

FuncDroid: Towards Inter-functional Flows for Comprehensive Mobile App GUI Testing

JINLONG HE, Institute of Software, Chinese Academy of Sciences, China
CHANGWEI XIA, University of Chinese Academy of Sciences, China
BINRU HUANG, University of Chinese Academy of Sciences, China
JIWEI YAN*†, Institute of Software, Chinese Academy of Sciences, China
JUN YAN*‡†, Institute of Software, Chinese Academy of Sciences, China
JIAN ZHANG‡†, Institute of Software, Chinese Academy of Sciences, China

As mobile application (app) functionalities grow increasingly complex and their iterations accelerate, ensuring high reliability presents significant challenges. While functionality-oriented GUI testing has attracted growing research attention, existing approaches largely overlook interactions across functionalities, making them ineffective at uncovering deep bugs hidden in inter-functional behaviors. To fill this gap, we first design a **F**unctional **F**low **G**raph (**FFG**), a behavioral model that explicitly captures an app's functional units and their inter-functional interactions. Based on the FFG, we further introduce an *inter-functional-flow-oriented GUI testing* approach with the dual goals of precise model construction and deep bug detection. This approach is realized through a *long–short-term-view-guided testing* process. By combining two complementary test-generation views, it can adaptively refine functional boundaries and systematically explore inter-functional flows under diverse triggering conditions. We implement our approach in a tool called **FuncDroid**, and evaluate it on two benchmarks: (1) a widely-used open-source benchmark with 50 reproducible crash bugs and (2) a diverse set of 52 popular commercial apps. Experimental results demonstrate that FuncDroid significantly outperforms state-of-the-art baselines in both coverage (+28%) and bug detection number (+107%). Moreover, FuncDroid successfully uncovers 18 previously unknown non-crash functional bugs in commercial apps, confirming its practical effectiveness.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Mobile App, GUI Testing, Functional Bug, Large Language Model

ACM Reference Format:
Jinlong He, Changwei Xia, Binru Huang, Jiwei Yan, Jun Yan, and Jian Zhang. 2026. FuncDroid: Towards Inter-functional Flows for Comprehensive Mobile App GUI Testing. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA112 (October 2026), 23 pages. https://doi.org/10.1145/3832203

*Corresponding authors.
†Authors are also affiliated with Beijing Key Laboratory of Intelligent Software Engineering.
‡Authors are also affiliated with Key Laboratory of System Software (Chinese Academy of Sciences) and Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences (UCAS).

Authors' Contact Information: Jinlong He, Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China, hejinlong@otcaix.iscas.ac.cn; Changwei Xia, Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China, xiacw@ios.ac.cn; Binru Huang, Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China, huangbr@ios.ac.cn; Jiwei Yan, Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China, yanjiwei@otcaix.iscas.ac.cn; Jun Yan, Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing, China, yanjun@ios.ac.cn; Jian Zhang, Institute of Software, Chinese Academy of Sciences, Beijing, China, zj@ios.ac.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA112
https://doi.org/10.1145/3832203

1 Introduction

Mobile applications (apps) are now deeply embedded in modern society, permeating nearly every aspect of daily life. These apps can be viewed as collections of functionalities, with user interaction primarily consisting of operating these functional components. As app functionality design grows increasingly complex and functional iterations accelerate, ensuring high reliability poses significant challenges. While a wide range of *coverage-oriented* automated GUI testing techniques have been proposed [9, 10, 14, 20, 26, 30, 36, 40], mobile app testing in practice still relies heavily on manual effort. Testers frequently design functionality-specific test cases to detect defects, highlighting the continued effectiveness of *functionality-oriented testing* in bug discovery. However, manually designing such functional test cases is time-consuming, labor-intensive, and difficult to scale. Consequently, a key research challenge in recent years has been to enhance automated GUI testing so that its bug-detection effectiveness can approach that of manual, functionality-oriented testing, while retaining the scalability benefits of automation.

To gain a deeper understanding of mobile apps and enhance traditional GUI testing methods, a growing body of recent research [22, 23, 29, 38, 39, 46] has increasingly incorporated large language models (LLMs) [1, 2, 8, 25, 27]. These studies leverage LLMs' strengths in semantic comprehension and contextual reasoning to enable a degree of *functionality-oriented* test generation, which can be broadly grouped into two categories. The first category identifies specific functional patterns. Approaches like GPTDroid [22] and VisionDroid [23] fall into this category. They employ LLMs to directly interpret GUI pages, summarize page sequences into reusable functional patterns, and subsequently apply these patterns to guide further exploration. The second category focuses on augmenting general function-related app testing knowledge. It comprises frameworks such as LLMDroid [34] and MemoDroid [7], which enhance existing testing tools (either traditional or LLM-based) by constructing knowledge bases that capture app functionality characteristics. These frameworks distill common interaction patterns and effective exploration strategies from historical testing sessions to guide more efficient exploration. While these approaches mark a significant shift from *coverage-oriented* to *functionality-oriented* testing, their modeling of functionality is often incomplete and oversimplified, typically represented as a mere linear sequence of events. More critically, existing approaches primarily operate by testing individual functionalities in isolation, lacking attention to interactions among functionalities. Therefore, such approaches often fall short in uncovering deeper bugs that are hidden in the interactions between different functionalities.

To achieve genuinely effective functionality-oriented GUI testing for deep bug detection, the following challenges may be faced. **First, how can we build an accurate behavioral model that precisely delineates functional boundaries and captures their interactions?** In practice, functional boundaries are often ambiguous, as functionality can be defined at multiple levels of granularity, ranging from the entire application to individual widget operations. This ambiguity hinders the automatic segmentation of the GUI into *meaningful functionalities*, since their boundaries may evolve as more knowledge about the app is acquired. Moreover, as functional boundaries shift, accurately identifying and modeling the flows between functionalities is also challenging. **Second, based on such a behavioral model, how can we achieve effective testing that systematically exercises complex interaction paths and increases the likelihood of uncovering deep bugs?** In real-world apps, the number of meaningful functionalities is often large, and the combinations of functional flows among them lead to an exponential growth of the test space. Without proper test guidance, testing efforts tend to degenerate into redundant and shallow exploration, becoming a blind search that is prone to missing subtle defects. **Last but not least**, given that a complete inter-functional model emerges only through iterative exploration, separating model construction from testing is neither practical nor cost-effective. Thus, it is essential to address **how to guide**

the exploration process such that model refinement and testing proceed in a mutually reinforcing manner, progressively improving the identification of functional boundaries and flows. This issue is a holistic challenge that is tightly intertwined with the two challenges above.

To address the overall challenges, we first design a behavioral model called **Functional Flow Graph (FFG)**, whose node represents a *meaningful functionality*, and edge represents an *inter-functional flow*. Upon the FFG, we further propose an **inter-functional-flow-oriented GUI testing approach**, which employs an iterative process that systematically constructs and exploits the FFG to detect deep interaction bugs. This approach is mainly realized through a *long–short-term-view-guided testing* process. By combining two complementary test-generation views, it can adaptively refine functional boundaries and systematically explore inter-functional flows under diverse triggering conditions.

Specifically, we perform two synergistic steps in each iteration: the *long-short-term view exploration* and the *functional flow graph updating*. **(1) Long-Short-Term View Exploration.** This method employs two complementary perspectives—long-term and short-term—for test scenario generation, each targeting one of the core challenges identified earlier. To address the first challenge, the *Long-Term View* generates test scenarios aimed at refining the FFG. These scenarios challenge the completeness and independence of hypothesized meaningful functionalities and examine the precision of flow conditions, thereby producing evidence to enhance the FFG in the next step. To address the second challenge, the *Short-Term View* generates test scenarios for in-depth exploration based on the current FFG. By applying both *single-flow* and *cross-flow* metamorphic transformations to the state conditions of existing flows, it systematically explores the relevant state subspaces to uncover bugs from complex functional interactions. **(2) Functional Flow Graph Updating.** Based on test scenarios generated from both views, corresponding GUI traces are constructed and executed as test cases. The execution results, in turn, support the updating of the functional flow graph. By analyzing observed behaviors and flow conditions, we define three types of *node-updating* operations and four types of *flow-updating* operations. These updates work in concert with the long–short-term view testing to continuously refine the model and enable in-depth GUI exploration.

To evaluate the effectiveness of our approach, we implement a tool named **FuncDroid**. We first conduct extensive experiments on the widely used open-source benchmark *Themis* [31], which comprises **50** usable crash bugs across 17 apps. On this benchmark, FuncDroid significantly outperforms five state-of-the-art baselines in both coverage and bug detection capability. Specifically, FuncDroid achieves a **28% improvement in activity coverage** and detects a total of **62 bugs**, including 49 crash bugs and 13 non-crash functional bugs. In comparison, the best-performing baseline detects 30 bugs, indicating that our approach achieves at least **107% increase in bug detection effectiveness**. To further assess its practicality on real-world apps, we evaluate FuncDroid on **52** popular, high-download commercial apps collected from Google Play and HUAWEI AppGallery. On their latest versions, FuncDroid discovers **18 previously unknown non-crash functional bugs**, showing its effectiveness in complex, real-world scenarios.

In conclusion, the major contributions of this paper are as follows:

- We present the first functionality-oriented model for mobile apps, called the **Functional Flow Graph (FFG)**, which captures both functionalities and their inter-functional flows.
- We propose an **inter-functional-flow-oriented GUI testing** approach that iteratively constructs and exploits the FFG to systematically explore state subspaces and effectively detect deep bugs arising from complex functional interactions.
- We implement our approach in a tool FuncDroid and evaluate it on both open-source and commercial apps, uncovering **62 existing bugs** and **18 previously unknown functional bugs**. The results demonstrate that a precise FFG substantially improves deep bug detection.

