# PropGen: Automated Property Generation for Property-Based Testing of Mobile Apps

Yiheng Xiong
Singapore Management University
Singapore, Singapore
yihengx98@gmail.com

Shiwen Song*
Singapore Management University
Singapore, Singapore
swsong@smu.edu.sg

Bo Ma
East China Normal University
Shanghai, China
boma@stu.ecnu.edu.cn

Ting Su
East China Normal University
Shanghai, China
tsu@sei.ecnu.edu.cn

Xiaofei Xie
Singapore Management University
Singapore, Singapore
xfxie@smu.edu.sg

## Abstract

Mobile apps often suffer from functional bugs that do not cause crashes but instead manifest as incorrect behaviors under specific user interactions. Such bugs are difficult to detect by conventional automatic testing techniques because they often lack explicit *test oracles*. Property-based testing can effectively expose them by specifying intended behavior as properties and checking them under diverse interactions. However, its practical use is limited by the reliance on manually written properties, which are difficult and expensive to construct.

To address this limitation, this paper explores the use of large language models (LLMs) to automate property construction for property-based testing of mobile apps. This is challenging in two ways. *First*, it is difficult to systematically uncover and execute diverse app functionalities. *Second*, it is difficult to derive valid properties from functionality execution results. To address these challenges, we introduce PropGen, which infers candidate app functionalities as hypotheses from GUI states, validates each hypothesis by executing it to collect behavioral evidence, synthesizes properties from the collected evidence, and refines imprecise properties based on testing feedback. We implemented PropGen and evaluated it on 12 real-world Android apps. The results show that PropGen can effectively identify and execute app functionalities, generate valid properties, and refine most imprecise ones. Across all apps, PropGen inferred 1,210 valid functionalities and correctly executed 977 of them, compared with 491 and 187 for the baseline. It generated 985 properties, 912 of which were valid, and successfully refined 118 of 127 imprecise ones exposed during testing. Using the resulting properties, we found 25 previously unknown functional bugs, many of which were missed by existing testing techniques.

## CCS Concepts

• **Software and its engineering → Software testing and debugging**.

*Corresponding author.

This work is licensed under a Creative Commons Attribution 4.0 International License.
*ASE '26, Munich, Germany*
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837536

## Keywords

Property-Based Testing, Mobile App Testing, Property Synthesis, Large Language Models

**ACM Reference Format:**
Yiheng Xiong, Shiwen Song, Bo Ma, Ting Su, and Xiaofei Xie. 2026. PropGen: Automated Property Generation for Property-Based Testing of Mobile Apps. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837536

## 1 Introduction

Mobile apps are highly interactive and stateful systems whose functionalities are largely driven by user interface interactions. Despite extensive testing efforts, functional bugs (*e.g.*, incorrect interaction logic) remain prevalent in real-world apps [72]. These bugs often do not manifest as crashes, making them difficult to detect using conventional GUI testing techniques that mainly emphasize code coverage or crash discovery [9, 19, 33, 35, 37, 50, 61, 64]. Manual testing (*e.g.*, writing GUI tests) is widely used in practice to validate the functional correctness of mobile apps [25, 30]. However, it is brittle, costly to maintain, and typically covers only pre-defined *happy paths*, often missing non-trivial functional bugs [71].

Property-based testing (PBT) offers a promising direction for addressing this challenge [6]. Recent work has demonstrated that carefully designed properties can reveal non-trivial functional bugs that are missed by other techniques [29, 55, 71]. In mobile app testing, developers specify expected behaviors as properties, and PBT automatically generates a large number of GUI events to explore diverse GUI states and check whether these properties hold. Compared with validating only a fixed set of manually crafted GUI test cases, this paradigm provides a more efficient way to assess functional correctness across diverse GUI states.

However, the effectiveness of PBT fundamentally depends on the availability of high-quality properties, whose manual construction remains a major barrier to practical adoption [17, 18]. This challenge is especially pronounced for mobile apps, where explicit behavioral specifications are often unavailable. As a result, testers must manually understand app functionalities, abstract their expected behaviors into executable properties, and refine the properties when reported violations are false positives. This manual and iterative process substantially limits the broader adoption of PBT.

To address this limitation, a promising direction is to leverage the reasoning and code-generation capabilities of Large Language Models (LLMs) to automate property construction. However, directly asking an LLM to generate properties is often unreliable: mobile apps lack explicit behavioral specifications, and without concrete evidence of how an app actually behaves, LLMs tend to hallucinate properties that the app never satisfies. Instead, our idea is to anchor property construction in the app's own functionalities: we collect the functionalities of the app, and generate the properties they are expected to satisfy.

**Challenges.** Realizing this idea faces two key challenges. *First*, how to systematically discover the functionalities of an app? Mobile apps rarely come with specifications: what an app can do, and how, is implicit in its user interface rather than stated anywhere. Its functionalities are further scattered across many GUI states, and among the enormous number of possible event sequences, only a few complete a functionality. Those that remain undiscovered yield no properties, leaving the corresponding app behavior unchecked. *Second*, how to generate properties from the discovered functionalities? A functionality reveals only what the app can do, whereas a property asserts what must hold every time it is used. Generating properties thus requires closing the gap between observed and expected behavior, and no specification is available to tell whether it has been closed correctly.

**Our approach.** To address the first challenge, we design a functionality hypothesis-guided exploration strategy that systematically uncovers app functionalities and executes them to collect behavioral evidence, *i.e.*, concrete records of how each functionality is carried out. Given a GUI state, it infers candidate functionalities as hypotheses, grounding each of them in a concrete actionable widget so that the hypothesis is directly executable. Each hypothesis is then validated by execution: the app is driven to carry out the hypothesized functionality, and the resulting trace both confirms the hypothesis and provides the behavioral evidence. During this process, it incrementally expands the functionality pool when new functionalities are inferred, reuses previously inferred functionalities across recurrent GUI states, and falls back to lightweight random exploration only when no unexecuted functionality remains in the current GUI state.

To address the second challenge, instead of generating properties directly from execution results, our approach first abstracts each functionality execution trace into a compact condition–event–outcome representation, derives a structured property description in natural language, and then translates it into an executable property. Because properties inferred from observed behavior may still be imprecise, it refines those that trigger false positives during testing. By grounding each refinement in the behavioral evidence from which the property was derived, it localizes the imprecision to one of three components, the *precondition*, *interaction*, or *postcondition*, and revises only that component. This makes the generated properties more precise while preserving their original intent.

**Evaluation and results.** We implemented our approach as a tool named PropGen and evaluated it on 12 real-world popular and diverse Android apps. Across all apps, PropGen inferred 1,282 functionalities, of which 1,210 (94.4%) were valid and 977 (76.2%) were correctly executed. In comparison, the baseline approach inferred 575 functionalities, with 491 valid and 187 correctly executed. It further generated 985 property descriptions, 912 (92.6%) of which were valid. During property-based testing, 127 properties were found to be imprecise, and 118 (92.9%) of them were successfully refined by our refinement technique. Using the generated properties, we found 25 previously unknown functional bugs in the latest versions of the subject apps, whereas existing functional testing techniques could find only 3 of them in practice. These results demonstrate the effectiveness of our approach in automating property construction for mobile apps, as well as the bug-finding capability of the resulting properties.

In summary, this paper makes the following contributions:

- At the conceptual level, we propose a novel approach that automatically explores app functionalities and generates properties from runtime behavioral evidence, without relying on manual specifications.
- At the technical level, we design a hypothesis-guided exploration technique that infers functionalities from GUI states and collects behavioral evidence from their executions, together with a property synthesis technique that derives properties from such evidence and refines imprecise ones through testing feedback.
- At the empirical level, we implement our approach as PropGen and evaluate it on 12 real-world Android apps. PropGen generates 912 valid properties, refines 92.9% of the imprecise ones, and uncovers 25 previously unknown functional bugs.

## 2 Background

**Property-based testing.** Property-based testing is a powerful testing methodology that validates whether a program satisfies general properties rather than specific input-output examples [6]. Instead of writing example-based test cases, testers specify high-level properties that describe the expected behavior of the system. Then, a PBT framework automatically generates a large number of test inputs and executes them to verify whether the properties hold. For example, for a sorting function `sort`, rather than enumerating concrete examples (*e.g.*, `sort([3,1,2])== [1,2,3]`), one can define a general property such as idempotence: `sort(sort(x)) = sort(x)`. The PBT tool then generates a large number of inputs to verify this property and reports any violation as a counterexample.

**GUI state, event, and functionality.** Android applications are GUI-driven and event-based. When an app $\mathcal{A}$ runs, its runtime state is represented by its current GUI layout, which we denote as a *GUI state* $s$. A GUI state corresponds to a hierarchical tree $\ell$, whose nodes are GUI widgets $w$ (*e.g.*, `Button`, `TextView`, `EditText`) with attributes (*e.g.*, `text`, `resourceId`) and interaction capabilities. User interactions are modeled as *events*. An event is defined as $e = \langle t, w, v \rangle$, where $t$ is the event type (*e.g.*, click), $w$ is the target widget, and $v$ is optional data (*e.g.*, text input). An app execution is modeled as a sequence of events. Given $E = [e_1, \ldots, e_n]$, executing $\mathcal{A}$ produces a trace $\tau = s_0 \xrightarrow{e_1} s_1 \rightarrow \cdots \xrightarrow{e_n} s_n$, or $s_0 \overset{E}{\rightsquigarrow} s_n$, where $s_0$ is the initial state.

