Fully Automated Functional Fuzzing of Android Apps for Detecting Non-crashing Logic Bugs

TING SU, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China
YICHEN YAN, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China
JUE WANG, State Key Lab for Novel Software Technology, Nanjing University, China
JINGLING SUN, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China
YIHENG XIONG, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China
GEGUANG PU, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China
KE WANG, Visa Research, USA
ZHENDONG SU, Department of Computer Science, ETH Zurich, Switzerland

Android apps are GUI-based event-driven software and have become ubiquitous in recent years. Obviously, functional correctness is critical for an app's success. However, in addition to crash bugs, *non-crashing functional bugs* (in short as "non-crashing bugs" in this work) like inadvertent function failures, silent user data lost and incorrect display information are prevalent, even in popular, well-tested apps. These non-crashing functional bugs are usually caused by program logic errors and manifest themselves on the graphic user interfaces (GUIs). In practice, such bugs pose significant challenges in effectively detecting them because (1) current practices heavily rely on expensive, small-scale manual validation (*the lack of automation*); and (2) modern *fully automated* testing has been limited to crash bugs (*the lack of test oracles*).

This paper fills this gap by introducing *independent view fuzzing, a novel, fully automated approach* for detecting non-crashing functional bugs in Android apps. Inspired by metamorphic testing, our key insight is to leverage the commonly-held *independent view property* of Android apps to manufacture property-preserving mutant tests from a set of seed tests that validate certain app properties. The mutated tests help exercise the tested apps under additional, adverse conditions. Any property violations indicate likely functional bugs for further manual confirmation. We have realized our approach as an automated, end-to-end functional fuzzing tool, GENIE. Given an app, (1) GENIE automatically detects non-crashing bugs without requiring human-provided tests and oracles (thus *fully automated*); and (2) the detected non-crashing bugs are diverse (thus *general and not limited to specific functional properties*), which set GENIE apart from prior work.

We have evaluated GENIE on 12 real-world Android apps and successfully uncovered 34 previously unknown non-crashing bugs in their latest releases — all have been confirmed, and 22 have already been fixed. Most of the detected bugs are nontrivial and have escaped developer (and user) testing for at least one year and affected many app releases, thus clearly demonstrating GENIE's effectiveness. According to our analysis, GENIE achieves a reasonable true positive rate of 40.9%, while these 34 non-crashing bugs could not be detected by prior fully automated GUI testing tools (as our evaluation confirms). Thus, our work complements and enhances existing manual testing and fully automated testing for crash bugs.

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

Authors' addresses: Ting Su, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China; Yichen Yan, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China; Jue Wang, State Key Lab for Novel Software Technology, Nanjing University, China; Jingling Sun, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China; Yiheng Xiong, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China; Geguang Pu, Shanghai Key Lab of Trustworthy Computing, East China Normal University, China; Ke Wang, Visa Research, USA; Zhendong Su, Department of Computer Science, ETH Zurich, Switzerland.

1 INTRODUCTION

Android apps are GUI-centered event-driven software. The number and diversity of them have grown rapidly. Recent studies show that app users highly value user experience — only 16% of the users will try a function-failing app more than twice [Compuware 2013; Localytics 2019]. Therefore, ensuring functional correctness is critical for improving an app's success and its user loyalty. However, besides crashes, non-crashing functional bugs[1] (*e.g.*, inadvertent function failures, silent user data lost, incorrect display information) are difficult to detect and frequently escape from standard developer testing. In fact, many such errors are only noticed by end users post-depolyment, and sometimes lead to severe consequences in real life [Motherboard 2020; Sixth Tone 2019].

Although tremendous progress has been made to improve and automate GUI testing of Android apps, existing techniques cannot *fully automatically* detect non-crashing bugs [Kong et al. 2019; Rubinov and Baresi 2018; Su et al. 2021; Tramontana et al. 2019]. For example, Monkey [Monkey 2020], Sapienz [Mao et al. 2016] and Stoat [Su et al. 2017] represent the state-of-the-art GUI testing techniques of Android apps [Choudhary et al. 2015; Wang et al. 2018], but all of them are limited to crashes, and none can detect non-crashing bugs. Neither can the state-of-the-practice automatically achieve this. For example, off-the-shelf test automation frameworks (*e.g.*, Espresso [Espresso 2020], Robotium [Robotium 2020], and Appium [Appium 2020]) can automate test execution, but require carefully-designed manual tests (with oracles) to verify functional correctness. Static analysis tools (*e.g.*, Lint [Lint 2020], FindBugs [FindBugs 2020], and Infer [Infer 2020]) can find generic coding errors, but are ineffective in detecting app-specific non-crashing logic bugs.

