermore, our method also demonstrates bug detection capabilities, effectively identifying different issues within real-world apps. Future optimizations aim to further enhance efficiency while balancing complexity and performance gains.

4.7 Threat to Validity

We adopted several strategies to mitigate the threats to the validity of our experimental results.

Internal validity threats concern external factors affecting the independent variable. To mitigate the risk of bias in selecting the case studies, we chose apps from an existing benchmark already used in previous studies. Additionally, the VLM used was proprietary, which might have evolved during the experiment (LLM drift), i.e., the model in the first part of the

experiment might differ from the model in the last part of the experiment. We mitigated this threat by running our experiment for a short period of time, thus the model evolution should have been limited or none.

Construct validity threats concern the relationship between theory and observation. We controlled these threats by using Emma tool (Rubtsov 2006) to measure code coverage, which is a standard coverage tool widely used in literature such as Choudhary et al. (2015); Su et al. (2017); Romdhana et al. (2022).

Conclusion validity threats concern the relationship between treatment and outcome. We used established statistical tests, Shapiro-Wilk test and paired t-test, to draw conclusions only when statistical significance was detected.

Eventually *external validity* concerns the generalization of the findings. Our results might not generalize to other case studies or mobile architecture (e.g., iOS). We controlled this threat by considering a large number of apps, but only further experiments with more apps and devices can confirm our results.

5 Related Work

One state-of-the-art approach that is most closely related to our approach is GPTDroid (Liu et al. 2024). Since GPTDroid tool is not available to us, in the following, we base our comparative discussion primarily on the methodological description provided in their paper, focusing on the conceptual differences between our approaches.

State tracking is managed by combining three key pieces of information: a summary of the last k=5 steps leading up to the current state, the number of times a widget is accessed and the name of the current component. Using this state information, the LLM determines what action to take next. However, this simplified approach to state tracking has limitations, particularly when handling complex GUI interactions. The problem becomes evident in scenarios where every action changes the main GUI. For instance, when the test starts on Activity A (State A1), opens a popup, returns to a modified version of Activity A (State A2), opens another popup, and then returns to yet another version of Activity A (State A3) that is equivalent to the first state (State A1), the system may struggle to differentiate between these states. Without more sophisticated state management, this can result in the approach getting trapped in an infinite loop as it fails to recognize the subtle differences between the various versions of the same GUI.

XML layout hierarchy attributes alone have significant limitations when interpreting user interface elements for input generation - which GPTDroid is based on. One major challenge is missing textual data – our analysis of 3,831 unique non-layout widget instances across the entire test apps revealed that 34.9% of the instances miss *text* attributes, 40.6% miss *resource-id*, and 24.5% are missing both making it challenging developing a heuristic based on these attributes. Table 10 summarizes the findings. Widget interactivity introduces additional complexity: elements that might be mistakenly considered non-interactive because *all* their boolean attributes (such as *"clickable"* or *"long-clickable"*) are set to `false` can still be interactive as they may inherit interactive properties from their parent containers. Additionally, the use of identical *resource-id* attributes across multiple widgets in the same GUI creates identification ambiguity. This limitation becomes particularly evident in sce-

Table 10 Widget attribute insight across the test apps

Attribute	%
text ✗, resource-id ✓	18.8%
text ✗, resource-id ✗	16.1%
text ✓, resource-id ✗	24.5%
text ✓, resource-id ✓	40.6%
text ✗	34.9%
resource-id ✗	40.6%

narios requiring visual reasoning, such as with password confirmation fields. In these cases, the approach needs to understand that it must enter identical text in both the original and confirmation fields - a relationship that may not be apparent from the XML layout attributes alone, especially if the widget attributes lack descriptive details. VLM-Fuzz , however, addresses these limitation by combining heuristic-based depth-first state exploration with VLM-assisted event generation.

Different from VLM-Fuzz , GPTDroid does not generate system Intents, hence, fails to simulate how an app might respond to system-wide broadcasts, potentially missing crucial functionality that relies on these events. Moreover, not considering intent-filters means it cannot adequately test various app launch scenarios or deep-linking capabilities. Finally, its limitation in simulating menu tap actions that could potentially trigger pop-up menus could result in incomplete GUI testing.

Trident (Liu et al. 2024) uses multimodal large language models to detect non-functional bugs by reasoning with the visual and functional logic of GUI pages. Different from ours, Trident purely relies on LLMs for both test generation and state space exploration. Our approach integrates static analysis, DFS, and a vision-language model for systematic state exploration. Since Trident has not been peer-reviewed (arXiv paper) yet, we did not compare with Trident in our experiments. But from our case studies (Section 4.6), we have observed that an optimal testing methodology should integrate VLM capabilities with conventional heuristic-based testing algorithms, for a more robust testing framework.