We define a *functionality* as a tuple $f = \langle d, T \rangle$, where $d$ is a natural-language description of the semantic intent of the functionality (*e.g.*, "create a note"), and $T = \{\tau_1, \tau_2, \ldots\}$ is the set of execution traces that realize this intent. Executing any $\tau \in T$ completes the functionality and changes the app state. These traces share the intent $d$ but may differ in input data, events, and starting

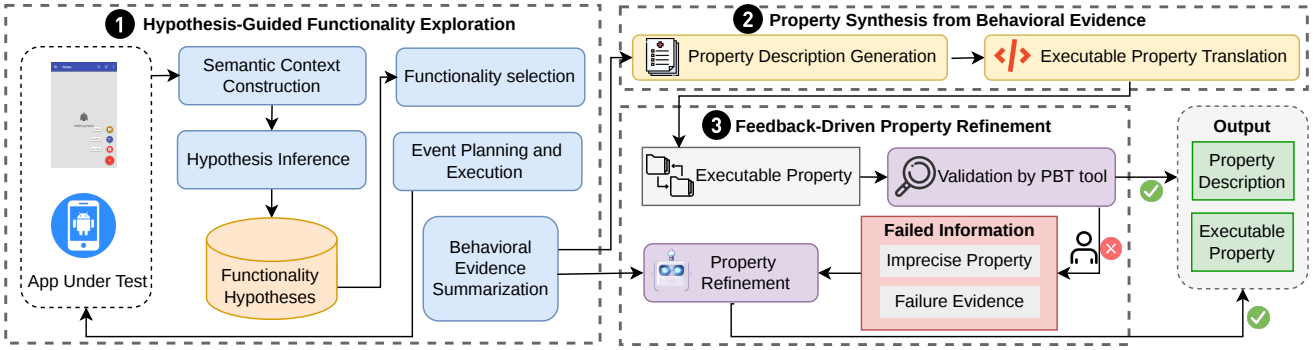


**Figure 1: Overview of PropGen.**

**Figure 2: An Illustrative Example of Behavioral Evidence Construction, Property Synthesis, and Feedback-Driven Property Refinement for a Note-Taking App.**

GUI states. We refer to what can be observed when a trace $\tau \in T$ is executed, *i.e.*, the GUI states it traverses and the changes it induces, as a *behavior* of $f$. Each trace thus witnesses one behavior of the same functionality, and such observed behaviors form the *behavioral evidence* from which properties are derived in this paper.

**Property-based testing for mobile apps.** A property is defined as a tuple $\phi = \langle P, I, Q \rangle$, where the *precondition* $P$ characterizes the GUI states in which the property applies, $I$ is the *interaction scenario*, *i.e.*, a sequence of statements that defines how to interact with the app, which may also contain control-flow constructs such as conditionals and loops, and the *postcondition* $Q$ specifies the expected outcome after executing $I$. During testing, executing $I$ from a GUI state $s$ that satisfies $P$ leads to a new state $s'$, and the property holds if $\forall s.\ (s \models P \wedge s \overset{I}{\rightsquigarrow} s') \Rightarrow s' \models Q$. Otherwise, a property violation is reported.

## 3 Approach

**Overview.** Given an app, PropGen automatically generates properties from runtime executions and further refines imprecise ones based on testing feedback. Figure 1 presents the overall workflow, which consists of three stages. First (§3.1), PropGen performs hypothesis-guided dynamic exploration to construct behavioral evidence. For each encountered GUI state $s$, it infers functionality hypotheses grounded in the visible UI widgets, executes selected functionalities, and summarizes the resulting execution traces into behavioral evidence for downstream property generation. Second (§3.2), PropGen synthesizes properties directly from the behavioral evidence constructed in the first stage. Specifically, it first generates natural-language property descriptions that capture the intended condition–event–outcome relation, and then translates them into

executable properties. Third (§3.3), PropGen validates the generated properties with the PBT framework, and refines those whose reported violations are manually confirmed to be false positives rather than real bugs.

**Illustrative example.** Figure 2 illustrates the workflow of PropGen on a note-taking app. From the page in Figure 2(a), PropGen infers candidate functionality hypotheses and selects one of them, *i.e.*, attaching a photo to the note. It then executes the selected hypothesis by opening the attachment menu, choosing Camera, taking a photo, and returning to the note page. The resulting trace is summarized into structured behavioral evidence (Figure 2(c)), from which PropGen synthesizes an executable property for this functionality (Figure 2(d)).

This property, however, is imprecise. Testing it with the PBT framework Kea reports violations that manual inspection confirms to be false positives: the postcondition requires the app to return to a page containing the text "Notes" and the attachment to appear as a thumbnail, but the app may legitimately return to either the Notes or the Archive page, depending on where the note was opened, and under reduced view the attachment appears as a compact icon (Figure 2(b)). Guided by this feedback, PropGen relaxes both assertions to accept either return page, checked through a menu button that both contain, and either attachment representation. Re-testing then uncovers a new functional bug: opening audio recording before taking a photo prevents the photo attachment from appearing.

**Algorithm 1** Behavioral Evidence Construction

**Require:** Target app $\mathcal{A}$, time budget $B$
**Ensure:** Behavioral evidence set $\hat{\mathcal{T}}$

1: Launch $\mathcal{A}$; obtain initial state $s$
2: $\mathcal{G}_F \leftarrow \emptyset, \mathcal{U} \leftarrow \emptyset, \hat{\mathcal{T}} \leftarrow \emptyset$
3: **while** elapsed time $< B$ **do**
4: $\quad u(s) \leftarrow \textsc{Extract}(s)$ ▷ Extract the widgets
5: $\quad$ **if** $\textsc{UnseenWidgets}(u(s), \mathcal{U})$ **then**
6: $\quad\quad F(s) \leftarrow \textsc{Inferhypothesis}(s)$ ▷ Infer the hypotheses
7: $\quad\quad \mathcal{G}_F \leftarrow \mathcal{G}_F \cup F(s)$ ▷ Add hypotheses to global pool
8: $\quad\quad \mathcal{U} \leftarrow \mathcal{U} \cup u(s)$
9: $\quad$ **end if**
10: $\quad$ **if** $\textsc{HasUnexplored}(\mathcal{G}_F(u(s)))$ **then**
11: $\quad\quad f \leftarrow \textsc{Select}(\mathcal{G}_F(u(s)))$
12: $\quad\quad (\tau_f, s', \mathcal{G}_F, \mathcal{U}) \leftarrow \textsc{Execute}(f, s, \mathcal{G}_F, \mathcal{U})$
13: $\quad\quad \hat{\mathcal{T}} \leftarrow \hat{\mathcal{T}} \cup \{\textsc{Summarize}(\tau_f)\}$ ▷ Summarize evidence
14: $\quad\quad \textsc{MarkExplored}(f, \mathcal{G}_F)$
15: $\quad\quad s \leftarrow s'$
16: $\quad$ **else**
17: $\quad\quad s \leftarrow \textsc{RandomExplore}(s)$
18: $\quad$ **end if**
19: **end while**
20: **return** $\hat{\mathcal{T}}$

## 3.1 Hypothesis-Guided Functionality Exploration

The goal of this stage is to construct execution-grounded behavioral evidence for downstream property synthesis. Instead of relying on external specifications, PropGen systematically explores executable app functionalities and records their runtime interaction traces as behavioral evidence. Algorithm 1 summarizes the workflow.

Given a target app $\mathcal{A}$ and a time budget $B$, PropGen launches the app, initializes the global functionality pool $\mathcal{G}_F$, the set of observed UI contexts $\mathcal{U}$, and the behavioral evidence set $\hat{\mathcal{T}}$ (Lines 1–2), and then iteratively explores the app until the budget is exhausted (Lines 3–19). In each iteration, it extracts the current GUI context (Line 4); if the context contains previously unseen widgets, it invokes a Multimodal Large Language Model (MLLM) to infer functionality hypotheses for the current state and adds them to $\mathcal{G}_F$ (Lines 5–8). It then executes an unexplored hypothesis together with its triggering widget, and summarizes the resulting trace into a behavioral evidence item added to $\hat{\mathcal{T}}$ (Lines 10–15). When no unexplored hypothesis remains in the current context, PropGen performs lightweight random exploration to leave the current local GUI region and expose new interaction opportunities (Lines 16–17).

*3.1.1 Functionality Hypothesis Generation.* Given a GUI state $s$ encountered during exploration, this step aims to infer which app functionalities may be executable under the current interface context. To support subsequent execution, each inferred functionality is associated with a concrete *triggering widget* in the current state, *i.e.*, the widget from which the functionality is initiated (*e.g.*, a "+" button initiates "create a note").