One *key difficulty* of automatically finding non-crashing bugs is the lack of test oracles [Barr et al. 2015; Vásquez et al. 2017b]. Due to frequent requirement changes, app developers seldom maintain detailed functional specifications [Vásquez et al. 2017a]. As a result, the current practices heavily rely on expensive, painstaking manual validation [Kochhar et al. 2015; Vásquez et al. 2017a]. Relevant prior work either leverages the codified oracles in the developer tests [Adamsen et al. 2015; Fard et al. 2014] or manually defines oracles for specific app functionalities [Hu et al. 2018; Köroglu and Sen 2019; Lam et al. 2017; Mariani et al. 2018; Rosenfeld et al. 2018; Zaeem et al. 2014] to detect non-crashing bugs, however with limited usability, effectiveness and scalability.[2]

Independent View Property. To tackle this challenge, we propose *independent view fuzzing, a novel, automated approach* to effectively uncovering non-crashing bugs of Android apps without requiring human-provided tests and oracles, and is not limited to specific functionalities. Inspired the idea of metamorphic testing [Chen et al. 1998, 2018; Segura et al. 2016], our key insight is that many apps hold the *independent view property*, that is *interacting with one GUI view (widget) does not affect the states of the others and only adds additional GUI effects*. For example, Fig. 1 shows *ActivityDiary*, an app released on Google Play [ActivityDiary 2020a], which is used to record one's daily activities (we have simplified the GUI pages of this app and its workflow). For example, the three buttons *Cinema*, *Sleeping* and *Cleaning* on ℓ_1 are some predefined activities for selection, which act like the options in a list (these buttons are always visible after selection). Fig. 1(a) shows

---

[1]Our work focuses on non-crashing functional bugs which are caused by program logic errors and lead to GUI issues. Other non-crashing bugs like performance or energy bugs violate non-functional requirements and thus are not within our scope.
[2]Our investigation of 1,752 Android apps on GitHub reveals that only 62 apps ($\approx$ 3.5%) contain the GUI-level tests written by developers, and the number of such tests is typically very small, resulting in limited code coverage.
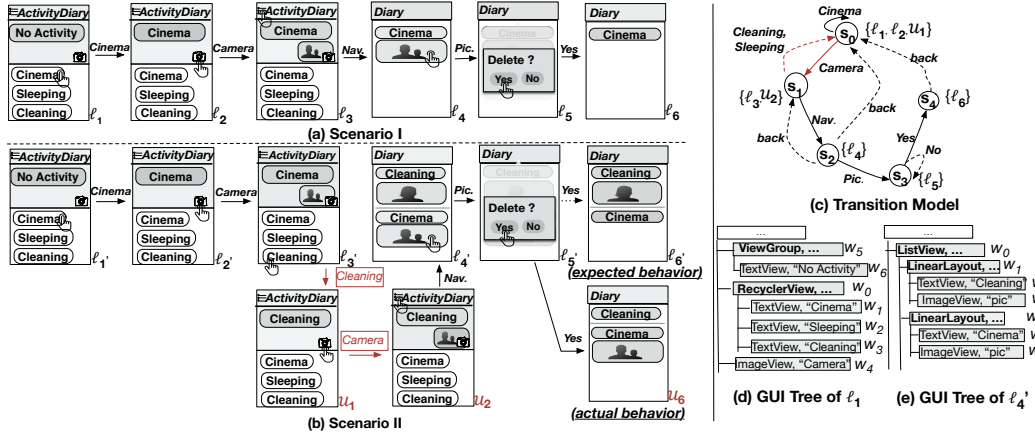
Fig. 1. *ActivityDiary*. (a) Scenario I: one typical usage scenario. The texts on the arrows denote user events (the finger icon is the event location). (b) Scenario II: the expected behavior (ℓ_6') *v.s.* actual behavior (u_6) of *ActivityDiary* by adding two additional events (annotated in the two red boxes) into Scenario I. (c) A partial GUI transitional model. (d) and (e) are the GUI trees of ℓ_1 and ℓ_4', respectively.