There are many other Android GUI fuzzing approaches that focus on improving code coverage. Choudhary et al. (Androtest) (Choudhary et al. 2015) conducted a comparison of *automated* test input generation tools for Android. The compared approaches include Monkey (Android 2024b), Dynodroid (Machiry et al. 2013), Droidfuzzer (Ye et al. 2013), Intent fuzzer (Sasnauskas and Regehr 2014), GUIRipper (Amalfitano et al. 2012), Orbit (Yang et al. 2013), SwiftHand (Choi et al. 2013), Puma (Hao et al. 2014), A3E-Targeted (Azim and Neamtiu 2013), EvoDroid (Mahmood et al. 2014), and ACTEve (Anand et al. 2012). In terms of the state space exploration strategy, these approaches employ either random exploration or model-based exploration. In model-based approaches, state space exploration is guided by a model of the app. For example, SwiftHand (Choi et al. 2013) uses finite state machine model of the app and A3E-Targeted (Azim and Neamtiu 2013) uses Static Activity Transition Graph of the app to guide the exploration of activity components and transitions. They may also employ a search strategy such as depth first search (Amalfitano et al. 2012; Yang et al. 2013), evolutionary search (Mahmood et al. 2014), or symbolic execution (Anand et al. 2012).

To further improve the scalability and test coverage, some approaches such as Sapienz (Mao et al. 2016) combines random fuzzing and search-based test generation. DroidBot (Li et al. 2017) proposes a lightweight GUI-guided test input generator that can generate GUI-guided test inputs based on a state transition model generated dynamically. Stoat (Su et al. 2017) uses dynamic analysis to construct a stochastic finite state machine based on only GUI events, for which a probability is assigned and updated based on their execution frequency. Next, it generates test sequences with GUI events and randomly injected system-level events.

To address the problem of incomplete model of the app when built statically, APE (Gu et al. 2019) applies both static and dynamic analyses. It first builds a static GUI model (finite state machine) and updates the model during testing. It generates test cases based on the model to explore various states/transitions/sequences. Its evaluation on the AndroTest benchmark demonstrated a significant improvement over both model-less approaches like Monkey and earlier model-based tools.

In general, random exploration strategy often fails to create reasonable testing paths tailored to the app's characteristics, leading to low test coverage. While model-based exploration is systematic and can enhance test coverage, its code coverage can be further improved by reasoning with the semantic information of the app's GUI and its widget items.

Since machine learning techniques are capable of learning from data, they have been incorporated into model-based exploration, to further improve the test coverage. For example, DeepGUI (YazdaniBanafsheDaragh and Malek 2021) complements fuzzing with a more intelligent form of GUI input generation and it was also validated on the AndroTest benchmark, showing its effectiveness compared to earlier approaches including Monkey. Given screenshots of apps, Deep GUI first employs deep learning to construct a model of valid GUI interactions. It then uses this model to generate effective inputs, given screenshots from the app under test.

However, the above-mentioned approaches may still struggle in dealing with complex and dynamic layout of activity screens, reasoning about the context of certain widget items, and getting stuck in the same activity screen or keep generating similar inputs that cause the app to crash or exit due to the lack of coordination between current input and previous inputs. These challenges were demonstrated in our experiments with existing approaches. In addition, these approaches do not consider the complexity of the components for allocating time budget for testing. In our experiments, we demonstrated that clever budgeting of testing time to complex components (those with several interactive GUI elements) result in better code coverage.

6 Conclusion

In this paper we present VLM-Fuzz , a novel approach to Android app UI fuzzing that uniquely combines Vision Language Models with traditional heuristic-based exploration strategies. Our work addresses critical limitations in existing Android testing approaches, particularly the challenge of achieving high code coverage when dealing with complex UI