2 Motivating Example

Before presenting our approach, we introduce a motivating example that underscores the need for considering inter-functional flows and exposes the limitations of existing techniques for testing complex cross-functional interactions. Figure 1 shows two bugs in the *Blood Pressure* app on Google Play, which has been downloaded over 10 million times. We give the details about the two bugs in the following. Both bugs are triggered only by a specific sequence of actions spanning several distinct functionalities. Testing a single functionality in isolation may therefore miss these bugs. The second bug is particularly subtle, as it requires a precise condition: the same alarm configuration must be deleted in one functional context and then re-added in another, revealing an inconsistency in how the app manages alarm identities across modules. Notably, neither of these two bugs was detected by any of the baseline testing tools.

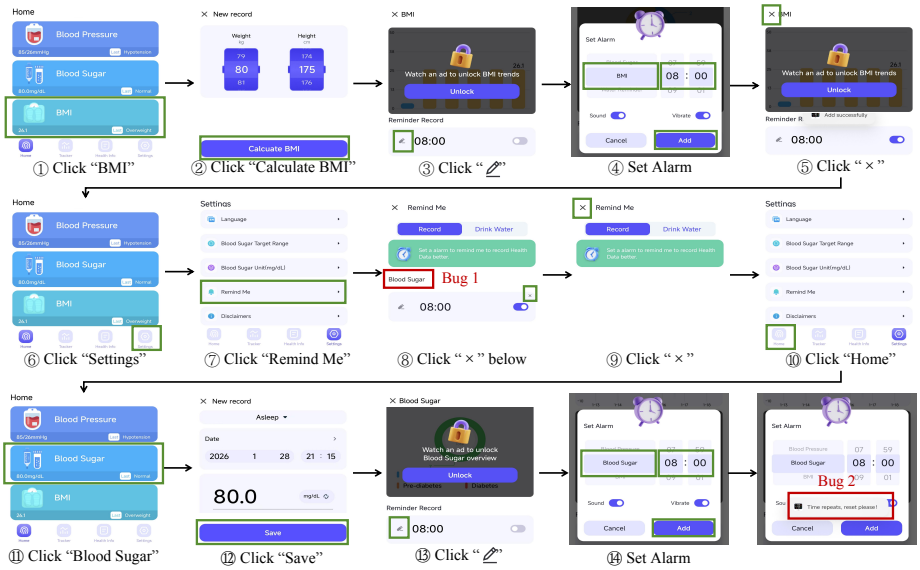


Fig. 1. Two Functional Bugs in the Motivating App *Blood Pressure*.

Functional Bug 1: BMI Alarm is Misclassified

Trigger Scenario. Sets a new BMI alarm in the app.
GUI Trace (Steps).

(1) On the "Home" page, click "BMI" (①), click "Calculate BMI" (②), then click "Edit" to set an alarm (③).
(2) Choose "BMI" and "08:00", then click "Add" (④).
(3) Click the page close icon "×" to go back to the "Home" page (⑤), tap "Settings" (⑥), and click "Remind Me".

Observed Bug. The "Remind Me" page shows an alarm for "blood sugar" instead of "BMI".

Functional Bug 2: Re-adding a Deleted Alarm Fails

Trigger Scenario. Re-adds a Blood Sugar alarm at the same time as a previously deleted one.
GUI Trace (Steps).

(1) On the "Remind Me" page, click the alarm delete button"×" (⑧), then click the page close icon "×" (⑨) to return to the "Settings" page, and tap "Home" (⑩) to go back to the "Home" page.
(2) Click "Blood Sugar" on the "Home" page (⑪), click "Save" (⑫), then click "Edit" (⑬) to set an alarm.
(3) Choose "Blood Sugar" and "08:00", then click "Add" (⑭).

Observed Bug. The app shows a "Time repeat" warning message, and the alarm is not added.

3 Approach of FuncDroid

To uncover complex bugs such as those illustrated in Figure 1, we design the **Functional Flow Graph (FFG)** to model functional interactions in terms of the state conditions under which they occur, thereby enabling in-depth cross-functionality testing. In the FFG, each node represents a meaningful functionality, and each edge, denoted as $n \xrightarrow{(\pi,\phi,\pi')} n'$, represents a flow: it specifies that after executing the trace π of functionality n, if the app state satisfies the condition ϕ, then the trace π' of functionality n' becomes executable. Crucially, each such flow, through its condition ϕ, defines a meaningful subset of the overall app state space. By constructing a graph of these flows, the FFG thus provides a structured partition of the state space into coherent, functionally relevant regions. FuncDroid's key idea is that, *guided by this partition, testing can then focus on thorough exploration within each state subset, rather than attempting a blind, exhaustive search of the entire space.* Based on this idea, we propose an **Inter-Functional-Flow-oriented GUI testing approach**, which iteratively builds and exploits the FFG to cover state subspaces systematically and detect deep interaction bugs effectively.

3.1 Definition of Functional Flow Graph

While the concept of functionality is frequently mentioned in GUI testing research, its practical definition remains ambiguous. This ambiguity arises because functionality can be conceptualized at multiple granularity levels, spanning from fine-grained widget-level to coarse-grained app-level, with no universally accepted standard. In this work, we argue that a *meaningful functionality* is the most pertinent level of granularity for testing. A meaningful functionality, as defined here, must satisfy two key criteria: (1) it is **complete**, i.e., it accomplishes a well-defined, user-centric task; and (2) it is **independent**, i.e., it is logically indivisible and minimal, meaning it does not itself contain other meaningful sub-functionalities.

Based on the concept of "meaningful functionality", we accordingly define the **functional flow graph (FFG)** as a directed graph $G = (N, E)$, where N denotes the set of meaningful functionalities and E represents the inter-functional flows (*flow* for short) between them. Specifically:

- A *functionality* $n \in N$ is defined as a triple (*goal*, *vars*, *traces*), where:
  - *goal* is the semantic intent of the functionality (e.g., "Alarm Management");
  - *vars* is the set of state variables that are accessed or modified during the execution of the current functionality, spanning GUI property, data entity, and system setting dimensions;
  - *traces* is a set of execution traces, each consisting of a sequence of widget actions involved in the functionality.
- A *flow* $(n, n') \in E$ is annotated with a triple (π, ϕ, π'). This denotes that executing the trace π of functionality n leads the app state to satisfy the condition ϕ, which enables the activation of the trace π' in functionality n'. Formally, we write it as $n \xrightarrow{(\pi,\phi,\pi')} n'$.

3.2 Inter-Functional-Flow-oriented GUI Testing

Based on that model design, we then propose an inter-functional-flow-oriented GUI testing approach, which targets covering more flows to detect deep bugs. As shown in Figure 2, it first constructs the initial functional flow graph (FFG) from the execution trace collected by the lightweight automated GUI exploration (Section 3.2.1). Then it employs an iterative process to systematically uncover deep interaction bugs by exploring the state subspaces defined by the FFG. Each iteration performs two synergistic steps:

(1) *Long-Short-Term View Exploration (Section 3.2.2).* It is composed of three coordinated phases:
   (a) *Long-Term View (LTV) test Scenario Generation* generates tests that target refining the FFG

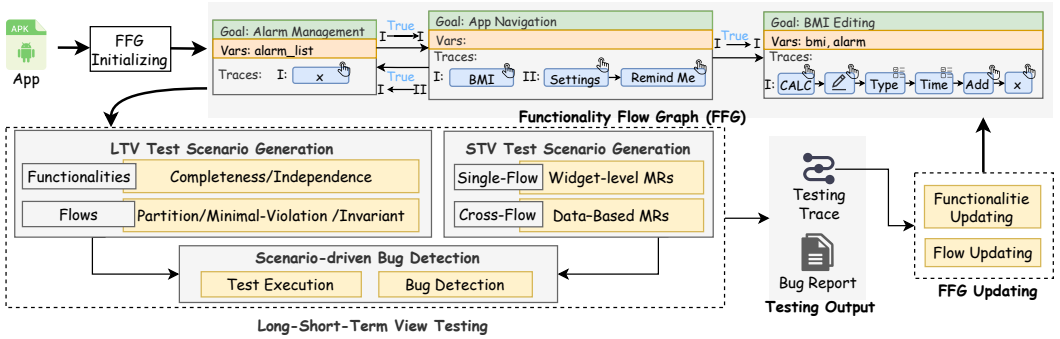


Fig. 2. Overview of FuncDroid

model; (b) *Short-Term View (STV) Test Scenario Generation* generates tests aimed at intensive exploration of current flow hypotheses; and (c) *Scenario-driven Bug Detection* performs both test execution and bug detection;

(2) *Functional Flow Graph Updating (Section 3.2.3).* It continuously carries out the actual functionality and flow updates on the FFG based on the collected execution information.

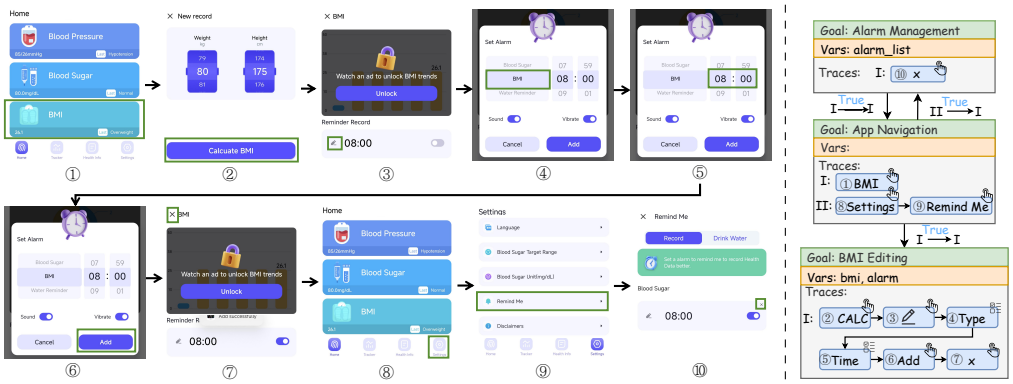
Through the iterative cycle with both steps, FuncDroid progressively deepens its understanding of the app while efficiently covering diverse and effective flows.

3.2.1 Functional Flow Graph Initializing. This step aims to initialize the FFG. It begins with a lightweight, automated GUI exploration (e.g., using random or systematic exploration strategies) to collect an initial execution trace, which serves as the basis for hypothesizing an initial set of functionalities and flows among them.