one of its basic functionality: a user can start an activity (*e.g.*, *Cinema* in the list on ℓ_1) and attach pictures to this activity to record information (*e.g.*, the movie the user watched in a cinema on ℓ_3). Specifically, the user starts (by clicking) *Cinema* ($\ell_1 \Rightarrow \ell_2$, "No Activity" changes to "Cinema") and attaches a picture to it by clicking the *Camera* button ($\ell_2 \Rightarrow \ell_3$, a picture is added). The user can switch to the *diary* page ($\ell_3 \Rightarrow \ell_4$), where the app shows the history of the user's activities in the reverse chronological order (*i.e.*, *Cinema* in this case), and the user can delete the picture ($\ell_5 \Rightarrow \ell_6$, click "Yes"). Common knowledge tells us that *Sleeping* or *Cleaning* are obviously independent from *Cinema* on page ℓ_3, and thus switching to another activity (*e.g.*, *Cleaning*) on page ℓ_3 following the subsequent events, the deletion of *Cinema*'s picture will not be affected. Fig. 1(b) shows this expected behavior (denoted by the ending page ℓ_6'): when the user switches to *Cleaning* ($\ell_3' \Rightarrow u_1$) and takes a picture for it ($u_1 \Rightarrow u_2$), *Cinema*'s picture can still be successfully deleted ($\ell_5' \Rightarrow \ell_6'$). The only GUI differences are the additions of *Cleaning* and its picture on ℓ_4' and ℓ_6'.

Inspired by the above observation, our key idea is to leverage this independent view property to manufacture app property-preserving mutant tests from a set of seed tests that witness certain properties of an app. These mutant tests validate these properties under additional, adverse conditions. Any property violations could indicate likely non-crashing bugs for manual confirmation. For example, Fig. 1(a) (Scenario I) witnesses one specific app property, *i.e.*, whether *Cinema*'s picture can be correctly deleted. Fig. 1(b) (Scenario II) can be viewed as a mutant test that validates this property by inserting two additional events (*i.e.*, "click *Cleaning* and take a picture") at ℓ_3'. Intuitively, this app property should hold, otherwise a likely functional bug happens. In fact, a real non-crashing bug [ActivityDiary 2020b] did exist. Fig. 1(b) shows the actual app behavior (denoted by the ending page u_6), *Cleaning*'s picture was erroneously deleted while *Cinema*'s picture was still kept. We aim to identify such critical GUI inconsistencies between the seed test and its mutant tests to capture likely non-crashing functional bugs.

Prevalence of the Independent View Property. We find that validating the independent view property can detect many non-crashing bugs. We selected five popular, open-source apps, namely, *Ankidroid* [AnkiDroid Team 2020], *AmazeFileManager* [AmazeFileManager Team 2020], *Antenna-Pod* [AntennaPod Team 2020], *k-9mail* [K-9 Mail Team 2020], and *TeamNewPipe* [NewPipe Team 2020], and manually inspected all their bugs reported within the recent one-year duration. Note that
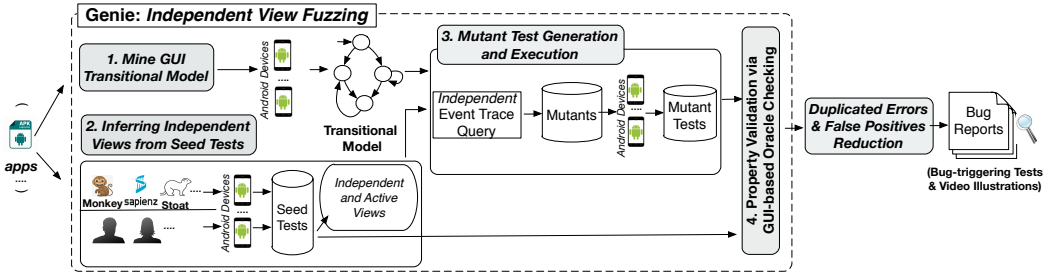
Fig. 2. Workflow of our approach.

these apps are popular (with 1K+ stars on GitHub and 1M+ installations on Google Play), diverse (covering different app categories), and well-maintained (with many developer-written regression tests and oracles). It took us over one-person month in this process to manually inspect the bug reports. We identified 129 non-crashing bugs, 38 of which (29.5%≈38/129) violated the independent view property (the percentages range from 21~43% across these individual apps). Thus, the study results clearly show the general applicability of leveraging this property for detecting a large class of non-crashing bugs in real-world apps. Appendix A.1 details this investigation and the results.