**Semantic context construction.** To support functionality inference, PropGen first constructs a semantic context for the current GUI state $s$. This context includes three parts: the screen-level context of the current state, app-level semantic cues, and cross-state functionality memory. The screen-level context is derived from the current GUI screenshot. To obtain explicit widget references, PropGen retrieves the accessibility tree of $s$ via uiautomator2 [59], keeps the leaf-level interactive widgets, and draws a sequential number at each widget's top-right corner using its `bounds` attribute. These numbers let the MLLM name a widget rather than describe it in prose, grounding the inferred functionalities in concrete widgets. They are recomputed per screen and carry no meaning across screens; cross-screen identity is instead established by the widget signature described below. The app-level semantic cues include the app name and the list of activity names extracted from the `AndroidManifest.xml` file, helping the MLLM interpret the current screen under the broader semantic context of $\mathcal{A}$. Finally, the cross-state functionality memory stores previously inferred functionalities, helping avoid repeatedly rediscovering semantically similar functionalities in later GUI states. Together, these components provide the contextual information needed for inferring plausible user-facing functionalities from $s$.

**Functionality hypothesis inference.** Using the constructed semantic context, PropGen invokes an MLLM to infer candidate functionality hypotheses for the current GUI state $s$. Formally, the inferred hypotheses are represented as $F(s) = [\langle f_1, w_1\rangle, \langle f_2, w_2\rangle, \ldots, \langle f_k, w_k\rangle]$, where each functionality hypothesis $f_i$ is a concise natural-language description of a candidate app functionality, and $w_i$ denotes its corresponding triggering widget. This widget grounding serves two purposes. First, it constrains the MLLM to infer functionalities that are supported by the current GUI state, reducing unsupported or non-actionable predictions. Second, it makes the inferred functionalities directly executable in subsequent exploration.

The inferred functionality hypotheses, together with their associated triggering widgets, are then stored in the global functionality pool $\mathcal{G}_F$ to support later execution and cross-state reuse.

**Inference triggering and hypothesis reuse.** To avoid redundant MLLM invocations on similar GUI states, PropGen triggers functionality inference only when a newly visited state introduces unseen widgets. It maintains a global set of explored UI contexts $\mathcal{U} = \{u_1, u_2, \ldots\}$, where each context is defined as $u(s) = \langle a(s), W(s)\rangle$, with $a(s)$ denoting the activity and $W(s)$ the set of signatures of leaf-level interactive widgets.

For each state $s$, PropGen compares $W(s)$ with existing contexts under the same activity. If no new widget signatures are observed, it reuses previously inferred functionality hypotheses; otherwise, it invokes the MLLM to infer new functionalities and updates the global pool $\mathcal{G}_F$. To enable stable comparison across screens, each widget $w$ is represented by a signature of attributes $\langle class, resourceId, text, description\rangle$, which are commonly used to identify unique widgets in practice [50, 52, 67]. To reduce noise from dynamic content, PropGen retains only app-defined text in signatures by filtering widget text against a whitelist extracted via static analysis. This avoids treating semantically identical screens with transient text differences as distinct contexts.

*3.1.2 Functionality Hypothesis Validation.* Given the global functionality pool $\mathcal{G}_F$ generated in § 3.1.1, PropGen executes app functionalities in a targeted manner rather than through arbitrary GUI events: it treats each hypothesis $\langle f, w\rangle$ in $\mathcal{G}_F$ as an explicit execution target and prioritizes previously unexplored ones. Executing $\langle f, w\rangle$ induces state transitions $s \xrightarrow{E} s'$, which both collect behavioral evidence for the executed functionality and drive the app into new states $s'$, where further hypotheses $F(s')$ are inferred and added to $\mathcal{G}_F$. Execution and hypothesis generation thus form a closed exploration loop.

**Functionality selection.** At each GUI state $s$, PropGen retrieves from $\mathcal{G}_F$ the candidate functionality hypotheses associated with the current UI context $u(s)$, denoted as $\mathcal{G}_F(u(s))$. Since the exploration budget is limited, PropGen does not execute these candidates arbitrarily, but prioritizes those with higher expected exploration utility. This prioritization is implemented using lightweight heuristics guided by three considerations: whether the functionality corresponds to a main app behavior, whether it is semantically different from previously executed functionalities, and whether it is likely to be successfully executed in the current context. These considerations are operationalized by prompting the MLLM to rank the candidates accordingly. Based on these heuristics, the top-ranked unexplored functionality hypothesis $\langle f, w\rangle$ is selected as the next execution target. The selected hypothesis is then passed to a goal-directed interaction loop that incrementally plans, executes, and evaluates UI actions until the functionality is completed or a step limit is reached.

**Event planning and execution.** Once a functionality hypothesis $\langle f, w\rangle$ is selected, PropGen executes it through a goal-directed interaction loop consisting of event planning, guarded execution, and post-event evaluation. At each step $i$, PropGen first predicts the next GUI event based on three sources of information: the functionality goal $f$, the execution history $H_i$ of prior events and their outcomes, and the current GUI state $s_i$, represented as a screenshot with labeled widgets. The predicted event is denoted as $e_i = \langle t_i, w_i, v_i\rangle$, where the event type $t_i$ is chosen from a predefined event space including `click`, `long-click`, `edit`, `swipe`, and `back`, the target widget $w_i$ is specified by its numeric label, and $v_i$ specifies the attached data. Before executing $e_i$, PropGen verifies that the referenced widget $w_i$ is present in the current state $s_i$. If the predicted widget identifier is invalid, the event is skipped and directly labeled as a failed step. This guarded execution mechanism prevents invalid model-predicted interactions and improves execution robustness.

After execution produces a state transition $s_i \xrightarrow{e_i} s_{i+1}$, PropGen evaluates whether the step has advanced the selected functionality. This evaluation jointly considers the pre- and post-event GUI states, the functionality hypothesis $f$, and the accumulated execution history $H_i$. Based on this evidence, the MLLM assigns an outcome label $o_i \in \{success, fail, complete\}$ to the current step. Here, *success* indicates meaningful progress toward the functionality goal, *fail* indicates that the event was unproductive for the intended functionality, and *complete* indicates that the functionality goal has been achieved. The resulting step and outcome are then incorporated into the execution history for subsequent planning, while *complete* terminates execution of the current functionality.

**Behavioral evidence summarization.** To support downstream property synthesis, PropGen summarizes each executed interaction trace into compact behavioral evidence, since raw traces are low-level, noisy, and specific to a single execution. It incrementally converts each interaction step $s_i \xrightarrow{e_i} s_{i+1}$ into a structured transition with five elements: the state summary before the event, an event summary describing the interaction performed, the state summary after the event, a state-diff summary capturing the visible difference between the two GUI states, and the outcome label $o_i$.

The state summaries are functionality-oriented: they describe the screen context, visible actionable elements, current content state, and observable feedback cues, whereas the state-diff summary focuses on the concrete GUI changes induced by the interaction. The resulting summarized trace, denoted as $\hat{\tau}_f$, serves as the behavioral evidence used in the subsequent property synthesis stage.

**Exploration beyond local hypothesis exhaustion.** PropGen may reach a GUI state whose functionality hypotheses have all been executed. The only remaining task is to move the app into a new GUI region, for which querying the MLLM would cost tokens and time without yielding new behavioral evidence. PropGen therefore falls back to lightweight random exploration, issuing random events until it reaches a state that either has an unexecuted hypothesis, which it executes directly, or unseen widgets, for which it first infers new hypotheses.

## 3.2 Property Synthesis from Behavioral Evidence

The goal of this stage is to derive executable properties from the behavioral evidence collected during functionality exploration. As input, PropGen takes the summarized trace $\hat{\tau}_f$ produced in Section 3.1, which compactly captures how a functionality is exercised and what observable GUI outcome it induces. Each synthesized property takes the form $\phi = \langle P, I, Q\rangle$, where $P$ is a precondition, $I$ is an interaction scenario, and $Q$ is a postcondition assertion.

Rather than generating $\phi$ directly from $\hat{\tau}_f$, PropGen decomposes property synthesis into two steps, separating semantic property formulation from executable code generation. It first constructs a natural-language property specification that explicitly captures the intended $\langle P, I, Q\rangle$ relation, and then translates this intermediate specification into executable property code.

**Natural-language property description generation.** Based on the summarized trace $\hat{\tau}_f$, PropGen first prompts the MLLM to construct a natural-language property specification $\phi^{NL} = \langle P, I, Q\rangle$ for the functionality captured by the behavioral evidence. The summarized trace provides structured evidence about the execution context, performed interactions, observable state changes, and step outcomes. Rather than merely restating these observations, the MLLM is guided to abstract from $\hat{\tau}_f$ a generalized behavioral rule that should hold across executions of the same functionality.