Functionality Initialization. Given the execution trace, we partition it into a set of candidate meaningful functionalities based on semantic coherence. Initially, for the first page in the trace, we extract its screenshot, widget hierarchy tree, and textual content; subsequently, we employ an MLLM to infer its goal. Thus, we can initialize a candidate functionality $n = (goal, vars, traces)$, where *traces* initially contains only the first page, and *vars* is empty. Then, we analyze each subsequent page in order and infer its goal similarly. To determine whether the functionality of the current page belongs to the previously identified functionality or represents a new one, we perform a two-step comparison via the MLLM: (1) If the page's goal is a continuation of its predecessor's goal, we consider it part of the predecessor's functionality and append it to the corresponding trace. (2) Otherwise, we compare it against the goals of other existing functionalities; if a match is found, we record a new trace in that functionality; otherwise, we create a new functionality with a new trace. Finally, we extract the state variables *vars* for each functionality as follows. Note that these variables are abstractions (e.g., data entities, widget properties, system settings), not real program variables. For each page in the *traces*, we analyze the interactive widgets, displayed data, and runtime logs to infer which variables are accessed or modified on that page, and then merge them into the *vars* of the functionality.

Functional Flow Initialization. Then, after the functionalities are partitioned, we extract inter-functional flows at functionality switching points and add them to the initial FFG. Specifically, given the initial execution trace, if a trace π belonging to a functionality n is immediately followed by a trace π' belonging to another functionality n', we create a candidate edge $n \xrightarrow{(\pi,\phi,\pi')} n'$. At this stage, the state condition ϕ is conservatively set to *True*, indicating that the transition is

inferred solely from the observed temporal succession, without yet reasoning about the precise state conditions that enable it.



(a) The Initial Execution Trace. (b) The Initial FFG.

Fig. 3. FFG Initialization of the *Blood Pressure* App.

For example, Figure 3 illustrates the initialization process on the *Blood Pressure* app. Its initial execution trace (Figure 3(a)) consists of ten actions ①-⑩.

The Initial Execution Trace with 10 Actions

① clicking the "BMI" button on the "Home" page; ② clicking the "Calculate BMI" button on the reached "BMI" page; ③ clicking the "Edit" button; ④ configuring alarm type on the reached "Alarm Editing" page; ⑤ configuring alarm time; ⑥ clicking the "Add" button; ⑦ clicking the "×" button on the reached "BMI" page; ⑧ clicking the "Settings" button on the reached "Home" page; ⑨ clicking the "Remind me" button on the reached "Settings" page; ⑩ clicking the "×" button on the reached "Remind me" page.

By analyzing the *goal* of each page, the initial FFG is constructed as shown in Figure 3(b). The trace is partitioned into three functionalities. Specifically, the functionality *App Navigation* encompasses two separate traces: trace I containing action ①, and trace II containing actions ⑧ and ⑨; the functionality *BMI Editing* encompasses one trace I containing actions ②–⑦; the functionality *Alarm Management* encompasses one trace I containing action ⑩. The conditions of the flows between these three functionalities are initially set to be *True*. Moreover, the functionality *BMI Editing* includes the variable *alarm* because its trace I contains pages that configure alarm type (④), configure alarm time (⑤), and add an alarm (⑥), i.e., the pages accesses and modifies the *alarm*-related data entity.

3.2.2 Long-Short-Term View (LSTV) Exploration. This step implements our core testing strategy by systematically generating inter-functional test scenarios from two complementary perspectives: the *Long-Term View* (LTV) and the *Short-Term View* (STV). Although these two views differ in design intent, the scenarios they produce are uniformly consumed by a common *Scenario-driven Bug Detection* phase. To support this unified processing, we introduce a formal representation for scenarios generated from both views, referred to as *LSTV test scenarios*. An *LSTV test scenario* is defined as a 4-tuple (*type*, *strategy*, *object*, *guidance*), where *type* ∈ {LTV, STV} denotes the perspective of the scenario generation; *strategy* indicates the specific generation strategy or metamorphic

relation applied; *object* $\in N \cup E$ specifies the targeted functionality or functional flow to which the strategy is applied; and *guidance* is a natural-language description that explains how the *strategy* is instantiated and applied to the *object* in order to construct the scenario.

(1) Long-Term View Test Scenario Generation. Since the effectiveness of testing depends on the quality of the underlying model, i.e., the FFG, the long-term view focuses on generating test scenarios that iteratively refine the FFG for subsequent testing. This phase generates test scenarios that can refine the two core aspects of the FFG: its *functionality definitions* and *flow conditions*.

On one hand, under partial exploration, the initial identification of functionalities may be biased. To **refine functionality definitions**, we design test scenarios that can validate whether each recorded functionality $n \in N$ satisfies the criteria of a *meaningful functionality*, i.e., it is both complete and independent, with the following two strategies. Specifically, when multiple functionalities are frequently exercised together, we evaluate whether they collectively constitute an indivisible user task. If they do, we merge them into a single functionality; otherwise, they remain separate. Conversely, if a single functionality's traces can be clustered into multiple semantically coherent subgroups, we consider splitting it into smaller independent sub-functionalities to preserve independence.

- *Completeness Validation Strategy*. To challenge the *completeness* of a functionality n, we analyze its high-level semantic *goal* together with the GUI pages in its *traces* to identify potential gaps between the intended functionality and the recorded execution. Based on this analysis, we infer a set of potential widget actions that are semantically essential for achieving the *goal* but are absent from the recorded *traces*. We then generate a new test scenario that executes the existing trace of n and subsequently attempts to execute the missing actions.
- *Independence Validation Strategy*. To challenge the *independence* of a functionality n, we compute its *trace-level goals* by generating a semantic goal for each trace in the set *traces*. These trace-level goals are then clustered to identify potential subgroups. If the clustering reveals a clear separation of traces into groups with low inter-cluster similarity, it indicates that n may encompass multiple semantically coherent sub-tasks. For each trace cluster, we generate a new test scenario that executes only the traces within the cluster, in order to validate whether the corresponding "sub-task" can be executed as an independent functionality.

For example, Figure 4 illustrates how a test scenario is generated and used to refine functionality definitions through the completeness validation strategy. We take the FFG shown in Figure 3(b) as the current FFG in this example. We first observe that the action of clicking the "Blood Sugar" button (same for "Blood Pressure" brevity) is not present in the traces of the functionality *App Navigation*. Consequently, we generate a scenario that attempts to interact with the "Blood Sugar" button in order to assess whether the functionality *App Navigation* is complete. Using the technique described in the *Scenario-driven Bug Detection phase*, we can finally get an execution trace that includes the missing actions, which successfully explores the path after clicking that "Blood Sugar" button. By goal comparison, this exploration didn't update the existing functionality. Instead, it leads to the discovery of a new, independent Blood Sugar Editing functionality.

On the other hand, to **refine flow conditions**, we design a set of test scenarios that aim to sharpen the flow condition ϕ of a flow $e = n \xrightarrow{(\pi,\phi,\pi')} n'$. The core intuition is to determine whether the condition ϕ can be soundly weakened or strengthened while preserving the transition to n'. Moreover, each condition ϕ is expressed in disjunctive normal form (DNF), i.e., a disjunction of conjunctive clauses (e.g., $\phi = \phi_1 \vee \cdots \vee \phi_n$ where $\phi_i = \psi_1 \wedge \cdots \wedge \psi_{k_i}$ for each i). To handle the *True*

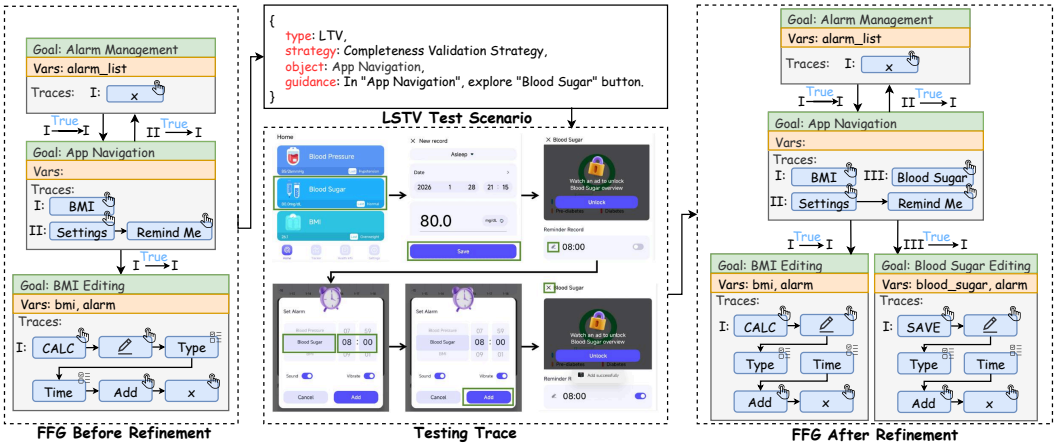


Fig. 4. Example of LTV Test Scenario for Refining Functionality Definitions.

condition, we construct a DNF by analyzing the state variables *vars* of the target functionality n'. To this end, we construct environment configurations that are highly correlated with π' through a series of targeted operations. Executing π' under these constructed environments yields a set of new test scenarios. These strategies are described in detail below.