Independent View Fuzzing. To realize the above idea, we face two technical challenges: (1) *the systematic generation of app property preserving mutants*[3]; and (2) *the precise identification of property violations*. This paper proposes two key techniques to overcome these challenges. The first technique analyzes GUI pages and their layouts during the execution of a seed test to infer independent views, and leverages a GUI transitional model to systematically generate independent events. The second technique compares the GUI effects between the seed test and each of its mutants to decide property violations. At the high level, it attempts to capture the intuition that, because of the inserted events are independent, the GUI effects of the seed test should be preserved in any mutant test's execution. In other words, the inserted events should only add, but not remove, GUI effects from the execution of the seed test. Otherwise, it indicates a likely functional bug.

We have realized our techniques as an automated, end-to-end testing tool, Genie. It can automatically find non-crashing bugs of Android apps and report them for manual confirmation. In practice, Genie operates in four main steps as depicted in Fig. 2: (1) mining a GUI transitional model from the app, (2) generating a set of random seed tests and executing each seed test to infer independent views (Genie could also accept seed tests from human or existing test generation tools), (3) leveraging the independent views and the transitional model to guide the generation of mutant tests and executing them, and (4) comparing each seed test and its corresponding mutant tests to identify property violations. Finally, Genie adopts a bug report reducer to remove duplicated reported errors and trivial false positives, and ranks the remaining distinct bug reports according to their occurrences for manual confirmation. Section 2 illustrates how these four main steps work on a real functional non-crashing bug. Section 4.3 details how the bug report reducer removes duplicated errors and false positives, and ranks the remaining bug reports.

Results. Our evaluation on twelve real-world Android apps demonstrates the effectiveness of Genie. To date, Genie has successfully discovered 34 non-crashing bugs, all of which have been confirmed and 22 have already been fixed. All these bugs were discovered in the latest app releases and previously unknown. Moreover, most of the detected non-crashing bugs are nontrivial and long latent — out of the 34 bugs, 26 escaped developer testing for a long duration (1-4 years), while

---

[3]To make it clear, our approach mutates the GUI test inputs rather than the program under test. Throughout the paper, we may use *mutants* for short to denote *mutant tests*.

19 affected more than 10 releases on the app market. According to our analysis, GENIE achieves a reasonable true positive rate of 40.9%, while prior fully automated GUI testing tools for Android cannot detect these 34 non-crashing functional bugs (as our evaluation confirms).

This paper makes the following key contributions:

- At the conceptual level, it proposes independent view fuzzing, the first automated approach to detecting non-crashing functional bugs in Android apps without requiring human-provided tests and oracles by exploiting the commonly-held independent view property of Android apps;
- At the technical level, it introduces two novel, effective techniques to tackle two challenges: the systematic generation of property-preserving mutants from the provided seed tests and the automated GUI-based oracle checking; and
- At the empirical level, it presents the realization of the approach, GENIE, and the evaluation of it on twelve real-world apps, leading to 34 previously unknown, mostly long latent non-crashing bugs (all confirmed; 22 fixed).

2 ILLUSTRATIVE EXAMPLE

We illustrate our approach via the real functional bug of app *ActivityDiary*, introduced in Section 1.

Step 1 (GUI transitional model construction). We dynamically explore the app's GUI pages and mine a GUI transitional model to characterize the app's behavior. In such a model, a node denotes a GUI state (abstracted from a set of structurally equivalent GUI runtime layouts), and an edge denotes a GUI event. For example, Fig. 1(c) shows a partial transitional model of *ActivityDiary*, which includes the app behavior of Scenario I (denoted by the solid lines). In the model, the runtime GUI layouts ℓ_1 and ℓ_2 in Scenario I are mapped to s_0 because of their structural similarity, while ℓ_3 is mapped to a different state s_1 because it has an additional picture view. Section 3.2 defines the state abstraction criterion and explains in detail how to mine such transitional models.

Step 2 (Independent view inference). We generate a set of random seed tests as the basis for property validation. During the execution of a seed test, we examine each page's GUI layout (*i.e.*, a GUI tree) to infer independent views within each group view. Specifically, a *group view* is a type of GUI views (*e.g.*, ListView and RecyclerView) for arranging/displaying a number of *relevant, but functionally independent* child views, suggested by GUI design guidelines [Material Design 2020]. On each layout, we also maintain the *active view information* within each group view to track which view is currently active (*i.e.*, executed) and update this information across subsequent layouts when possible. This enables us to infer independent events at any layout of the seed test with the previous execution information.