The inferred specification $\phi^{NL}$ consists of three parts: a *precondition* $P$ describing the observable GUI context in which the property should be checked, an *interaction scenario* $I$ capturing the user events needed to exercise the functionality, and a *postcondition* $Q$ specifying the immediate visible effect that should hold afterward. To improve precision and executability, PropGen constrains this inference process in three ways. First, $P$ must be grounded in observable UI evidence and sufficiently specific to avoid triggering the property on unrelated screens with superficially similar widgets. Second, $Q$ must focus on effects that are directly and reliably verifiable from the GUI, such as the appearance, disappearance, or modification of visible widgets or content. Third, the inferred specification should avoid trace-specific brittle details, such as incidental text instances or unstable widget states, and instead capture functionality semantics in a form that remains robust across executions. These three constraints are enforced as explicit instructions in the synthesis prompt.

**Executable property translation.** Given the inferred natural-language property specification $\phi^{NL} = \langle P, I, Q\rangle$, PropGen translates it into executable property code for the target PBT framework. Specifically, this step realizes the precondition $P$, interaction scenario $I$, and postcondition $Q$ using the framework's property structure and API conventions, thereby producing a runnable property implementation. Since the intended property semantics have already been made explicit in the preceding step, this translation primarily serves to operationalize the structured specification rather than to perform further behavioral inference.

This step is implemented by prompting the LLM with the generated natural-language specification together with framework-specific APIs and widget attributes, with reference to the prompt design in prior work on translating natural-language properties into executable ones [70]. The resulting code is then passed to the subsequent validation and refinement stage.

## 3.3 Feedback-Driven Property Refinement

Properties automatically generated from functionality traces may still be imprecise and therefore trigger false positives during testing. However, a reported failure should not be refined by simply adapting the property to that single execution, because such a fix may overfit the observed case and drift away from the original functionality intent.

Given a property whose reported violations are manually confirmed to be false positives, our key idea is that refining it requires first recovering the property's original testing intent, and then revising it so that it remains valid for both the original intended execution and the newly observed legitimate execution. To this end, we ground refinement in the source summarized trace from which the property was originally inferred, rather than treating refinement as unconstrained rewriting.

Specifically, PropGen reasons about the refinement using both the source summarized trace and the failure-triggering execution. It first recovers the original testing intent of the property by locating the relevant segment in $\hat{\tau}_f$ that matches the property's triggering context, interaction scenario, and expected outcome. It then compares this trace-grounded intended behavior with the failing execution to explain why the violation occurs and identify which component of $\phi$ has become imprecise, *i.e.*, the precondition $P$, the interaction scenario $I$, or the postcondition $Q$.

Based on this diagnosis, PropGen refines only the faulty component through a minimal modification. More specifically, it strengthens $P$ when additional UI guards are needed, revises $I$ when the event sequence does not faithfully reflect the intended behavior, and relaxes or simplifies $Q$ when the assertion is overly specific. In this way, the refined property remains consistent with both the source-trace execution and the newly observed legitimate execution, while preserving the original testing intent as much as possible.

# 4 Implementation

We implemented PropGen as an end-to-end prototype for automated property generation on Android apps. PropGen is built on top of DroidBot [28], an open-source Android app exploration tool. It is primarily written in Python and JavaScript. Specifically, PropGen uses uiautomator2 [59] to retrieve GUI layouts and Android Debug Bridge (ADB) [56] to capture screenshots and send GUI events, including `click`, `long-click`, `edit`, `swipe`, and `back`. The generated properties are translated into the executable format expected by Kea.

# 5 Evaluation

We aim to answer the following research questions:

- **RQ1**: How effective is PropGen in exploring app functionalities and generating valid properties?
- **RQ2**: To what extent do generated properties suffer from imprecision that leads to false positives, and how effective is our refinement technique in refining them?
- **RQ3**: Can the generated properties help find new functional bugs in real-world mobile apps?
- **RQ4**: How does PropGen compare with prior functional testing techniques in uncovering new functional bugs?

## 5.1 Setup and Method

**App subjects.** We selected 12 popular and representative open-source Android apps as experimental subjects. Among them, eight apps were adopted from prior studies on functional bug detection for Android apps [51, 55, 63, 71]. From the apps used in these studies, we excluded those that were either (1) no longer runnable or actively maintained, or (2) highly similar in functionality to apps already

**Table 1: App subjects used in our experiment (K=1,000, M=1,000,000).**

| App Name | App Feature | #Downloads | #Stars | LOC |
|---|---|---|---|---|
| *OmniNotes* | Note Manager | 100~500K | 2.8K | 57,529 |
| *Markor* | Text Editor | - | 5.3K | 79,749 |
| *RetroMusic* | Audio Player | 1~5M | 5K | 110,065 |
| *Amaze* | File Manager | 1~5M | 6.1K | 159,040 |
| *MyExpenses* | Financial Assistant | 1~5M | 1.1K | 317,899 |
| *AntennaPod* | Podcast Manager | 1~5M | 7.7K | 130,925 |
| *AnkiDroid* | Flashcards Manager | 10~50M | 10.9K | 403,785 |
| *OuterTune* | YouTube Music Player | - | 4.8K | 89,673 |
| *NewPipe* | Video Player | - | 37.5K | 187,187 |
| *MaterialFiles* | Storage Browser | 1~5M | 8K | 95,705 |
| *Orgzly* | To-do Lists Manager | 100~500K | 2.8K | 72,042 |
| *uhabits* | Habit Tracker | 5~10M | 9.7K | 69,652 |

selected. To further improve diversity, we additionally included four popular open-source apps from Google Play with different features. Table 1 summarizes the selected apps. In the table, *App Feature* denotes the primary functionality of each app, *#Downloads* and *#Stars* report the number of Google Play installations and GitHub stars, respectively, and *LOC* gives the lines of code.

**Experimental environment.** All experiments were conducted on a machine running Ubuntu 22.04 with 192 CPU cores (AMD EPYC 9654) and official Android emulators (Android 11, Pixel). We use GPT-5.2 (snapshot `gpt-5.2-2025-12-11`) as the backend MLLM for PropGen and the baseline tool, with the default decoding parameters (temperature = 1.0, top-$p$ = 1.0). For each app, we allocated 3 hours for behavioral evidence construction and property synthesis. After that, we used Kea to perform property-based testing with the generated properties for 6 hours per app, following the testing budget adopted in Kea's paper [71].

**Baselines.** We use six baselines for different research questions, according to evaluation goals. For RQ1, we compare PropGen with DroidAgent [75], a representative LLM-based mobile app functionality exploration approach, which can automatically identify and execute functionalities. To assess whether the multi-stage pipeline of PropGen is necessary at all, we additionally compare against a *naive LLM* baseline that asks the LLM to generate properties directly, without any runtime exploration. Concretely, we prompt the same MLLM with the app's official description and ask it to produce 20 properties per app, using the same property format as PropGen. For RQ4, which evaluates bug-finding capability, we compare PropGen with four representative prior techniques for Android functional bug detection: Genie [51], Odin [63], PBFDroid [52], and VisionDroid [34]. Among them, Genie and Odin rely on designed automated oracles, PBFDroid is a property-based testing technique for data manipulation functionalities (DMFs), and VisionDroid is an LLM-based multi-agent approach for functional bug detection.

**Evaluation method of RQ1.** RQ1 aims to evaluate whether PropGen can correctly infer and execute app functionalities and synthesize valid properties from the collected behavioral evidence. This evaluation cannot be performed fully automatically, because mobile apps typically lack precise functional specifications. Therefore, we manually assess the validity of both the inferred functionalities and the generated property descriptions, following prior work on functionality-level evaluation beyond structural coverage metrics [7, 75]. The manual evaluation is conducted by two graduate students in software engineering, each with at least four years of Android development experience. Neither annotator is a co-author or was involved in developing PropGen, which reduces assessment bias. Before the annotation, each annotator was given at least fifteen minutes to familiarize themselves with the functionalities of every subject app. During this process, annotators also referred to the app's official introduction page when available, so that the subsequent judgments were made with sufficient understanding of the app's functionalities. In addition, annotators could interact with the running app at any time to verify uncertain cases.

For each inferred functionality, we evaluate two aspects. The first is *functionality validity*, which examines whether the inferred functionality actually exists in the app. Annotators are given the inferred functionality description together with the GUI screenshot from which it was inferred, and determine whether the described functionality is genuinely supported by the interface. The second is *execution correctness*, which examines whether the system correctly executes the inferred functionality. For each inferred functionality, annotators are provided with the corresponding execution screenshots and interaction events, and judge whether the executed interaction sequence indeed realizes the intended functionality.

Because mobile apps lack ground-truth specifications, we cannot decide a property's validity against a reference oracle. We therefore assess validity along two complementary layers, one checked by an automated proxy and the other by human annotators. The *groundedness* layer is verified by an *automated proxy*: we automatically check that every widget and event referenced in the property is observable in its source trace, so that the property is executable on the app rather than referring to hallucinated UI elements. The *faithfulness* layer is judged by *human annotators*, who assess the three components: (1) whether the *precondition* appropriately constrains the UI state in which the property should be applied, (2) whether the *interaction scenario* accurately reflects the user interactions required to perform the functionality, and (3) whether the *postcondition* correctly captures the observable UI behavior that should hold immediately after the scenario. A property description is considered *valid* only if it is grounded in its source trace *and* all three components are faithful to the intended functionality; otherwise, it is labeled as *invalid*. Validity does not imply precision: whether a property avoids false positives under arbitrary executions is a separate question, examined in RQ2.