- *Condition Partition Strategy*. To assess whether a flow condition ϕ should be split into multiple distinct conditions, we generate test scenarios to examine its internal heterogeneity. Since ϕ is expressed as a disjunction (e.g., $\phi_1 \vee \cdots \vee \phi_n$), it can be partitioned into n groups. Each resulting sub-condition should be mutually exclusive and semantically meaningful. For each sub-condition ϕ_i, we create test scenarios that execute π and drive the system into states satisfying ϕ_i. If the outcomes differ across sub-conditions (e.g., π_i is activated under ϕ_i but leads to π_j under ϕ_j), this indicates that the original flow e should be split into multiple, more precise flows with distinct conditions.
- *Condition Minimal-Violation Strategy*. To assess whether the flow condition ϕ can be weakened, we generate test scenarios that execute π from states that minimally violate ϕ. To achieve this, we first identify the atomic minimal violation units in ϕ. For example, if there exists any clause in the form of $\psi_1 \wedge \psi_2$ in π, ψ_1 and ψ_2 are both taken as minimal violation conditions. For each minimal violation condition ψ_i, a test scenario is created in which ψ_i is false while all other sub-conditions hold. If π' can still be triggered with a consistent result, it indicates that ψ_i is an unnecessary constraint, suggesting that ϕ can be generalized by removing ψ_i; conversely, the result reinforces the necessity of ψ_i.
- *Condition Invariant Strategy*. To assess whether the flow condition ϕ can be strengthened, we generate test scenarios of the form $\pi \cdot \pi' \cdot \alpha \cdot \pi$. In such a scenario, the core flow $\pi \cdot \pi'$ is executed first. Next, an intermediate action α is performed to deliberately perturb the resulting app state. The action α is selected from the traces of n' that can navigate back to a page where π is executable, while ensuring that the condition ϕ is still satisfied. Finally, π is executed again. By comparing the outcomes of the two executions of π, we can determine whether the second execution leads to divergent behavior (e.g., triggering a different π'' or causing a failure), which would indicate that the current condition ϕ is overly permissive or incomplete.

Overall, we try to apply the above five LTV testing strategies to every functionality and functional flow, respectively comparing the functionality's and flow's behaviors across intentionally varied execution histories, which are prepared for the subsequent *FFG Updating* step. However, some

strategies may not be applicable in certain cases. For example, when a functional flow's condition is *True* and the target functionality has no state variables (i.e., $vars = \emptyset$), the Condition Partition Strategy and the Condition Minimal-Violation Strategy cannot be used, because the condition *True* cannot be decomposed into sub-conditions. Nevertheless, the test cases derived from these scenarios can also be used for bug detection.

For example, Figure 5 illustrates how a test scenario is generated and used to refine flow conditions through the condition invariant strategy. We take the refined FFG shown in Figure 4 as the current FFG in this example. We then apply the condition invariant strategy to the flow e = "App Navigation" $\xrightarrow{(\text{III},\text{True},\text{I})}$ "Blood Sugar Editing". This strategy generates a test scenario that first executes the flow, then navigates back to the page reached after trace III in *App Navigation*, and finally attempts to re-execute trace I of *Blood Sugar Editing*. Following the technique described in the *Scenario-driven Bug Detection* phase, we obtain the corresponding test trace. The execution reveals that trace I cannot be successfully triggered, more specifically, an alarm cannot be added after clicking the "Save" button. Since trace I of *Blood Sugar Editing* is designed to add an alarm, and this action cannot be re-executed, it indicates that the current condition *True* is too weak. A stronger condition related to the alarm state "*alarm* is not added" is inferred by the MLLM, and FFG is updated by replacing the condition of flow e with the stronger condition.

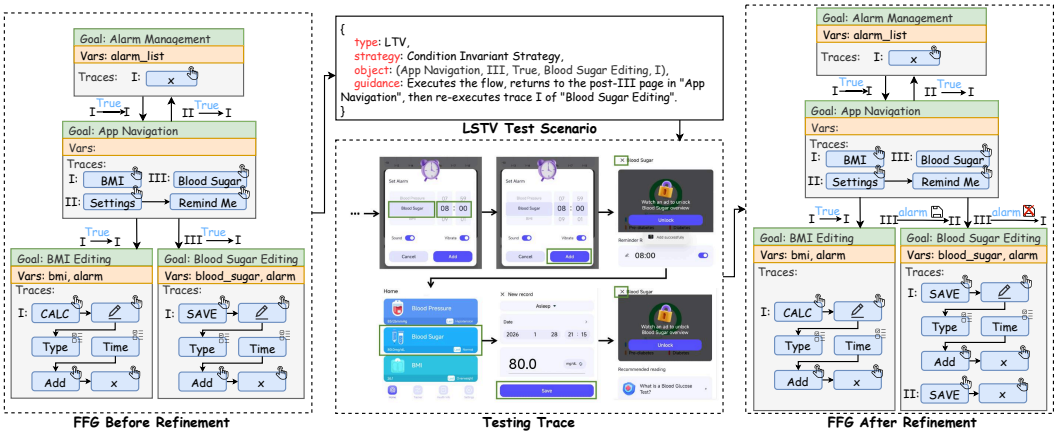


Fig. 5. Example of LTV Test Scenario for Refining Flow Conditions.

(2) Short-Term View Test Scenario Generation. Different from LTV, this type of view targets directly uncovering deep bugs through exploring the FFG, so it focuses on generating test scenarios that intensively probe current flow hypotheses. This phase employs a metamorphic testing approach, constructing scenarios by applying state transformations (metamorphic relations, MRs) to the variables appearing in a flow's condition ϕ. The MRs are organized into two complementary levels: those operating within a single flow and those spanning multiple flows based on data dependencies.

First, we design a set of **single-flow metamorphic relations**. For a given flow $e = n \xrightarrow{(\pi,\phi,\pi')} n'$, we generate alternative execution paths that still satisfy ϕ by altering the trace π within functionality n. This is achieved by applying widget-level MRs to the GUI actions in π. For instance, we may use the following transformations in single-flow MRs:

- *Hide/Show*. Hide or show dynamic GUI elements before interacting with them.
- *Change*. Change the order of filling form fields when order independence is semantically plausible.
- *Toggle*. Toggle optional GUI components (e.g., `switches`, `checkboxes`) that are not strictly required for ϕ but may affect the resulting state.

Their goal is to generate a variant execution trace π_{var} that, when executed, leads to a state satisfying ϕ and thus enables π'. In the subsequent phase, if executing this scenario fails to enable π', the scenario may be treated as a potentially buggy case. This probes the robustness of the flow against minor variations in how its condition is established.

Then, we design a set of **cross-flow metamorphic relations**. This level targets pairs of functionalities (n, n') that have no direct flow in the current FFG but share related data d, i.e., their variable sets *vars* and *vars*′ intersect. For each such pair, we identify an existing flow whose target functionality is n' and whose condition is related to the shared data d. We then construct scenarios in which a trace of n is executed to transform the data state variables so that they satisfy the condition ϕ. Typical transformations in cross-flow MRs include:

- *Create/Delete*. Node n executes a trace to create or delete the data entity d.
- *Modify-Attribute*. Node n executes a trace to alter the attributes of the data entity d.
- *Consume/Produce*. Node n executes a trace to alter the availability of a shared system or data resource associated with the data entity d.

The resulting scenarios evaluate how indirect data interactions, which lie outside the currently modeled flows, influence the behavior of the target flow e.

For example, Figure 6 illustrates how a test scenario is generated and used to uncover deep bugs through cross-flow metamorphic transitions. Taking the refined FFG from Figure 5 as the current model, we observe that the functionalities *Blood Sugar Editing* and *Alarm Management* share data entities (e.g., *alarm* and *alarm_list*). We therefore locate a flow e = "App Navigation" $\xrightarrow{(\text{III},\phi,\text{I})}$ "Blood Sugar Editing" where ϕ is "*alarm* is not added". Among the three MRs introduced above, two are applicable in this context: the *Create/Delete* MR and the *Modify-Attribute* MR. The *Consume/Produce* MR is not applicable here, because the alarm entity does not involve changes to the availability of a shared system or data resource. We choose to apply the *Create/Delete* MR. The *guidance* of this scenario is: (1) In *Blood Sugar Editing*, add an alarm; (2) in *Alarm Management*, delete it; and (3) re-add it in *Blood Sugar Editing*. By executing tests under this scenario, we successfully detect **Bug 2** shown in Figure 1. Note that the *Modify-Attribute* MR is also applicable, we modify the alarm's related attributes, but it did not reveal any bug in our experiments.

Overall, the STV scenarios are primarily designed for bug detection, yet executing the test cases derived from them can also contribute to the *FFG Updating* in Section 3.2.3. Similarly, we attempt to apply all STV strategies, though some may be inapplicable, as illustrated in the example above.

(3) Scenario-driven Bug Detection. This phase executes the testing traces guided by the high-level *LSTV test scenarios* and then performs bug detection upon them. First, **test execution** is performed to execute the testing trace guided by an LSTV test scenario (*type*, *strategy*, *object*, *guidance*). Beginning from the main page, the MLLM consults the current FFG to navigate toward the target functionality or flow. It then incrementally translates the high-level *guidance* into a sequence of widget actions. Each action is then attempted in turn. If an execution fails (e.g., the widget is missing, disabled, or its properties do not match the expected state), the failure is logged, and the MLLM uses the real-time screenshot along with the FFG to propose a recovery step or an

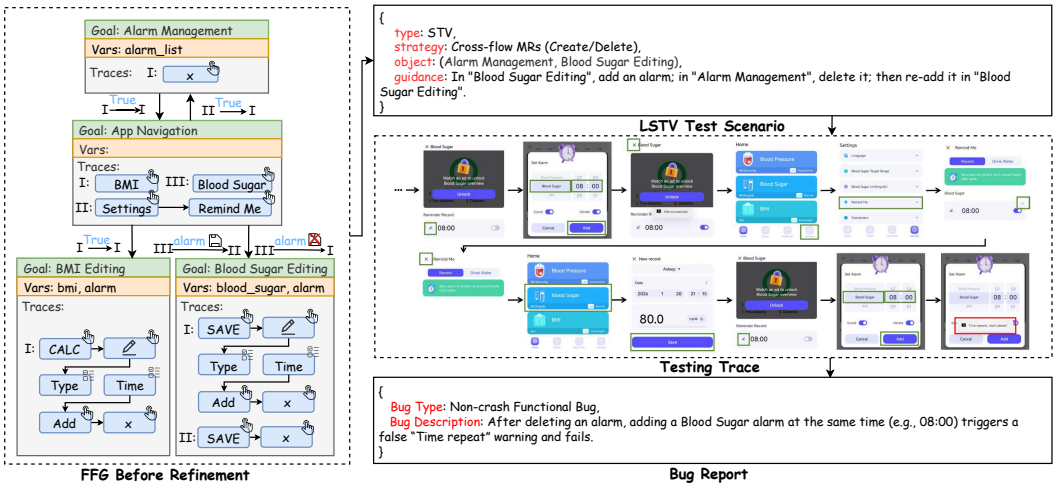


Fig. 6. Example of STV Test Scenario for Cross-Flow Metamorphic Relations.

alternative action. Throughout test execution, **bug detection** is performed to identify non-crash functional bugs. Unlike crash bugs, which manifest as obvious runtime failures, non-crash bugs lack a built-in failure signal and therefore require an explicit oracle to decide whether an observed behavior constitutes a defect. To this end, we adopt the approach established in prior work [23], in which an MLLM is augmented with a knowledge base of common non-crash bug patterns. Based on this knowledge base, the expected behavior for the testing trace can be inferred. Then we can analyze the final screenshot, widget hierarchy tree, testing trace, and system logs to compare the observed outcome with the expected one. Finally, it yields a structured bug report, which, together with the execution trace, is fed back to drive model refinement.