For example, we take Scenario I in Fig. 1(a) as a seed test, and execute it. On the layout ℓ_1, view *Cinema*, *Sleeping*, and *Cleaning* are independent views. Because from Fig. 1(d), ℓ_1's simplified GUI tree, we observe that *Cinema*, *Sleeping*, and *Cleaning* are three independent siblings rooted by the same group view RecyclerView. Specifically, on layout ℓ_1, *Cinema* will be annotated as *active* within its group view when it is clicked. The other two views, *Sleeping* and *Cleaning*, are by default annotated as *inactive*. This information will also be updated on ℓ_2 and ℓ_3, but will not be updated on ℓ_4 (because *Cinema* does not exist on ℓ_4). On ℓ_4, *picture* will be annotated as active when it is clicked. Note that on ℓ_2, *Camera* will not be annotated as active when clicked because it does not belong to any group view (see Fig. 1(d)). In this way, we can know, on each layout, which views are active or inactive within each group view. Section 3.3 details how we infer independent views and maintain their active view information on each GUI layout during the execution of seed tests.

Step 3 (Mutant test generation and execution). In this step, we leverage the independent view property to systematically generate independent event traces for creating mutant tests. Specifically, we (1) pick a *pivot layout* from the seed test, and use the information from Step 2 to select an

inactive view from any group view as the start of an independent event trace, and (2) use the transitional model to guide the generation of remaining events and return back to this pivot layout. Conceptually, we assume that such a "loop trace" will exercise some independent functionalities *w.r.t.* the subsequent events in the seed test, and thus preserve the app behavior of the seed test.

For example, if we pick ℓ_3 (in Fig. 1(a)) as the pivot layout, based on the active view information, we know that *Cleaning* is inactive and independent from *Cinema*. Then, we start from *Cleaning*, and query the transitional model (see Fig. 1(c)) to obtain any possible event traces that can return back to layout ℓ_3. Here, one possible independent event trace is "click *Cleaning* and take a picture" (denoted by the red lines). In this way, we can successfully create the mutant test, *i.e.*, the test of Scenario II in Fig. 1(b), which can manifest the bug. Section 3.4 details how we systematically generate mutant tests by leveraging the active view information and the transitional model.

Step 4 (GUI-based oracle checking). This step computes the differences of GUI effects between the seed test and each of its mutants to identify possible property violations, thus functional bugs. Specifically, we use the differences of GUI views (denoted by Δ) between two GUI pages to characterize the GUI effects of executing the event trace in between. This formulation is general and enables us to perform flexible oracle checking. If Δ of the seed test is not contained in Δ' of the mutant test (*i.e.*, $\Delta \not\subset \Delta'$), a likely functional bug is detected.

For example, in Fig. 1(a), by comparing layouts ℓ_4 and ℓ_6, the GUI effect of the event trace between ℓ_4 and ℓ_6 can be characterized by the deletion of a picture under *Cinema* (denoted as Δ). Similarly, in Fig. 1(b), we can know the GUI effect of the event trace between ℓ'_4 and u_6 can be characterized by the deletion of a picture under *Cleaning* (denoted as Δ'). Obviously, $\Delta \not\subset \Delta'$, *i.e.*, these two GUI effects are not the same — the picture was deleted from different activities. Thus, we detect a likely functional bug and report it for manual confirmation, which is indeed a real functional bug. Section 3.5 details how we formulate and perform this GUI-based oracle checking.

3 INDEPENDENT VIEW FUZZING

This section presents the formulation and technical details of our approach. It starts by introducing the needed notations and definitions, and then describes the approach's four core steps.

3.1 Notations and Definitions

Given an Android app, our goal is to generate property-preserving mutant tests from a set of seed tests. An Android app is a GUI-centered event-driven program P. Each of its GUI pages is a runtime GUI layout ℓ, *i.e.*, a GUI tree T. Each node of this tree is a GUI view (or widget) w ($w \in \ell$). Specifically, each view w has a type $w.type$, which denotes its view type. For example, a leaf view's $w.type$ can be a button (Button) or a text field (EditText). A non-leaf view's $w.type$ can be a layout view, *e.g.*, ListView, LinearLayout, which arranges or displays other views in specific orders and appearances. When P is not in the foreground, we define $\ell = \bot$.

A GUI event $e = \langle t, r, o \rangle$ is a tuple, where $e.t$ denotes e's event type (*e.g.*, click, edit, swipe, or a system event), $e.r$ is a function to obtain e's receiver view (specifically, we use $e.r(\ell) = w$ to denote that e can be executed on the view $w \in \ell$ at runtime), and $e.o$ denotes the optional data associated with e (*e.g.*, a string/number for edit).