To reduce bias, the two annotators first labeled all results fully independently, without discussing their judgments with each other or with the authors. We measured inter-rater agreement on these independent annotations using Cohen's $\kappa$, obtaining 0.91, 0.82, and 0.81 for functionality validity, execution correctness, and property validity, respectively, which indicates high agreement across all three aspects. Only after the independent annotation was complete did the two annotators discuss the remaining disagreements between themselves and reach a consensus label for each disagreed item. Based on these final annotations, we report the proportions of inferred functionalities that are valid, inferred functionalities that are correctly executed, and generated valid property descriptions.

**Evaluation method of RQ2 and RQ3.** RQ2 evaluates two aspects of property refinement: (1) how many generated properties are

imprecise and thus lead to spurious violations during testing, and (2) how many of these imprecise properties can be successfully refined through refinement. RQ3 evaluates whether the generated properties can find new functional bugs. Specifically, for each app, we load all initially generated executable properties into Kea [71] for testing with 6 hours. Whenever Kea reports a property violation, we manually inspect the property description, executable property code, violation-triggering execution trace, corresponding screenshots and interaction events, and the assertion outcome. Based on this, we determine whether the violation is caused by a real app bug or by property imprecision.

For RQ2, we focus on the properties whose reported violations are diagnosed as spurious. We apply refinement only to those caused by property imprecision. For each such property, the refinement module takes the original property description, executable property code, and violation-triggering execution evidence as input, and produces a revised executable property. We then re-execute the refined property on two traces and inspect the results: the source trace from which the property was originally derived, and the trace that triggered the spurious violation. We call the revised property *plausibly correct* if it passes two automated checks: (a) it still holds on the source trace, and (b) it no longer reports the previously observed spurious violation on the violation-triggering trace. These two checks form an *automated proxy*: they are necessary but not sufficient, since a revision can pass both while still failing to faithfully specify the intended functionality. We therefore reserve the term *correct* for a plausibly correct property that human annotators further confirm to be *valid* (*i.e.*, grounded and faithful, as defined in RQ1) and to *preserve the original testing intent*, by inspecting the full $\langle P, I, Q \rangle$ triple. A refinement is considered successful only when the revised property is both plausibly correct and human-confirmed correct. Based on this process, we report the number of properties sent to refinement due to property imprecision, the number and rate of successful refinements, and the breakdown of successfully refined properties by the modified component (*i.e.*, precondition, interaction, and postcondition).

For RQ3, we focus on the violations diagnosed as real app bugs through manual inspection. For each confirmed bug, we prepare a bug report containing the bug description, reproduction steps, expected and actual behaviors, and submit it to the corresponding app developers.

**Evaluation method of RQ4.** RQ4 compares PropGen with existing functional testing techniques in uncovering new functional bugs. Non-crash functional bug detection is *oracle-dependent*: each technique can only expose the bugs that its oracle is able to express. Raw bug counts alone are therefore not directly comparable across these tools, as a tool that reports fewer bugs is not necessarily less capable, but may target a narrower class of bugs by design. To draw a meaningful comparison, we report absolute bug counts for completeness *and* complement them with a scope analysis, which separates the two distinct reasons a tool can miss a bug: its oracle cannot express the bug at all, or its oracle can express the bug but the tool failed to expose it within the testing budget. Reporting both lets us measure overall bug-finding effectiveness while also explaining where the differences come from.

**Table 2: Validity of functionalities and synthesized properties across subject apps (RQ1).**

| App Name | #Inferred Func | | #Valid Func | | #Correctly Executed | | #Prop | #Valid Prop |
|---|---|---|---|---|---|---|---|---|
| | D | P | D | P | D | P | | |
| *OmniNotes* | 53 | 108 | 53 (100.0%) | 102 (94.4%) | 18 (34.0%) | 80 (74.1%) | 81 | 80 (98.8%) |
| *Markor* | 48 | 117 | 36 (75.0%) | 116 (99.1%) | 12 (25.0%) | 85 (72.6%) | 88 | 83 (94.3%) |
| *RetroMusic* | 48 | 113 | 42 (87.5%) | 104 (92.0%) | 9 (18.8%) | 88 (77.9%) | 89 | 70 (78.7%) |
| *Amaze* | 37 | 84 | 24 (64.9%) | 83 (98.8%) | 8 (21.6%) | 64 (76.2%) | 57 | 52 (91.2%) |
| *MyExpenses* | 54 | 106 | 54 (100.0%) | 104 (98.1%) | 21 (38.9%) | 82 (77.4%) | 81 | 78 (96.3%) |
| *AntennaPod* | 47 | 142 | 38 (80.9%) | 139 (97.9%) | 28 (59.6%) | 116 (81.7%) | 120 | 111 (92.5%) |
| *AnkiDroid* | 50 | 116 | 28 (56.0%) | 107 (92.2%) | 9 (18.0%) | 93 (80.2%) | 91 | 85 (93.4%) |
| *OuterTune* | 41 | 76 | 41 (100.0%) | 67 (88.2%) | 9 (22.0%) | 55 (72.4%) | 48 | 44 (91.7%) |
| *NewPipe* | 46 | 106 | 45 (97.8%) | 92 (86.8%) | 20 (43.5%) | 81 (76.4%) | 79 | 72 (91.1%) |
| *MaterialFiles* | 47 | 110 | 44 (93.6%) | 103 (93.6%) | 23 (48.9%) | 82 (74.5%) | 86 | 82 (95.3%) |
| *Orgzly* | 49 | 124 | 48 (98.0%) | 115 (92.7%) | 14 (28.6%) | 88 (71.0%) | 96 | 91 (94.8%) |
| *uhabits* | 55 | 80 | 38 (69.1%) | 78 (97.5%) | 16 (29.1%) | 63 (78.8%) | 69 | 64 (92.8%) |
| **Total** | **575** | **1282** | **491 (85.4%)** | **1210 (94.4%)** | **187 (32.5%)** | **977 (76.2%)** | **985** | **912 (92.6%)** |

Concretely, following prior comparative analysis practice [52, 71], we evaluate these tools from two perspectives. First, we measure the *overall bug-finding results* of each tool empirically: we run each prior technique on the same 12 apps and count all previously unknown functional bugs it uncovers, using the same manual-confirmation criterion as for PropGen, *i.e.*, each reported violation is inspected and counted only if it is confirmed to be a real, previously unreported functional bug. For Genie, Odin, and VisionDroid, we follow the default configurations described in their original papers. PBFDroid requires users to manually specify properties for data manipulation functionalities (DMFs); therefore, we manually defined the required DMF properties for detecting the corresponding bugs. To ensure fairness, we align the time budget with each tool's workflow: PropGen uses 3 hours for property generation and 6 hours for bug finding; accordingly, we allocate 9 hours per app to Genie, Odin, and VisionDroid, and 6 hours of automated testing to PBFDroid after manual property construction. This single run yields both each tool's total bug count, independent of the bug set found by PropGen, and, by intersecting with PropGen's 25 new bugs, how many of them each tool rediscovers in practice.

Second, we conduct a scope analysis to determine whether each of the 25 bugs uncovered by PropGen theoretically falls within the detection scope of each prior technique. This analysis is performed manually based on bug characteristics and the detection capabilities claimed by each technique. To improve reliability, we further consulted the authors of Genie, Odin, and PBFDroid to validate our analysis. For VisionDroid, we do not perform a separate scope analysis, since its LLM-based design makes its theoretical detection scope difficult to characterize precisely; instead, we rely only on its empirical bug-finding performance.

## 5.2 Results of RQ1

Table 2 reports the results of functionality inference, execution, and property synthesis on all subject apps, where *#Inferred Func* denotes the number of functionalities inferred by tools, *#Valid Func*, *#Correctly Executed* and *#Valid Prop* denote the numbers of functionalities and properties validated by human annotators, and *#Prop* denotes the number of generated properties. For functionality-related evaluation, we compare DroidAgent (D) and our approach (P). Overall, our approach consistently outperforms DroidAgent in both

functionality inference and execution. Across the 12 apps, our approach infers 1,282 functionalities, of which 1,210 are judged valid, achieving 94.4% functionality validity, compared with 575 inferred functionalities and 491 valid ones (85.4%) for DroidAgent. It also correctly executes 977 functionalities, yielding 76.2% execution correctness, substantially higher than DroidAgent's 187 correctly executed functionalities and 32.5% execution correctness. On average, this corresponds to 101 valid functionalities and 81 correctly executed functionalities per app for our approach, compared with 41 and 16, respectively, for DroidAgent.

One likely reason for the performance gap is the difference in functionality inference. DroidAgent infers functionalities without explicitly grounding them to concrete GUI widgets, making some inferred results loosely related to the current interface context and thus more likely to be invalid. In contrast, our approach grounds functionality inference in the current GUI context, which helps produce more valid functionality hypotheses and makes subsequent execution more reliable. Also, we find DroidAgent tends to repeatedly execute failed events during functionality execution.