3.2.3 Functional Flow Graph Updating. During the *LSTV Testing* phase, the newly generated test cases are executed. This process produces additional execution traces, which are collected to iteratively refine our FFG model, progressively making it a more accurate and complete representation of the app. Within the FFG, two types of targets need to be refined: the set of functionalities N and the set of flows E. We discuss each in turn.

First, **functionality updating** is performed by analyzing the semantic similarity between the *goal* of each newly generated testing trace and those of existing functionalities. This process is similar to that used in *FFG Initializing*, where semantic similarity is evaluated via MLLM. However, while *FFG Initializing* only identifies new functionalities (i.e., the functionality *create* operation) by analyzing traces, *FFG Updating* can further determine whether a new trace triggers a functionality *merge* or *split* operation. Here are all the functionality-related operations used.

- *New Functionality Creation.* If a testing trace accomplishes a distinct task whose goal is not similar to any existing functionality, a new functionality node is created and added to N.
- *Existing Functionality Merging*. If a testing trace exhibits a goal and manipulates a set of state variables that align closely with an existing functionality n, it is merged into n. This expands n's *traces* set and may update its *vars* set, reinforcing the functionality's completeness.
- *Existing Functionality Splitting*. If a testing trace reveals that the *traces* of an existing functionality n can be partitioned into subsets that accomplish semantically independent, user-centric tasks, n is split into multiple finer-grained functionalities. This refines the model's granularity to hold the independence criterion.

Then, **flow updating** is performed by examining the relationship between a newly generated testing trace and existing flows, with a particular focus on comparing their state conditions. To identify the conditions of a new trace, we first locate its functionality-changing points. After the previous phase, each testing trace is segmented into contiguous trace segments, denoted as π and π', where π is labeled as belonging to functionality n, and the subsequent segment π' belongs to functionality n'. At each functionality-changing point, we further query the MLLM to identify the condition ϕ that holds after executing π and before initiating π'. This step yields a new flow $e_{new} = n \xrightarrow{(\pi,\phi,\pi')} n'$. We then update the flow set E by comparing the newly derived flow e_{new} with each existing flow $e_{cur} = n \xrightarrow{(\pi,\phi_{cur},\pi'_{cur})} n'_{cur}$, which shares the same source functionality n and source trace π. Based on the comparison results, the following flow-related operations are applied.

- *New Flow Creation.*
  - If no existing flow shares the same source functionality n and source trace π, a new flow $n \xrightarrow{(\pi,\phi,\pi')} n'$ is created.
  - If the new condition ϕ is logically incomparable to an existing condition ϕ_{cur} and leads to a different outcome ($\pi' \neq \pi'_{cur}$), a new flow $n \xrightarrow{(\pi,\phi,\pi')} n'$ is created.
  - If $\phi \models \phi_{cur}$ (i.e., ϕ is stronger) and $\pi' \neq \pi'_{cur}$, a new flow $n \xrightarrow{(\pi,\phi,\pi')} n'$ is created.
  - If $\phi_{cur} \models \phi$ (i.e., ϕ_{cur} is stronger) and $\pi' \neq \pi'_{cur}$, a new flow $n \xrightarrow{(\pi,\phi\wedge\neg\phi_{cur},\pi')} n'$ is created.
- *Existing Flow's Condition Strengthening.* If $\phi \models \phi_{cur}$ and $\pi' \neq \pi'_{cur}$, besides adding a new flow with condition ϕ, the existing flow's condition ϕ_{cur} is also refined to exclude ϕ: e_{cur} is updated to $n \xrightarrow{(\pi,\phi_{cur}\wedge\neg\phi,\pi'_{cur})} n'_{cur}$.
- *Existing Flow's Condition Weakening.* If $\phi_{cur} \models \phi$ and $\pi' = \pi'_{cur}$ (with $n' = n'_{cur}$), the existing condition ϕ_{cur} is generalized to the weaker condition ϕ: e_{cur} is updated to $n \xrightarrow{(\pi,\phi,\pi'_{cur})} n'_{cur}$.
- *Existing Flow's Condition Merging.* If ϕ and ϕ_{cur} are logically incomparable and $\pi' = \pi'_{cur}$ (with $n' = n'_{cur}$), the two conditions are merged by replacing e_{cur} with $n \xrightarrow{(\pi,\phi\vee\phi_{cur},\pi')} n'$.

Overall, with the above node updating and flow updating operations, the *FFG* model can be iteratively refined during continuous test scenario generation and execution. These modules work together to enable model refinement and systematic GUI exploration, thereby completing the *inter-functional-flow-oriented GUI testing* approach.

4 Evaluation

To evaluate the performance of our approach, we raise several research questions as follows:

- **RQ1 (Effectiveness and Efficiency)**: How effective and efficient is our approach in GUI exploration and bug detection on benchmark apps?
- **RQ2 (Ablation Study)**: How does the long-short-term-view-guided testing contribute to the overall effectiveness?
- **RQ3 (Practicality)**: Can our approach be successfully applied to large-scale, closed-source commercial apps to uncover new functional defects?

4.1 Experimental Setup

Implementation. FuncDroid is implemented based on UIAutomator2, which supports UI information extraction and interaction. The tool takes an Android APK as input (both open-source and closed-source apps are supported) and outputs its corresponding FFG and defect report. All implementation details are consistent with Section 3.

Benchmark. For **RQ1** and **RQ2**, we evaluate our approach on the Themis benchmark [31], which initially comprises 72 reproducible crash bugs (corresponding to 72 versions) across 20 open-source apps from F-Droid [12]. We choose Themis because it is a widely used [6, 22, 45, 48] Android GUI testing benchmark with real-world reproducible bugs, making it suitable for comparing bug detection capability with prior work. After filtering out versions that failed to launch or register properly, our final dataset includes **50** usable crash bugs (corresponding to 50 versions) across 17 apps. For **RQ3**, we evaluate our approach on 52 popular commercial apps from two official marketplaces, Google Play [13] and HUAWEI AppGallery [17], spanning 13 distinct categories, including entertainment, health, social, travel, etc. We choose these two marketplaces because they cover different Android apps and user populations, thereby increasing the diversity of the commercial-app evaluation. We filter out apps that (1) repeatedly crash during startup or normal interaction, (2) cannot be exercised by one or more tools due to incompatibility, or (3) require mandatory registration/login that cannot be bypassed or automated with scripts.

Baseline. To cover the main technical categories of Android GUI testing, we consider two approaches: traditional GUI testing and LLM-based testing. For traditional GUI testing, we select three representative tools: Droidbot [20] (random-based), Fastbot [5] (model-based), and Humanoid [21] (learning-based). For LLM-based testing, we select two representative tools: GPTDroid [22] and VisionDroid [23]. Prior work [7, 34] shows that traditional tools can benefit from LLM-based semantic guidance, while LLM-based tools can further benefit from memory-based reuse of exploration experience. To ensure a fair comparison against their best achievable performance, each baseline tool is further enhanced with a state-of-the-art, LLM-based GUI testing booster. The three traditional exploration tools are augmented with LLMDroid [34], a framework that integrates code coverage feedback to dynamically adjust the exploration strategy, thereby improving the depth and diversity of testing. While the two native LLM-based testing tools are augmented with MemoDroid [7], a system designed to enhance LLM-driven testing through a multi-layered memory mechanism that supports the retention of execution traces, summarization of exploration experience, and strategic reuse across sessions.

We deploy all baselines and our approach on a 64-bit Ubuntu 24.04 machine (64 cores, Intel CPU) and evaluate them using Android emulators. We run open-source apps on Google Android 7.1 emulators and commercial apps on Google Android 15 emulators; all emulators share the same configuration. To facilitate file access from apps, we preload the SD card with various types of external files. Following common practice [20, 22, 23], for each bug that requires login, we register a dedicated account and implement the corresponding login script; before each run, we reset the account data to avoid interference. To ensure fair and reasonable resource usage, we allocate 60 minutes to each tool per app, consistent with prior GUI testing studies [20, 22, 23, 31]. We execute each tool three times and report the best performance to mitigate potential randomness. To avoid potential data leakage, all tools use the same backend MLLM, GPT-4o [27], and no ground-truth bug information or evaluation results are provided during testing.

4.2 Effectiveness and Efficiency

Table 1 demonstrates the performance of FuncDroid and baseline approaches, which shows that FuncDroid outperforms all baseline tools across every metric (code coverage is measured as line coverage using JaCoCo). In terms of tools' exploration ability, FuncDroid achieves the highest average activity coverage (**0.55**) and code coverage (**0.40**), surpassing the best-performing baseline (VisionDroid with MemoDroid (VD+MD), 0.43 activity coverage and 0.38 code coverage) by a significant margin. This represents a relative improvement of approximately 28% in activity coverage and 5% in code coverage. This indicates that the higher coverage achieved by FuncDroid expands the exploration scope of the app, thereby increasing the likelihood of detecting potential bugs.