interactions. By leveraging the visual reasoning capabilities of VLMs alongside a recursive depth-first search exploration strategy, VLM-FUZZ demonstrates significant improvements over current state-of-the-art solutions. The experimental results across the test apps show that VLM-FUZZ achieves better performance metrics, showing notable improvements over the best baseline approach. This performance enhancement validates our hypothesis that incorporating visual reasoning capabilities through VLMs can significantly improve the effectiveness of automated Android UI testing. Our ablation studies further confirm the value of integrating VLMs into the testing process, demonstrating that their ability to reason about complex GUI objects contributes significantly to improved test coverage. The results of VLM-FUZZ suggests that the combination of traditional search-based strategies with advanced AI models represents a promising direction for future research in automated software testing. While prior work has explored using LLMs for GUI testing, the core innovation of VLM-FUZZ lies in its strategic and cost-effective hybrid framework. Our approach is novel in its on-demand invocation of the VLM engine, reserving its powerful but computationally expensive reasoning capabilities for only the most complex UI screens where the traditional heuristics fail. By seamlessly integrating this with a fast, state-aware DFS for simpler interactions, VLM-FUZZ strikes a crucial balance between intelligent exploration and efficiency. This strategic combination allows our tool to achieve higher code coverage than both traditional fuzzers and expensive approaches that may rely more heavily on VLMs for all interactions, demonstrating a new and practical direction for UI testing. By making VLM-FUZZ publicly available, we provide a practical tool that can be immediately used by the software engineering community. Future work could explore extending this approach to other mobile platforms or web apps and pretraining/fine-tuning a specialized vision model, or investigating the integration of more advanced VLMs as they become available. The results of this study demonstrate that combining classical software testing methodologies with modern AI capabilities can lead to more effective and efficient testing solutions.

Appendix A UI interaction algorithms

A.1 performVisionActions()

Algorithm 3 outlines the `performVisionActions()` function. The function begins by capturing a screenshot of the current UI (line 2). It then labels the interactive widgets based on the dynamically retrieved UI hierarchy (line 3). Next, it queries the VLM to generate action sequences (line 4). The function then iterates through the identified actions, executing each one on its corresponding widget. For example, a text input is sent at line 17 to a widget that accepts text inputs. A tap action is sent at line 20 to a clickable widget. If an action triggers a UI change (the check at line 26), the current component is captured (line 27) and instantiates a new `UI_Analyzer()` (line 28) to process the components in the updated interface.

Algorithm 3 Perform Vision Generated Actions.

```
    Input: Inferred interactive widgets list (view_items_in_component)
 1: function PERFORMVISIONACTIONS(widgetsList)
 2:   screenshot ← takeScreenshot()
 3:   labelledScreenshot ← labelScreenshot(screenshot)
 4:   actionSequence, summary = getVisionActions(labelledScreenshot)
 5:   actionsList ← getActions(actionSequence)
 6:   for action ∈ actionsList do
 7:     if getCurrentComponent()! = currentVisibleUI() then
 8:       if not replay() then
 9:         // give up testing this component, we couldn't backtrack
10:         return
11:       end if
12:     end if
13:     label ← getLabel(action)
14:     widget ← getWidgetWithLabel(label)
15:     if "input" ∈ action then
16:       text ← getText(action)
17:       sendText(widget, text)
18:     end if
19:     if "tap" ∈ action then
20:       sendTapOrLongPress(widget)
21:     end if
22:     if "scroll" ∈ action then
23:       scrollDirection ← getDirection(action)
24:       scroll(scrollDirection)
25:     end if
26:     if actionCausedUiChange() then
27:       currentComponent ← getCurrrentComponent()
28:       UI_Analyzer(currentComponent)
29:     end if
30:   end for
31:   tap_menu()
32:   rotateAndRestoreScreen()
33:   if actionCausedUiChange() then
34:     currentComponent ← getCurrrentComponent()
35:     UI_Analyzer(currentComponent)
36:   end if
37:   return
38: end function
```

A.2 performNonVisionActions()

Algorithm 4 outlines the `performNonVisionActions()` function. The function traverses the list of widget-action map (line 2) in the current UI. For each identified widget, it executes its inferred associated action based on predefined mappings. For example, a text input is sent to widgets that expect text at line 11. While tap actions are sent to clickable widgets at line 14. If any action results in a UI state change (the check at line 19), it captures the new component (line 20) and initiates a new `UI_Analyzer()` (line 21) instance to evaluate the components within the modified interface.