For property synthesis, our approach generates 985 property descriptions, among which 912 are judged valid, corresponding to 92.6% property validity. Moreover, property validity exceeds 90% on most apps, indicating that the synthesized properties are generally well aligned with observed app behaviors. In contrast, the naive LLM baseline, which is given only the app description, produces 240 properties across the 12 apps, *none* of which is valid: without observing the app at runtime, the LLM hallucinates functionalities and widgets that do not exist in the app, so the resulting properties reference non-existent UI elements and are not executable. The LLM alone is thus insufficient without runtime behavioral evidence.

**Failure analysis.** *Functionality validity.* The MLLM occasionally proposes a functionality that does not exist in the app, likely because the semantic context for inference (§ 3.1.1) is sometimes insufficient. An *interact-then-infer* strategy, which first probes a widget and then infers the functionality from the observed effect, could help. *Execution correctness.* Failed executions either do not complete the functionality within the step budget, or are prematurely labeled as successful. The former could be mitigated by an activity transition graph derived via static analysis to guide step planning, the latter by an explicit completion check grounded in post-state evidence rather than the MLLM's own judgment. *Property validity.* The LLM occasionally hallucinates widgets absent from the GUI when synthesizing $P$, $I$, or $Q$, despite the grounding requirement (§ 3.2). A post-synthesis check that verifies every referenced widget against the source trace could reject or regenerate such properties.

**LLM usage cost.** For *behavioral evidence construction* and *property synthesis*, each LLM call consumes 6,198 tokens / $0.0145 on average. Overall, our approach uses 6,364k tokens / $14.86 per app on average. In comparison, DroidAgent uses 5,109k tokens / $13.01 per app.

## 5.3 Results of RQ2

Table 3 reports the effectiveness of property refinement across all subject apps, including how many generated properties produce false positives during property-based testing, how many are successfully refined, and which property components are modified.

**Table 3: Effectiveness of property refinement across subject apps (RQ2).**

| App Name | #Imprecise Prop | #Successful Refinements | Modified Component | | |
|---|---|---|---|---|---|
| | | | Pre | I | Post |
| *OmniNotes* | 12 | 11 (91.7%) | 5 | 0 | 6 |
| *Markor* | 15 | 12 (80.0%) | 2 | 1 | 9 |
| *RetroMusic* | 15 | 14 (93.3%) | 9 | 0 | 5 |
| *Amaze* | 11 | 10 (90.9%) | 2 | 0 | 8 |
| *MyExpenses* | 8 | 8 (100.0%) | 6 | 0 | 2 |
| *AntennaPod* | 20 | 18 (90.0%) | 7 | 0 | 11 |
| *AnkiDroid* | 9 | 9 (100.0%) | 7 | 0 | 2 |
| *OuterTune* | 3 | 3 (100.0%) | 3 | 0 | 0 |
| *NewPipe* | 11 | 10 (90.9%) | 2 | 0 | 8 |
| *MaterialFiles* | 6 | 6 (100.0%) | 2 | 0 | 4 |
| *Orgzly* | 12 | 12 (100.0%) | 5 | 0 | 7 |
| *uhabits* | 5 | 5 (100.0%) | 3 | 0 | 2 |
| **Total** | **127** | **118 (92.9%)** | **53** | **1** | **64** |

Among all generated executable properties, 127 produce spurious violations during testing and are thus identified as imprecise properties that lead to false positives. This result shows that property imprecision is not uncommon in LLM-generated properties, and therefore refinement is necessary to improve their practical usability. Overall, our refinement technique is highly effective. Across all apps, 127 properties are sent to refinement, and 118 of them are successfully refined, yielding an overall refinement rate of 92.9%. Moreover, the refinement performance is consistently strong across apps: six apps achieve a 100% refinement rate, while the remaining apps still achieve rates at least 80%.

We further analyze which property components are modified during refinement. Among the 118 successfully refined properties, 53 involve precondition modifications, 64 involve postcondition modifications, and only 1 involves an interaction modification. This suggests that most false positives can be resolved by refining when a property should be triggered or what outcome it should assert, rather than changing the core interaction sequence. Among the 127 imprecise properties, 9 remain not refined. We find that 5 of them are caused by incorrect diagnosis of spurious violations, while the others occur because the revised property can no longer remain consistent with the original functionality. More specifically, in the 5 misdiagnosed cases the refinement attributes the false positive to the wrong component among $P$, $I$, and $Q$, and therefore revises a component that is not the actual cause; in the remaining 4 cases the revision drifts away from the original testing intent. These two patterns suggest two directions for improvement: providing richer context (*e.g.*, more detailed traces or intermediate GUI states) to help localize the true cause of a false positive, and adding a post-refinement intent-verification step (*e.g.*, an LLM judge against the source trace) that rejects or regenerates revisions that drift from the original intent. Note that for *property refinement*, each property requires 8,768 tokens / $0.02 on average.

## 5.4 Results of RQ3

Table 4 summarizes the bug-finding results, including the app name, bug ID, and the brief description of the violated property. Overall, PropGen uncovered 25 unique previously unknown functional bugs in the latest app versions. Currently, 5 of the reported bugs

**Table 4: Statistics of the 25 new functional bugs found by the generated properties.**

| App Name | ID | Violated Property |
|---|---|---|
| *OmniNotes* | 1 | Note info dialog should contain statistical data |
| | 2 | The date should appear on the selection page |
| | 3 | The category selection should be changeable |
| | 4 | The category page should be accessible from the drawer |
| | 5 | The captured photo should appear in the note content |
| | 6 | Returning from the sketch should display the note title |
| | 7 | The reminder icon should be displayed after setting a reminder |
| | 8 | The attachment should appear after selection |
| | 9 | The image can be opened in the note content |
| | 10 | The FAB should appear after return |
| *RetroMusic* | 11 | The specific item should disappear from the list |
| | 12 | The lyrics should be displayed after saving |
| | 13 | The item should not exist in the list |
| | 14 | The image should be displayed after change |
| | 15 | The artist can be reset to default |
| | 16 | The metadata should remain consistent |
| *Amaze* | 17 | The item should be deleted successfully |
| *OuterTune* | 18 | The song list should open from navigation menu |
| *MaterialFiles* | 19 | The file can be successfully created |
| | 20 | The item can be found |
| | 21 | Navigation to the sub-directory should succeed |
| *uhabits* | 22 | The added number should keep consistent |
| *Orgzly* | 23 | The imported file should be present |
| *Markor* | 24 | Star and "Favourite" checkbox should stay in sync |
| | 25 | The insert image should be displayed after preview |

have been fixed by developers, while the remaining reports are waiting for responses.

The bugs span four categories: (i) *functionality does not take effect* (12 bugs, *e.g.*, a captured photo is not attached to the note, or a deleted item remains in the list), (ii) *state inconsistencies* (6 bugs, *e.g.*, the reminder icon is not shown after a reminder is set, or the lyrics of a song are not updated after saving), (iii) *unexpected wrong behaviors* (4 bugs, *e.g.*, the floating action button fails to reappear after returning to the note list, or an info dialog does not contain the expected statistical data), and (iv) *navigation failures* (3 bugs, *e.g.*, a page is unreachable from the drawer). The bugs also cover diverse functionalities (*e.g.*, note management, attachment insertion, content display), suggesting that the generated properties are not tied to a single type of expected behavior and can capture a broad range of behavioral constraints. These bugs arise when the actual app behavior deviates from the expected behavior encoded by the properties. For example, Bug 5 in *OmniNotes* is triggered by the property *The captured photo should appear in the note content*. During testing, it generates a GUI event sequence that first opens audio recording and then checks this property. After executing the interaction sequence, the captured photo fails to appear, thus violating the property. This bug is difficult to uncover through manual testing, as testers typically focus on the main interaction path and may not consider interleavings with other events.

## 5.5 Results of RQ4

Table 5 reports how the 25 new bugs uncovered by PropGen relate to the four prior techniques: only 7 (28%) are within their scope, and only 3 (12%) are actually found in practice. Genie, Odin, and PBFDroid rely on predefined automated oracles or manually specified DMF properties, and can therefore only find the bug categories emphasized in their original designs, whereas PropGen targets more broadly. VisionDroid, in contrast, is a more general LLM-based approach, but its exploration validates a functionality by following one plausible interaction path at a time, whereas most bugs uncovered by PropGen are exposed only when a property is checked under specific event sequences.