Regarding bug detection, FuncDroid demonstrates the strongest performance in finding crash bugs, uncovering **49** in total. This significantly exceeds the 24 (96%) crash bugs found by the best baseline (VD+MD). Furthermore, our approach exhibits a unique advantage in detecting *non-crash functional bugs*. The first three baselines (DB+LD, FB+LD, HM+LD) are built upon traditional GUI exploration tools, which lack the mechanism to detect or assert functional correctness, and thus found zero non-crash bugs. The other two baselines (GD+MD, VD+MD), augmented with advanced LLMs for semantic understanding, begin to show a nascent capability in this regard, finding 4 and 5 non-crash bugs, respectively. FuncDroid significantly outperforms them, discovering **13** non-crash bugs. Furthermore, we evaluate the accuracy of bug detection by considering a unified bug set of 79 bugs, which includes the 50 Themis ground-truth crash bugs, 16 additional crash bugs detected beyond the benchmark ground truth, and 13 detected non-crash functional bugs. FuncDroid detects 62 of them, achieving the highest detection rate of 78.5%. This superior performance directly validates the effectiveness of our inter-functional-flow-oriented testing methodology.

Table 1. Performance of Coverage and Bug Detection.

Metric \ Tool	DB+LD	FB+LD	HM+LD	GD+MD	VD+MD	FuncDroid
Avg. Coverage$_{activity}$	0.31	0.36	0.36	0.42	0.43	**0.55**
Avg. Coverage$_{code}$	0.25	0.29	0.23	0.36	0.38	**0.40**
#Bug$_{crash}$	12	15	15	22	25	**49**
#Bug$_{non-crash}$	–	–	–	4	5	**13**
#Bug$_{total}$	12	15	15	26	30	**62**
Accuracy	15.2%	19.0%	19.0%	32.9%	38.0%	**78.5%**

*The tools: **DB+LD** is Droidbot with LLMDroid, **FB+LD** is Fastbot with LLMDroid, **HM+LD** is Humanoid with LLMDroid, **GD+MD** is GPTDroid with MemoDroid, and **VD+MD** is VisionDroid with MemoDroid.*

Figure 7 summarizes the number of bugs in the overlaps detected by different tool combinations using a simplified UpSet plot. The results indicate that only FuncDroid successfully uncovered all bugs in the benchmark, including **21 bugs** that were uniquely detected by it. This indicates that our approach not only achieves higher quantitative results but also discovers a distinct class of defects that remain hidden to other testing techniques. Furthermore, the analysis reveals that the two LLM-based baselines (GD+MD and VD+MD), together with FuncDroid, jointly found 15 bugs that the three traditional testing tools (DB+LD, FB+LD, HM+LD) missed. Even after excluding the 4 non-crash bugs from this set, 11 crash bugs were still only uncovered by the LLM-based tools. This illustrates that while LLM-enhanced traditional tools (LLMDroid) improve over purely heuristic explorers, their capability remains substantially limited compared to a truly LLM-based approach. We further analyze the relationship between the number of explored flows and detected bugs over time in each app, as shown in Figure 8. The results show that FuncDroid detects more bugs as the number of explored flows increases. The blue markers on the line indicate the occurrence of unique bugs detected by FuncDroid. Notably, FuncDroid is able to uncover unique bugs within the first 10 minutes, benefiting from sufficient exploration of inter-functional flows. As more flows are explored, FuncDroid continues to identify additional unique bugs.

Moreover, to evaluate the efficiency of our approach, we compare FuncDroid with the five baseline tools by measuring their activity coverage and the number of detected bugs over varying time. Figure 9 shows that FuncDroid consistently maintains higher values in both metrics at any given time point. Notably, as indicated by the dashed lines, for both activity coverage and the number of detected bugs, FuncDroid reaches, at around 20 minutes, the peak performance that the best-performing baseline attains only at 60 minutes. Additionally, we observe that the two

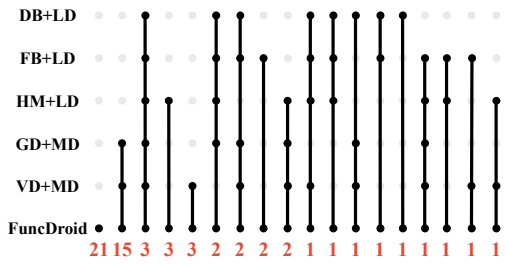


Fig. 7. Detected Bug Overlap across Different Tool Combinations.

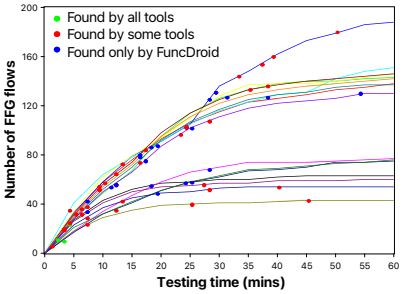


Fig. 8. Number of Flows and Detected Bugs across Apps with Varying Time.

LLM-based baselines (GD+MD and VD+MD) also outperform the three traditional tools in this efficiency comparison. Although these LLM-based tools incur additional overhead from interacting with the LLM, the experimental results indicate that the functionality awareness provided by the LLM yields a net positive gain in exploration speed and bug-finding rate, allowing them to reach meaningful coverage and detect bugs faster than traditional exploration strategies.

To further quantify the overhead introduced by LLM usage, we show the average number of LLM calls, token consumption, and estimated monetary cost per app in Table 2. The cost is estimated according to the public GPT-4o pricing at the time of our experiments based on the logged input and output tokens. For traditional exploration tools enhanced with LLMDroid, token usage tends to be relatively low. However, their bug-finding capability is also weaker than that of LLM-based tools. For LLM-based tools, GD+MD (resp. VD+MD) uses 615 (resp. 486) calls and 1.8M (resp. 2.1M) tokens. In comparison, FuncDroid requires 432 calls and 1.6M tokens per app. The analysis shows that, while achieving 96% more bugs found compared to the best LLM-based baseline (i.e., VD+MD), FuncDroid consumes only 77% of the tokens required by that baseline. Overall, FuncDroid achieves the lowest token consumption among LLM-based baselines while outperforming others on effectiveness.

Table 2. LLM Calls, Token Consumption, and Estimated Cost

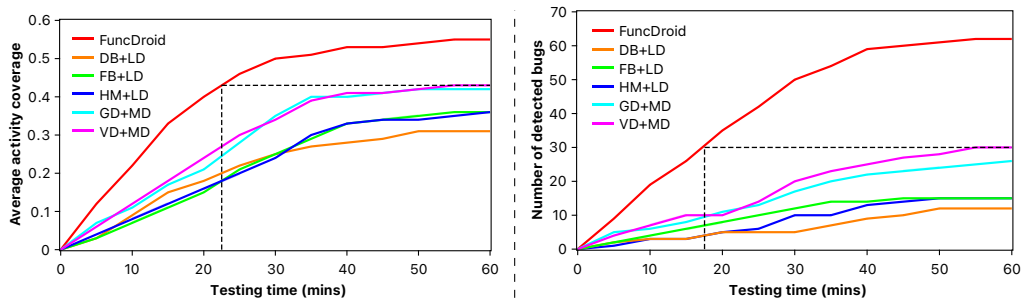
Metric	DB+LD	FB+LD	HM+LD	GD+MD	VD+MD	FuncDroid
Avg. Tokens	113,861	132,239	107,608	1,817,692	2,074,965	1,594,931
Avg. #Calls	53	59	40	615	486	432
Avg. Cost (USD)	0.71	0.83	0.67	10.11	12.97	9.97

*The tools: **DB+LD** is Droidbot with LLMDroid, **FB+LD** is Fastbot with LLMDroid, **HM+LD** is Humanoid with LLMDroid, **GD+MD** is GPTDroid with MemoDroid, and **VD+MD** is VisionDroid with MemoDroid.*

In summary, this demonstrates that FuncDroid not only achieves superior final results but also attains high effectiveness much more rapidly. This provides a strong affirmative answer to **RQ1**.

4.3 Ablation Study

In this part, we perform extra ablation studies to evaluate the contributions of the *long-short-term-view-guided testing* approach. For a more fine-grained comparison, the long-term view (LTV) test scenario generation is divided into scenarios for refining functionality definitions and flow conditions, while the short-term view (STV) test scenario generation is divided into scenarios for single-flow and cross-flow. The results are shown in Table 3, which confirms that each component independently contributes to both flow coverage and bug detection.



(a) Activity Coverage with Varying Time. (b) Bug Detection with Varying Time.

Fig. 9. Performance of Coverage and Bug Detection Ability with Varying Time.

Table 3. Results of the Ablation Study on FuncDroid's Long-Short-Term View Testing and FFG Initialization

Config	Avg. #Flow	#Bug$_{total}$	#Bug$_{crash}$	#Bug$_{non-crash}$
FuncDroid	**105**	**62**	**49**	**13**
w/o FFG-Init	87 (↓ 17%)	55 (↓ 11%)	44 (↓ 10%)	11 (↓ 15%)
w/ 5-min FFG-Init	95 (↓ 10%)	59 (↓ 5%)	47 (↓ 4%)	12 (↓ 8%)
w/o LTV-Func	71 (↓ 32%)	48 (↓ 23%)	38 (↓ 22%)	10 (↓ 23%)
w/o LTV-Flow	89 (↓ 15%)	50 (↓ 19%)	39 (↓ 20%)	11 (↓ 15%)
w/o LTV	65 (↓ 38%)	43 (↓ 31%)	34 (↓ 31%)	9 (↓ 31%)
w/o STV-Single-Flow	97 (↓ 8%)	46 (↓ 26%)	37 (↓ 24%)	9 (↓ 31%)
w/o STV-Cross-Flow	101 (↓ 4%)	49 (↓ 21%)	39 (↓ 20%)	10 (↓ 23%)
w/o STV	94 (↓ 10%)	39 (↓ 37%)	33 (↓ 33%)	6 (↓ 54%)
w/o LSTV	58 (↓ 45%)	27 (↓ 56%)	23 (↓ 53%)	4 (↓ 69%)

The FFG initialization time mainly affects the early construction of functional flows, while its impact on the final bug detection results is relatively moderate. Our full configuration uses 10-minute FFG initialization, while the ablation study considers 0-minute and 5-minute initialization. Table 3 shows the results under the default 60-minute testing time. Without FFG initialization, FuncDroid decreases by 17% in average flows and 11% in detected bugs; with 5-minute initialization, the decreases are reduced to 10% and 5%, respectively. This indicates that FFG initialization accelerates early flow construction and bug discovery. Furthermore, when we extended the total testing time by 30 minutes, the configurations with 0-minute or 5-minute initialization achieved results comparable to the 10-minute initialization configuration, demonstrating that LTV-driven iterative refinement can compensate for weaker initial traces over longer exploration.