Algorithm 4 Perform Non-Vision Generated Actions.

```
    Input: Inferred interactive widgets list (widgetsList)
 1: function PERFORMNONVISIONACTIONS(widgetsList)
 2:     for widget ∈ widgetsList do
 3:         if getCurrentComponent()! = currentVisibleUI() then
 4:             if not replay() then
 5:                 // give up testing this component, we couldn't backtrack
 6:                 return
 7:             end if
 8:         end if
 9:         if widget.action == ACTION_TEXT then
10:             text ← getRandomOrLLMText()
11:             sendText(widget, text)
12:         end if
13:         if widget.action == ACTION_TAP then
14:             sendTapOrLongPress(widget)
15:         end if
16:         if widget.action == ACTION_SCROLL then
17:             scrollDown(widget)
18:         end if
19:         if actionCausedUiChange() then
20:             currentComponent ← getCurrrentComponent()
21:             UI_Analyzer(currentComponent)
22:         end if
23:     end for
24:     tap_menu()
25:     rotateAndRestoreScreen()
26:     if actionCausedUiChange() then
27:         currentComponent ← getCurrrentComponent()
28:         UI_Analyzer(currentComponent)
29:     end if
30:     return
31: end function
```

Author Contributions – Biniam Fisseha Demissie: Writing - Review & Editing, Software, Investigation, Validation
– Yan Naing Tun: Writing - Review & Editing, Software, Investigation, Validation
– Lwin Khin Shar: Writing - Review & Editing, Supervision, Investigation, Validation
– Mariano Ceccato: Writing - Review & Editing, Supervision, Investigation, Validation

Funding Open access funding provided by Università degli Studi di Verona within the CRUI-CARE Agreement. The research leading to the results presented in this submission did not receive funding.

Data Availability The replication package containing VLM-Fuzz tool, its documentation, and detailed experiment results are available at VLM-Fuzz (2025).

Declarations

Ethical Approval The empirical evaluation presented in this submission is not subject to ethical approval in the organizations where the authors work.

Informed consent Not applicable.

Conflict of Interest The authors of this submission certify that they have NO affiliations with or involvement in any organization or entity with any financial interest (such as honoraria; educational grants; participation in speakers' bureaus; membership, employment, consultancies, stock ownership, or other equity interest; and expert testimony or patent-licensing arrangements), or non-financial interest (such as personal or professional relationships, affiliations, knowledge or beliefs) in the subject matter or materials discussed in this manuscript.

Clinical Trial Number Clinical trial number: not applicable.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM (2012) Using gui ripping for automated testing of android applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 258–261

Amalfitano D, Fasolino AR, Tramontana P, Ta BD, Memon AM (2014) Mobiguitar: Automated model-based testing of mobile apps. IEEE Softw 32(5):53–59

Anand S, Naik M, Harrold MJ, Yang H (2012) Automated concolic testing of smartphone apps. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, pages 1–11

Android (2024a) Android uiautomator. https://developer.android.com/training/testing/other-components/ui-automator##ui-automator

Android (2024b) Ui/application exerciser monkey. https://developer.android.com/studio/test/other-testing-tools/monkey

Azim T, Neamtiu I (2013) Targeted and depth-first exploration for systematic testing of android apps. In: Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pages 641–660

Choi K, Ko M, Chang B-M (2018) A practical intent fuzzing tool for robustness of inter-component communication in android apps. KSII Transactions on Internet & Information Systems 12(9):4248–4270

Choi W, Necula G, Sen K (2013) Guided gui testing of android apps with minimal restart and approximate learning. Acm Sigplan Notices 48(10):623–640

Choudhary SR, Gorla A, Orso A (2015) Automated test input generation for android: Are we there yet?(e). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 429–440. IEEE

Cruz L, Abreu R, Lo D (2019) To the attention of mobile software developers: guess what, test your app! Empir Softw Eng 24(4):2438–2468

Demissie BF, Ceccato M, Shar LK (2020) Security analysis of permission re-delegation vulnerabilities in android apps. Empir Softw Eng 25(6):5084–5136

Dong Z, Böhme M, Cojocaru L, Roychoudhury A (2020) Time-travel testing of android apps. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, pages 481–492

Du X, Liu M, Wang K, Wang H, Liu J, Chen Y, Feng J, Sha C, Peng X, Lou Y (2024) Evaluating large language models in class-level code generation. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13

Feng S, Chen C (2024) Prompting is all you need: Automated android bug replay with large language models. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24, New York, NY, USA. Association for Computing Machinery

Google (2025) Espresso. https://developer.android.com/training/testing/espresso

Gu T, Sun C, Ma X, Cao C, Xu C, Yao Y, Zhang Q, Lu J, Su Z (2019) Practical gui testing of android applications via model abstraction and refinement. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), pages 269–280. IEEE

Hao S, Liu B, Nath S, Halfond WGJ, Govindan R (2014) Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In: Proceedings of the 12th annual international conference on Mobile systems, applications, and services, pages 204–217

Hugging Face (2024) Wildvision arena : Benchmarking multimodal llms in the wild. https://huggingface.co/spaces/WildVision/vision-arena