**Table 5: The 25 new bugs found by PropGen *w.r.t.* prior testing tools: how many fall within each tool's detection scope, and how many it rediscovers in practice. Totals are deduplicated across tools.**

| Tool | #In Scope | #Rediscovered |
|---|---|---|
| Genie | 1 (4.0%) | 0 (0.0%) |
| ODIN | 2 (8.0%) | 0 (0.0%) |
| PBFDroid | 5 (20.0%) | 2 (8.0%) |
| VisionDroid | - | 1 (4.0%) |
| **Total** | **7 (28.0%)** | **3 (12.0%)** |

We further compare the overall bug-finding results. On the same 12 apps, the four prior techniques together uncover 13 previously unknown functional bugs (Genie 3, Odin 3, PBFDroid 6, and VisionDroid 4; the per-tool counts sum to 16 because three bugs are found by more than one tool), fewer than the 25 bugs uncovered by PropGen. Both counts follow the same criterion, *i.e.*, every bug is manually confirmed to be a real, previously unknown functional bug; as reported in RQ3, 5 of those found by PropGen have already been fixed by developers. The 3 rediscovered bugs above are exactly the overlap between the two bug sets: PropGen finds 22 bugs that the prior techniques miss, while they find 10 that PropGen misses. The two are thus complementary rather than substitutable, as both directions stem from the oracles each side can check. PropGen misses those 10 bugs mainly because its generated properties do not cover the oracles these techniques rely on, *e.g.*, the differential oracles of Genie and Odin; it also generates only one property per functionality, and generating more would cover more expected behaviors.

The two bug sets also differ in type. The bugs found by each prior technique are concentrated in the category its oracle targets: all bugs found by PBFDroid are data manipulation errors; Genie and Odin mainly expose unexpected wrong behaviors; and VisionDroid mainly exposes display errors. In contrast, the bugs found by PropGen span four categories reported in RQ3. PropGen therefore achieves broader coverage without manual effort, but does not subsume a specialized oracle within its own niche; combining the two is a promising direction for future work.

# 6 Discussion

**Impact of pre-existing app faults on generated properties.** A pre-existing app fault could in principle corrupt the properties PropGen infers or refines. In the *generation* phase, the risk is limited because PropGen synthesizes properties only from successfully executed traces, so a fault that blocks a functionality yields no property, and it abstracts the observed behavior into a general

property rather than copying a concrete trace. In the *refinement* phase, a correct property could be weakened only if a fault-induced violation is misjudged as spurious; violations attributed to app faults are reported as bugs instead, and refinement yields only a *plausibly correct* property whose final correctness is confirmed by humans.

**Threats to validity.** First, our evaluation involves manual inspection to assess the correctness of inferred functionalities and generated properties, which may introduce subjectivity and potential bias. To mitigate this threat, each case is independently labeled by two experienced graduate students following consistent evaluation criteria; disagreements are further discussed until agreement is reached. Second, the apps used in our evaluation may not fully represent the diversity of real-world mobile apps. To mitigate this threat, most of the apps are selected from prior relevant studies, and we further include four additional apps to improve diversity. In the future, we plan to evaluate PropGen on a larger and broader set of apps. Third, the subject apps are open-source and may appear in GPT-5.2's training data, which raises the concern that PropGen relies on the model's prior knowledge of these apps rather than genuinely inferring the bugs from the collected runtime evidence. To mitigate this threat, we checked each of the 25 bugs against the corresponding app's issue tracker and confirmed that none had been previously reported. Moreover, six of the 12 app builds we tested were released after GPT-5.2's training cutoff, and PropGen still finds bugs in them.

## 7 Related Work

**Automated mobile app GUI testing.** Automated testing for mobile apps has been extensively studied. Choudhary *et al.* [5] conducted a systematic comparison of Android input-generation tools and highlighted both the promise and limitations of automated mobile testing. Prior work has proposed a variety of techniques to automatically explore app GUIs and generate event sequences for detecting crash bugs [9, 19, 33, 35, 37, 50, 61, 64]. For example, Sapienz [37] uses multi-objective search to generate event sequences for improving coverage and exposing crashes. To find non-crash functional bugs, most of prior work [1, 45, 51, 53, 54, 63, 72, 77] designs *automated oracles* to overcome the oracle problem. However, these work are limited to specific types of functional bugs (*e.g.*, data losses [1, 45, 77]). Some work leverages LLM to analyze GUI pages during exploration to find data inconsistency bugs [20], functional bugs [34], or inconsistencies between app design and implementation [31].

Property-based testing (PBT) is a powerful testing methodology and has been adopted into many different software systems to find logic bugs [2, 6, 21, 22, 24, 36, 39, 41, 46]. Recent work has begun to bring property-based testing to mobile apps. Specifically, PBFDroid [52], PDTDroid [55], and Kea [29, 71] have demonstrated that PBT can be effectively applied to GUI-driven mobile apps and can find non-crashing functional bugs that are difficult for traditional automated GUI testing tools to detect. However, these work still assumes that meaningful properties are manually written by developers or testers. Our work complements these approaches by automating the executable property generation.

**Automated test and specification generation.** Traditional automated test generation techniques, such as fuzzing [38], symbolic execution [16, 48, 57], and search-based testing [14, 40], mainly aim to improve code coverage and expose crashes, but they struggle to produce effective assertions for functional correctness [42, 49]. To alleviate this oracle problem, one line of work mines likely specifications from program executions, learning value-based invariants (*e.g.*, Daikon [12]), temporal API rules (*e.g.*, Perracotta [73] and Texada [27]), or behavioral models and typestates (*e.g.*, TzuYu [68] and Synoptic [3]). More recently, learning-based and LLM-based approaches generate unit tests and assertions directly from source code [4, 8, 10, 13, 15, 23, 26, 47, 58, 62, 74, 76], and LLMs have been used to generate specifications or properties in specific domains, *e.g.*, postconditions for functions from natural-language comments [11], property-based tests from docstrings for Python libraries [60], and properties for smart contracts from a curated property library [32].

Like these works, PropGen automatically infers behavioral specifications, but differs from all of them in both setting and methodology. All the above start from a given unit under analysis together with a structured source of intent, either human-written specifications such as docstrings, comments, or a curated property library, or white-box execution data such as method entry/exit values and API call traces. PropGen has neither: it must first *discover* which functionalities to specify by exploring a black-box app, and infers their intended behavior from runtime GUI states, screenshots, and event sequences alone. It further refines the synthesized properties against testing feedback, which these works do not.

Beyond test and specification generation, recent work proposes agents that automatically perform tasks on mobile apps [43, 44, 65–67, 75, 78]. These agents focus on executing user-provided tasks, whereas PropGen centers on automatically exploring app functionalities and generating executable properties for testing.

## 8 Conclusion

In this paper, we present PropGen, an automated approach for constructing properties of mobile apps. By exploring app functionalities and deriving properties from behavioral evidence, PropGen reduces the need for manual property specification. We further propose a feedback-driven refinement technique to refine imprecise properties exposed during testing. Experiments on real-world Android apps show that PropGen can effectively generate valid executable properties. These results demonstrate the practical value of automated property construction for mobile app property-based testing.

## 9 Data Availability

We have made all the artifacts publicly available [69].

## Acknowledgments

We thank the anonymous ASE reviewers for their valuable feedback. This research is supported by the National Research Foundation, Singapore and CyberSG R&D Programme Office under its Translation and Innovation Grant (CRPO-GC4-SMU-002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the National Research Foundation and CyberSG R&D Programme Office. This work was also partially supported by NSFC Project No. 62572192 and 62072178.

## References


[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 83–93. doi:10.1145/2771783.2771786

[2] Thomas Arts, John Hughes, Joakim Johansson, and Ulf Wiger. 2006. Testing telecoms software with Quviq QuickCheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*. 2–10. doi:10.1145/1159789.1159792

[3] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 267–277. doi:10.1145/2025113.2025151

[4] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. Chatunitest: A framework for llm-based test generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 572–576. doi:10.1145/3663529.3663801

[5] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for Android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440. doi:10.1109/ASE.2015.89

[6] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP'00*. 268–279. doi:10.1145/351240.351266

[7] Riccardo Coppola and Emil Alégroth. 2022. A taxonomy of metrics for GUI-based testing research: A systematic literature review. *Information and Software Technology* 152 (2022), 107062. doi:10.1016/j.infsof.2022.107062

[8] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435. doi:10.1145/3597926.3598067

[9] 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*. 481–492. doi:10.1145/3377811.3380402

[10] Kohei Dozono, Tiago Espinha Gasiba, and Andrea Stocco. 2024. Large language models for secure code assessment: A multi-language empirical study. *arXiv preprint arXiv:2408.06428* (2024). doi:10.48550/arXiv.2408.06428

[11] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K Lahiri. 2024. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1889–1912. doi:10.1145/3660791

[12] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. 1999. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st international conference on Software engineering*. 213–224. doi:10.1145/302405.302467

[13] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[14] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419. doi:10.1145/2025113.2025179

[15] Cuiyun Gao, Xing Hu, Shan Gao, Xin Xia, and Zhi Jin. 2025. The current challenges of software engineering in the era of large language models. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–30. doi:10.1145/3712005

[16] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 213–223. doi:10.1145/1065010.1065036

[17] Harrison Goldstein, Joseph W Cutler, Daniel Dickstein, Benjamin C Pierce, and Andrew Head. 2024. Property-based testing in practice. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. doi:10.1145/3597503.3639581

[18] Harrison Goldstein, Joseph W Cutler, Adam Stein, Benjamin C Pierce, and Andrew Head. 2022. Some problems with properties. In *Proc. Workshop on the Human Aspects of Types and Reasoning Assistants (HATRA)*, Vol. 1. 3.