Disabling the **Test Scenarios Generation for Functionality Definitions (*w/o* LTV-Func)** phase within LTV reduces the average number of flows in FFG by 32% and the total bugs found by 23% (from 62 to 48). This indicates that refining the functionality definitions through completeness and independence is essential for building more accurate functionalities, which in turn yields a richer set of flows. Similarly, disabling the **Test Scenarios Generation for Flow Conditions (*w/o* LTV-Flow)** phase within LTV reduces the average number of flows by 15% and total bugs by 19%. This result highlights the importance of sharpening state preconditions ϕ; without such refinement, the model fails to capture precise enabling conditions, leading to both fewer flows and missed bugs. Although it contributes fewer new flows than LTV-Func, the flows it refines become more reliable for detecting bugs. Together, these components are critical for constructing a high-fidelity FFG: one expands the flows, while the other ensures its logical accuracy, together forming a robust foundation for effective testing. Disabling the entire **Long-Term View Testing**

(*w/o* LTV) reduces the average number of flows by 38% and total bugs by 31%. This underscores that the combined effort of refining both functionalities and flows is critical for constructing a high-fidelity FFG, which serves as the foundation for effective testing.

Disabling the **Single-Flow Metamorphic Testing (*w/o* STV-Single)** phase within STV reduces the total bugs found by 26% (from 62 to 46), with a decrease of 8% in the average number of flows. This indicates that varying how a flow's condition is satisfied is essential for effectively uncovering crash bugs triggered by minor GUI variations. Similarly, disabling the **Cross-Flow Metamorphic Testing (*w/o* STV-Cross)** phase reduces the total number of bugs by 21% and the average number of flows by 4%, highlighting the importance of testing indirect data dependencies across functionalities. Without exploring these shared data dependencies, the testing misses subtle non-crash bugs that arise from complex, multi-functional interactions. Disabling the entire **Short-Term View Testing (*w/o* STV)** reduces the total bugs by 37% and the average number of flows by 10%. This underscores that the combined power of both metamorphic testing strategies is critical for intensive, in-depth exploration based on the FFG model, directly driving the detection of a wide spectrum of interaction bugs, especially non-crash bugs, which see a 54% decline.

Disabling the combined **Long-Short Term View Testing (*w/o* LSTV)**—which removes both the model-refinement capability of LTV and the intensive testing capability of STV—reduces the average number of flows by 45% and the total bugs by 56%. This configuration leaves only the initial FFG bootstrap without any iterative refinement or exploitation. The drastic decline in flow count highlights the indispensable role of LTV in refining the FFG, while the severe drop in bug detection, with non-crash bugs plummeting by 69%, underscores that STV is essential for bug detection.

To evaluate whether FuncDroid depends on a specific MLLM backend, we further conducted a supplementary cross-model experiment on a subset of apps by replacing GPT-4o with Gemini-2.5-flash and Claude-haiku-4.5. As shown in Table 4, FuncDroid achieves comparable results across different backends, suggesting that its effectiveness is not tied to a single foundation model.

Table 4. Effect of Different MLLM Backends

MLLM Backend	Avg. Coverage$_{activity}$	Avg. Coverage$_{code}$	#Bug$_{crash}$	#Bug$_{non-crash}$	#Bug$_{total}$
Gemini-2.5-flash	0.56	0.41	49	13	62
Claude-haiku-4.5	0.52	0.38	47	12	59
GPT-4o	0.55	0.40	49	13	62

4.4 Practicality

Finally, to assess the practicality of FuncDroid on complex real-world apps, we conduct an extensive evaluation on a benchmark consisting of 52 highly downloaded commercial mobile apps spanning 13 categories. Our tool FuncDroid successfully detects **18** non-crash bugs on this benchmark. Table 5 presents detailed information about the apps in which these bugs are found. We further conduct the same evaluation using the two LLM-based baselines from RQ1, namely GPTDroid and VisionDroid, both augmented with MemoDroid. Neither baseline discovers any new bugs beyond those already reported. As shown in the last two columns in Table 5, GPTDroid with MemoDroid (GD+MD) identifies only **4** bugs, while VisionDroid with MemoDroid (VD+MD) detects **6** bugs. Notably, all bugs found by GD+MD are also covered by VD+MD.

The bug reporting for commercial apps is often challenging, but we still attempted to contact developers to validate the bugs via in-app feedback systems, email communications, etc. Among the **18** bugs reported by FuncDroid, **2** have been fixed by developers, **2** have been confirmed; notably, none of the reported bugs were rejected. Their latest status is also shown in Table 5.

Table 5. Bugs Detected by FuncDroid on Commercial Apps.

Id	App Name	Category	Download	Size (MB)	Status	GD+MD	VD+MD
1	YouTuBe Kids	Entertainment	500M+	27	waiting	✗	✗
2	PiPiXia	Entertainment	100M+	107	fixed	✗	✗
3	ZuiYou	Entertainment	100M+	159	fixed	✗	✗
4	Blood Pressure	Health	10M+	26	waiting	✗	✗
5	MeiYou	Health	100M+	175	waiting	✓	✓
6	AnJuKe	Lifestyle	400M+	88	waiting	✗	✗
7	Live Earth Map	Navigation	1M+	28	waiting	✗	✗
8	KugouMusic	Music	1B+	162	waiting	✗	✗
9	FOX LOCAL	News	500K+	18	waiting	✗	✗
10	Substack	News	5M+	31	waiting	✓	✓
11	PDF File Reader	Productivity	5M+	24	waiting	✗	✗
12	Instagram	Social	5B+	97	waiting	✓	✓
13	Wechat	Social	1B+	243	confirmed	✗	✗
14	Victory+	Sports	100K+	39	confirmed	✗	✓
15	QR Code&Barcode Scanner	Tools	1M+	20	waiting	✗	✗
16	Greyhound: Buy Bus Tickets	Travel	1M+	14	waiting	✗	✓
17	Allegiant	Travel	5M+	54	waiting	✓	✓
18	Qunar Travel	Travel	200M+	82	waiting	✗	✗

5 Related Work

Automated GUI testing for mobile applications has evolved through several distinct phases, broadly categorized by their primary objective: maximizing exploration coverage or understanding and testing application functionality. Accordingly, we discuss related work from these two perspectives.

Coverage-oriented GUI Testing. The primary goal of this line of work is to explore as many distinct states and activities within an application as possible. Early random testing methods [9–11, 20] and fuzzing methods [3, 18, 44] generate unpredictable event sequences. While efficient and simple, their lack of structural or semantic guidance often results in low coverage and redundant exploration. To mitigate these limitations, model-based testing techniques [4, 5, 14–16, 20, 24, 26, 30, 36, 42, 43] construct explicit representations of the GUI, such as state transition graphs, to guide systematic exploration. These models help avoid redundant states but often suffer from state explosion and, more critically, remain agnostic to the semantic purpose of the interfaces they traverse. To further enhance exploration intelligence, learning-based testing approaches [19, 21, 28, 33] leverage techniques from reinforcement learning and deep learning. By learning interaction patterns from data, these methods prioritize promising actions and generate test cases that align more closely with actual user behaviors. However, the learned interaction patterns are often sensitive to the available exploration feedback or historical execution experience, limiting their generalization to rapidly evolving application functionalities.

Functionality-oriented GUI Testing. This line of work shifts focus from covering states to understanding and executing user-centric tasks. To effectively test functionality, property-based testing method [32, 35, 41] generates input sequences to verify specific semantic properties of the apps. However, they typically lack a deeper understanding of functional behaviors. The advent of LLMs (especially MLLMs) has been pivotal in bridging this gap, enabling a leap in semantic interpretation of GUI content [7, 22, 23, 29, 34, 37–39, 46, 47]. Early works such as AppAgent [46] and AutoDroid [38] focus on multimodal perception and task planning. However, these methods still focus on the interface or widget level and do not model the functional semantics or logic between functions. GPTDroid [22] and VisionDroid [23] employ LLMs to directly interpret GUI pages, summarize page sequences into reusable functional patterns, and then apply these patterns

to guide further exploration. Although such approaches improve semantic comprehension and planning, they still focus primarily on individual functionalities. Frameworks such as LLMDroid [34] and MemoDroid [7] enhance existing testing tools by constructing knowledge bases that capture app-specific functional characteristics. These frameworks distill common interaction patterns and exploration strategies from historical testing sessions to steer exploration more efficiently.

In summary, while coverage-oriented methods provide broad but shallow exploration, and modern LLM-based methods enable functionality-aware task execution, the field still lacks a model that explicitly represents and facilitates the systematic testing of inter-functional flows, which is the precise gap our work addresses with the inter-functional-flow-oriented GUI testing method.