Kochhar PS, Thung F, Nagappan N, Zimmermann T, Lo D (2015) Understanding the test automation culture of app developers. In 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), pages 1–10. IEEE

Kong P, Li L, Gao J, Liu K, Bissyandé TF, Klein J (2018) Automated testing of android apps: A systematic literature review. IEEE Trans Reliab 68(1):45–66

Lai D, Rubin J (2019) Goal-driven exploration for android applications. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 115–127. IEEE

Li Y, Yang Z, Guo Y, Chen X (2017) Droidbot: a lightweight ui-guided test input generator for android. In 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 23–26. IEEE

Li Y, Yang Z, Guo Y, Chen X (2019) Humanoid: A deep learning-based approach to automated black-box android app testing. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1070–1073. IEEE

Liu Z, Chen C, Wang J, Chen M, Wu B, Che X, Wang D, Wang Q (2024) Make llm a testing expert: Bringing human-like interaction to mobile gui testing via functionality-aware decisions. In: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, pages 1–13

Liu Z, Li C, Chen C, Wang J, Wu B, Wang Y, Hu J, Wang Q (2024) Vision-driven automated mobile gui testing via multimodal large language model. arXiv:2407.03037

Machiry A, Tahiliani R, Naik M (2013) Dynodroid: An input generation system for android apps. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pages 224–234

Mahmood R, Mirzaei N, Malek S (2014) Evodroid: Segmented evolutionary testing of android apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 599–609

Mao K, Harman M, Jia Y (2016) Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th international symposium on software testing and analysis, pages 94–105

Moran K, Linares-Vásquez M, Bernal-Cárdenas C, Vendome C, Poshyvanyk D (2017) Crashscope: A practical tool for automated testing of android applications. In: 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), pages 15–18. IEEE

OpenAI (2024) Gpt-4-vision. https://help.openai.com/en/articles/8555496-gpt-4-vision-api

Pan M, Huang A, Wang G, Zhang T, Li X (2020) Reinforcement learning based curiosity-driven testing of android applications. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 153–164

Peng C, Zhang Z, Lv Z, Yang P (2022) Mubot: Learning to test large-scale commercial android apps like a human. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 543–552. IEEE

Romdhana A, Merlo A, Ceccato M, Tonella P (2022) Deep reinforcement learning for black-box testing of android apps. ACM Transactions on Software Engineering and Methodology (TOSEM) 31(4):1–29

Rubtsov V (2006) Emma: Java code coverage tool

Ryan G, Jain S, Shang M, Wang S, Ma X, Ramanathan MK, Ray B (2024) Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. Proceedings of the ACM on Software Engineering, 1(FSE):951–971

Sasnauskas R, Regehr J (2014) Intent fuzzer: crafting intents of death. In Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), pages 1–5

Su T, Meng G, Chen Y, Wu K, Yang W, Yao Y, Pu G, Liu Y, Su Z (2017) Guided, stochastic model-based gui testing of android apps. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 245–256

Su T, Wang J, Su Z (2021) Benchmarking automated gui testing for android against real-world bugs. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, page 119–130, New York, NY, USA. Association for Computing Machinery

VLM-Fuzz (2025) Replication package of vlm-fuzz. https://bit.ly/VLM-Fuzz

Yang W, Prasad MR, Xie T (2013) A grey-box approach for automated gui-model generation of mobile applications. In: International Conference on Fundamental Approaches to Software Engineering, pages 250–265. Springer

YazdaniBanafsheDaragh F, Malek S (2021) Deep gui: Black-box gui input generation with deep learning. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 905–916. IEEE

Ye H, Cheng S, Zhang L, Jiang F (2013) Droidfuzzer: Fuzzing the android apps with intent-filter tag. In: Proceedings of International Conference on Advances in Mobile Computing & Multimedia, pages 68–74

Zhang C, Yang Z, Liu J, Han Y, Chen X, Huang Z, Fu B, Yu G (2023) Appagent: Multimodal agents as smartphone users. arXiv:2312.13771

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Biniam Fisseha Demissie[1] · Yan Naing Tun[2] · Lwin Khin Shar[2] · Mariano Ceccato[3]

✉ Mariano Ceccato
mariano.ceccato@univr.it

Biniam Fisseha Demissie
biniam.demissie@tii.ae

Yan Naing Tun
yannaingtun@smu.edu.sg

Lwin Khin Shar
lkshar@smu.edu.sg

[1] Technology Innovation Institute, 9639 Masdar City, Abu Dhabi, UAE

[2] Singapore Management University, Singapore, Singapore

[3] University of Verona, Verona, Italy