[19] 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)*. IEEE, 269–280. doi:10.1109/ICSE.2019.00042

[20] Yongxiang Hu, Hailiang Jin, Xuan Wang, Jiazhen Gu, Shiyu Guo, Chaoyi Chen, Xin Wang, and Yangfan Zhou. 2024. Autoconsis: Automatic gui-driven data inconsistency detection of mobile apps. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 137–146. doi:10.1145/3639477.3639748

[21] John Hughes. 2016. Experiences with QuickCheck: testing the hard stuff and staying sane. In *A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. Springer, 169–186. doi:10.1007/978-3-319-30936-1_9

[22] John Hughes, Benjamin C Pierce, Thomas Arts, and Ulf Norell. 2016. Mysteries of dropbox: property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 135–145. doi:10.1109/ICST.2016.37

[23] Zongze Jiang, Ming Wen, Jialun Cao, Xuanhua Shi, and Hai Jin. 2024. Towards Understanding the Effectiveness of Large Language Models on Directed Test Input Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1408–1420. doi:10.1145/3691620.3695513

[24] Stefan Karlsson, Adnan Čaušević, and Daniel Sundmark. 2020. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 131–141. doi:10.1109/ICST46399.2020.00023

[25] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10. doi:10.1109/ICST.2015.7102609

[26] Shuvendu K Lahiri, Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, Madanlal Musuvathi, Piali Choudhury, Curtis von Veh, Jeevana Priya Inala, Chenglong Wang, et al. 2022. Interactive code generation via test-driven user-intent formalization. *arXiv preprint arXiv:2208.05950* (2022). doi:10.48550/arXiv.2208.05950

[27] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL specification mining (T). In *2015 30th IEEE/ACM international conference on automated software engineering (ASE)*. IEEE, 81–92. doi:10.1109/ASE.2015.71

[28] 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)*. IEEE, 23–26. doi:10.1109/ICSE-C.2017.8

[29] Xixian Liang, Cheng Peng, Bo Ma, Xiangchen Shen, Yiheng Xiong, and Ting Su. 2026. Kea2: Practical Property-based Testing for Mobile Apps. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*. 182–186. doi:10.1145/3803437.3806416

[30] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test Android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 613–622. doi:10.1109/ICSME.2017.47

[31] Ruofan Liu, Xiwen Teoh, Yun Lin, Guanjie Chen, Ruofei Ren, Denys Poshyvanyk, and Jin Song Dong. 2025. GUIPilot: A Consistency-Based Mobile GUI Testing Approach for Detecting Application-Specific Bugs. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 753–776. doi:10.1145/3728909

[32] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2024. Propertygpt: LLM-driven formal verification of smart contracts through retrieval-augmented property generation. *arXiv preprint arXiv:2405.02580* (2024). doi:10.14722/ndss.2025.241357

[33] Zhe Liu, Chunyang Chen, Junjie Wang, Mengzhuo Chen, Boyu Wu, Xing Che, Dandan Wang, and Qing Wang. 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*. 1–13. doi:10.1145/3597503.3639180

[34] 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). doi:10.1109/TSE.2025.3614469

[35] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234. doi:10.1145/2491411.2491450

[36] David R MacIver, Zac Hatfield-Dodds, et al. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. doi:10.21105/joss.01891

[37] 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

[38] Michał Zalewski. 2016. *American Fuzzy Lop - Whitepaper*. https://lcamtuf.coredump.cx/afl/technical_details.txt

[39] Liam O'Connor and Oskar Wickström. 2022. Quickstrom: property-based acceptance testing with LTL specifications. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 1025–1038. doi:10.1145/3519939.3523728

[40] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816. doi:10.1145/

1297846.1297902

[41] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 398–401. doi:10.1145/3293882.3339002

[42] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J Hellendoorn. 2020. Revisiting test smells in automatically generated tests: limitations, pitfalls, and opportunities. In *2020 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 523–533. doi:10.1109/ICSME46990.2020.00056

[43] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326* (2025). doi:10.48550/arXiv.2501.12326

[44] Dezhi Ran, Hao Wang, Zihe Song, Mengzhou Wu, Yuan Cao, Ying Zhang, Wei Yang, and Tao Xie. 2024. Guardian: A Runtime Framework for LLM-based UI Exploration. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. doi:10.1145/3650212.3680334

[45] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. 2020. Data loss detector: automatically revealing data loss bugs in Android apps. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152. doi:10.1145/3395363.3397379

[46] André Santos, Alcino Cunha, and Nuno Macedo. 2018. Property-based testing for the robot operating system. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. 56–62. doi:10.1145/3278186.3278195

[47] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2023. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering* 50, 1 (2023), 85–105. doi:10.1109/TSE.2023.3334955

[48] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A concolic unit testing engine for C. *ACM SIGSOFT software engineering notes* 30, 5 (2005), 263–272. doi:10.1145/1081706.1081750

[49] Sina Shamshiri. 2015. Automated unit test generation for evolving software. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 1038–1041. doi:10.1145/2786805.2803196

[50] 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 *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 245–256. doi:10.1145/3106237.3106298

[51] 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 (2021), 1–31. doi:10.1145/3485533

[52] Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-Based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1088–1100. doi:10.1145/3611643.3616286

[53] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and finding system setting-related defects in Android apps. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 204–215. doi:10.1145/3460319.3464806

[54] Jingling Sun, Ting Su, Kai Liu, Chao Peng, Zhao Zhang, Geguang Pu, Tao Xie, and Zhendong Su. 2023. Characterizing and Finding System Setting-Related Defects in Android Apps. *IEEE Trans. Software Eng.* 49, 4 (2023), 2941–2963. doi:10.1109/TSE.2023.3236449

[55] Jingling Sun, Ting Su, Jun Sun, Jianwen Li, Mengfei Wang, and Geguang Pu. 2024. Property-Based Testing for Validating User Privacy-Related Functionalities in Social Media Apps. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 440–451. doi:10.1145/3663529.3663863

[56] Android Team. 2021. *Android Debug Bridge (adb)*. Retrieved 2026-3 from https://developer.android.com/tools/adb

[57] Nikolai Tillmann, Jonathan De Halleux, and Tao Xie. 2014. Transferring an automated test generation tool to practice: From Pex to Fakes and Code Digger. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. 385–396. doi:10.1145/2642937.2642941

[58] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, Shao Kun Deng, and Neel Sundaresan. 2020. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020). doi:10.48550/arXiv.2009.05617

[59] uiautomator2 Team. 2021. *uiautomator2*. Retrieved 2026-3 from https://github.com/openatx/uiautomator2

[60] Vasudev Vikram, Caroline Lemieux, Joshua Sunshine, and Rohan Padhye. 2023. Can large language models write good property-based tests? *arXiv preprint arXiv:2307.04346* (2023). doi:10.48550/arXiv.2307.04346

[61] 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

[62] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[63] 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)*. 434–446. doi:10.1145/3540250.3549170

[64] 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 *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 469–480. doi:10.1145/3377811.3380382

[65] Junyang Wang, Haiyang Xu, Haitao Jia, Xi Zhang, Ming Yan, Weizhou Shen, Ji Zhang, Fei Huang, and Jitao Sang. 2024. Mobile-agent-v2: Mobile device operation assistant with effective navigation via multi-agent collaboration. *arXiv preprint arXiv:2406.01014* (2024). doi:10.48550/arXiv.2406.01014

[66] 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

[67] Hao Wen, Hongming Wang, Jiaxuan Liu, and Yuanchun Li. 2023. Droidbot-gpt: Gpt-powered ui automation for Android. *arXiv preprint arXiv:2304.07061* (2023). doi:10.48550/arXiv.2304.07061

[68] Hao Xiao, Jun Sun, Yang Liu, Shang-Wei Lin, and Chengnian Sun. 2013. Tzuyu: Learning stateful typestates. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 432–442. doi:10.1109/ASE.2013.6693101

[69] Yiheng Xiong, Shiwen Song, Bo Ma, Ting Su, and Xiaofei Xie. 2026. PropGen: Automated Property Generation for Property-Based Testing of Mobile Apps (Artifact). doi:10.5281/zenodo.21770804

[70] Yiheng Xiong, Ting Su, Jingling Sun, Jue Wang, Qin Li, Geguang Pu, and Zhendong Su. 2026. From Natural Language to Executable Properties for Property-based Testing of Mobile Apps. arXiv:2603.21263 [cs.SE] doi:10.48550/arXiv.2603.21263

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

[72] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An empirical study of functional bugs in Android apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'23)*. 1319–1331. doi:10.1145/3597926.3598138

[73] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*. 282–291. doi:10.1145/1134285.1134325

[74] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, et al. 2024. On the evaluation of large language models in unit test generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1607–1619. doi:10.1145/3691620.3695529

[75] 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)*. IEEE, 129–139. doi:10.1109/ICST60714.2024.00020

[76] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 1703–1726. doi:10.1145/3660783

[77] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. 183–192. doi:10.1109/ICST.2014.31

[78] 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*. 1–20. doi:10.1145/3706598.3713600

Received 2026-03-26; accepted 2026-06-18