6 Threats to Validity

Dataset Limitations. Our evaluation covers Android apps from Themis, Google Play, and HUAWEI AppGallery, including both open-source and commercial apps. However, the dataset contains only 69 apps in total. The distribution of crashes is uneven: some apps have multiple confirmed bugs, while others have few or none. This limits the statistical power of our bug detection evaluation. Moreover, the results may not directly generalize to other platforms such as iOS or HarmonyOS, and all experiments are conducted on emulators, whose behaviors may differ from real devices.
Bug Detection Limitations. FuncDroid is effective for bugs caused by explicit inter-functional dependencies, especially those involving shared state or data entities. However, it may miss bugs that require very long cross-functional sequences or depend on external asynchronous events. Extending FuncDroid to model richer environmental and temporal conditions is left as future work.
Risk of Data Leakage. Our approach uses the MLLM for goal inference and semantic comparisons. If the MLLM has been pre-trained on data that includes the test apps or similar bug patterns, there is a risk of data leakage, which could artificially inflate performance.

7 Conclusion

Mobile application testing plays a critical role in ensuring the reliability and quality of software systems that are deeply embedded in everyday life. However, the key limitation of functionality-oriented GUI testing lies in the absence of a systematic modeling and testing approach for inter-functional interactions. This work addresses this limitation by providing a scalable and practical solution that bridges the capability gap in detecting complex, interaction-dependent bugs. By focusing on the coordinated interactions among functionalities at a higher level of abstraction, this work advances automated GUI testing beyond traditional coverage- or single-functionality-centric paradigms. Specifically, we propose a novel **Functional Flow Graph** (FFG) to explicitly model inter-functional interactions, and introduce an *inter-functional-flow-oriented testing* methodology that iteratively refines the FFG while guiding testing toward deep interaction paths. Extensive experimental results demonstrate the effectiveness, efficiency, and practical applicability of our approach in detecting both crash and non-crash bugs. Looking ahead, future work will focus on extending this approach to additional mobile platforms (e.g., iOS, HarmonyOS) and enhancing the knowledge base for non-crash bug detection, further improving the overall bug detection capability.

Acknowledgement

This work is supported by the National Natural Science Foundation of China (No.62132020).

Data Availability

The implementation of FuncDroid and all experimental datasets and results are publicly available at https://github.com/SQUARE-RG/FuncDroid. Additionally, we provide supplementary experimental information and results at https://fim-bugs.sites.veilaxis.com.

References

[1] Alibaba Cloud. 2026. Qwen. https://qwen.ai/home.

[2] Anthropic. 2026. Claude. https://claude.com/.

[3] Michael Auer, Andreas Stahlbauer, and Gordon Fraser. 2023. Android Fuzzing: Balancing User-Inputs and Intents. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 37–48. doi:10.1109/ICST57152.2023.00013

[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps *(OOPSLA '13)*. 641–660. doi:10.1145/2509136.2509549

[5] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System. In *2020 IEEE/ACM 15th International Conference on Automation of Software Test (AST)*. 93–96. doi:10.1145/3387903.3389308

[6] Mengzhuo Chen, Zhe Liu, Chunyang Chen, Junjie Wang, Boyu Wu, Jun Hu, and Qing Wang. 2025. Standing on the Shoulders of Giants: Bug-Aware Automated GUI Testing via Retrieval Augmentation. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 825–846. doi:10.1145/3715755

[7] Mengzhuo Chen, Zhe Liu, Chunyang Chen, Junjie Wang, Yangguang Xue, Boyu Wu, Yuekai Huang, and Qing Wu, Libinand Wang. 2025. Beyond Static GUI Agent: Evolving LLM-based GUI Testing via Dynamic Memory *(ASE 2025)*. doi:10.1109/ASE63991.2025.00135 Accepted for publication.

[8] DeepSeek AI. 2026. DeepSeek. https://www.deepseek.com/.

[9] Android Developers. 2012. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey.

[10] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 481–492. doi:10.1145/3377811.3380402

[11] Jordan Doyle, Takfarinas Saber, Paolo Arcaini, and Anthony Ventresque. 2021. Improving Mobile User Interface Testing with Model Driven Monkey Search. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 138–145. doi:10.1109/ICSTW52544.2021.00034

[12] F-Droid Community. 2026. F-Droid. https://f-droid.org/.

[13] Google. 2026. Google Play. https://play.google.com/.

[14] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications Via Model Abstraction and Refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 269–280. doi:10.1109/ICSE.2019.00042

[15] Jinlong He, Binru Huang, Changwei Xia, Hengqin Yang, Jiwei Yan, and Jun Yan. 2027. HACMony: automatically detecting hopping-related audio-stream conflict issues on HarmonyOS. *Frontiers of Computer Science* 21 (2027), 2106201–.

[16] Jinlong He, Zhilin Wu, and Taolue Chen. 2025. Formalization of Android Activity-Fragment Multitasking Mechanism and Static Analysis of Mobile Apps. *Form. Asp. Comput.* 37, 2, Article 17 (April 2025), 86 pages. doi:10.1145/3708562

[17] Huawei. 2026. HUAWEI AppGallery. https://consumer.huawei.com/en/mobileservices/appgallery/.

[18] Gennaro Imparato. 2015. A Combined Technique of GUI Ripping and Input Perturbation Testing for Android Apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 760–762. doi:10.1109/ICSE.2015.241

[19] Yuanhong Lan, Yifei Lu, Zhong Li, Minxue Pan, Wenhua Yang, Tian Zhang, and Xuandong Li. 2024. Deeply Reinforcing Android GUI Testing with Deep Reinforcement Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. 1–13. doi:10.1145/3597503.3623344

[20] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-Guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 23–26. doi:10.1109/ICSE-C.2017.8

[21] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 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)*. 1070–1073. doi:10.1109/ASE.2019.00104

[22] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 2023. Make LLM a Testing Expert: Bringing Human-Like Interaction to Mobile GUI Testing via Functionality-Aware Decisions. *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)* (2023), 1222–1234. doi:10.1145/3597503.3639180

[23] Zhe Liu, Cheng Li, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Yawen Wang, Jun Hu, and Qing Wang. 2025. Seeing is Believing: Vision-driven Non-crash Functional Bug Detection for Mobile Apps. *IEEE Transactions on Software Engineering* (2025), 1–14. doi:10.1109/TSE.2025.3614469

[24] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105. doi:10.1145/2931037.2931054

[25] Meta AI. 2026. Llama. https://github.com/meta-llama/llama3.

[26] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *Proceedings of the 38th International Conference on Software Engineering*. 559–570. doi:10.1145/2884781.2884853

[27] OpenAI. 2026. GPT-4o. https://openai.com/index/gpt-4o/

[28] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 153–164. doi:10.1145/3395363.3397354

[29] Yunpeng Song, Yiheng Bian, Yongtao Tang, Guiyu Ma, and Zhongmin Cai. 2024. VisionTasker: Mobile Task Automation Using Vision Based UI Understanding and LLM Task Planning. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology*. ACM, 1–17. doi:10.1145/3654777.3676386

[30] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256. doi:10.1145/3106237.3106298

[31] Ting Su, Jue Wang, and Zhendong Su. 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)*. 119–130. doi:10.1145/3468264.3468620

[32] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 156 (Oct. 2021), 31 pages. doi:10.1145/3485533

[33] Tuyet Vuong and Shingo Takada. 2019. Semantic analysis for deep Q-network in android GUI testing. In *Proceedings - SEKE 2019*. 123–128. doi:10.18293/SEKE2019-080

[34] Chenxu Wang, Tianming Liu, Yanjie Zhao, Minghui Yang, and Haoyu Wang. 2025. LLMDroid: Enhancing Automated Mobile App GUI Testing Coverage with Large Language Model Guidance. *Proceedings of the ACM on Software Engineering* 2, FSE (2025), 1001–1022. doi:10.1145/3715763

[35] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 434–446. doi:10.1145/3540250.3549170

[36] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 469–480. doi:10.1145/3377811.3380382

[37] Junyang Wang, Haiyang Xu, Jiabo Ye, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-Agent: Autonomous Multi-Modal Mobile Device Agent with Visual Perception. doi:10.48550/arXiv.2401.16158

[38] Hao Wen, Yuanchun Li, Guohong Liu, Shanhui Zhao, Tao Yu, Toby Jia-Jun Li, Shiqi Jiang, Yunhao Liu, Yaqin Zhang, and Yunxin Liu. 2024. AutoDroid: LLM-powered Task Automation in Android. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*. 543–557. doi:10.1145/3636534.3649379

[39] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. DroidBot-GPT: GPT-powered UI Automation for Android. doi:10.48550/arXiv.2304.07061

[40] Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2019. Analyses for specific defects in android applications: a survey. *Front. Comput. Sci.* 13, 6 (Dec. 2019), 1210–1227. doi:10.1007/s11704-018-7008-1

[41] Yiheng Xiong, Ting Su, Jue Wang, Jingling Sun, Geguang Pu, and Zhendong Su. 2024. General and Practical Property-based Testing for Android Apps. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 53–64. doi:10.1145/3691620.3694986

[42] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 658–668. doi:10.1109/ASE.2015.76

[43] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. Springer-Verlag, 250–265. doi:10.1007/978-3-642-37057-1_19

[44] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag *(MoMM '13)*. 68–74. doi:10.1145/2536853.2536881

[45] Juyeon Yoon, Robert Feldt, and Shin Yoo. 2024. Intent-Driven Mobile GUI Testing with Autonomous Large Language Model Agents. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 129–139. doi:10.1109/ICST60714.2024.00020

[46] Chi Zhang, Zhao Yang, Jiaxuan Liu, Yanda Li, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2025. AppAgent: Multimodal Agents as Smartphone Users. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. Association for Computing Machinery, 70:1–70:20. doi:10.1145/3706598.3713600

[47] Jiayi Zhang, Chuang Zhao, Yihan Zhao, Zhaoyang Yu, Ming He, and Jianping Fan. 2024. MobileExperts: A Dynamic Tool-Enabled Agent Team in Mobile Devices. doi:10.48550/arXiv.2407.03913

[48] Yuxin Zhang, Sen Chen, Xiaofei Xie, Zibo Liu, and Lingling Fan. 2025. Scenario-Driven and Context-Aware Automated Accessibility Testing for Android Apps. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 2777–2789. doi:10.1109/ICSE55347.2025.00093

Received 2026-01-30; accepted 2026-06-25