A GUI test is a sequence of events $E = [e_1, e_2, \ldots, e_n]$. E can be executed on P to obtain an execution trace $\Pi_P(E) = \langle L, I, H \rangle$, where L is a sequence of runtime layouts $L = [\ell_1, \ell_2, \ldots, \ell_{n+1}]$, I and H (a map) record the independent views and the active view information for each layout, respectively (discussed in Section 3.3). Specifically, we use $\ell_{i+1} = e_i(\ell_i)$ to denote the execution of e_i, where ℓ_{i+1} is the layout due to the execution of e_i on ℓ_i. In practice, we use these runtime GUI layouts to approximate app states [Yuan and Memon 2007].

Next, we describe the details of our approach's four steps as illustrated in Section 2.

3.2 Mining GUI Transitional Model

This step aims to mine a GUI transitional model to represent an app's behavior. This model is represented as a tuple $M = \langle S, \Sigma, \delta \rangle$, where S denotes a set of abstract states $\{s|\ s$ is a subset of layouts in $L\}$, Σ the set of events that P accepts, and $\delta : S \times \Sigma \rightarrow S$ the transition function with each transition $\langle s, e, s' \rangle \in \delta$ denoting a state transition from s to s' triggered by e.

In practice, to balance the model size and its precision, we designed a variant of the C-Lv4 GUI comparison criterion of AMOLA [Baek and Bae 2016] to decide whether two runtime layouts ℓ and ℓ' are structurally equivalent. If ℓ and ℓ' are structurally equivalent, they are grouped together as one state s in M to mitigate state explosion. This criterion compares view composition between two layouts to decide equality. We define the Equivalent-with relation (denoted by $\simeq$) below.

DEFINITION 1. **Equivalent-with** ($\simeq$). *Let layout ℓ be represented as $\{w_1, w_2, \dots, w_{|\ell|}\}$ ($|\ell|$ is the number of widgets in ℓ), we abstract ℓ into $\ell_{abs} = \bigcup_{w \in \ell} \{w.type\}$, which approximates the structure information of ℓ via a type abstraction that differentiates views through their types.* [4] *Given layout ℓ and ℓ', if $\ell_{abs} = \ell'_{abs}$, Equivalent-with(ℓ, ℓ') holds, i.e., $\ell \simeq \ell'$.*

Example. In Fig. 1(a), $\ell_1 \simeq \ell_2$ because their structures are equivalent (both are similar to Fig. 1(d)). $\ell_2 \neq \ell_3$ because ℓ_3 has a picture view. For ℓ_4 in Fig. 1(a) and ℓ'_4 in Fig. 1(b), $\ell_4 \simeq \ell'_4$ because a ListView will have only two abstract states, *i.e.*, empty and non-empty (with any number of children) under this relation. Fig. 1(e) shows the structure of ℓ'_4. Fig. 1(c) shows the GUI model M.

State coverage-optimized model mining. We propose a *state coverage-optimized* GUI mining algorithm for constructing the transitional model. The basic mining process is, from a given GUI layout ℓ, we select and execute an event e on its receiver view $e.r(\ell)$. Then, e takes us to a new layout ℓ', from which we continue to select the next event. During this process, a self-loop transition (s, e, s') is added into M if $\ell_{abs} \simeq \ell'_{abs}$ (*i.e.*, Equivalent-with(ℓ, ℓ')). Otherwise, a transition connecting the two (different) states is added. Here, s and s' are the states in M corresponding to the abstract layouts ℓ_{abs} and ℓ'_{abs}, respectively. If $\ell' = \bot$, we restore P to the foreground and continue. To improve the mining performance, the model mining algorithm synergistically combines two existing strategies, *i.e.*, *systematic event selection* (adapted and significantly extended from Stoat [Su et al. 2017]'s weighted GUI exploration strategy) and *random event selection* (mimicking Android Monkey [Monkey 2020]'s random exploration strategy).

- *Systematic event selection.* This strategy prioritizes an event e if e is less frequently executed and can transit to a new layout with more new events. Specifically, we assign each event e a weight $\Upsilon(e)$ to capture these two pieces of information. Assume we are at the ith round of event execution, *i.e.*, $\ell_{i+1} = e_i(\ell_i)$, we first recursively update the weight $\Upsilon(\cdot)$ for each event e in the worklist by Formula (1) (the worklist stores all the events that are identified on previous runtime GUI layouts before ℓ_{i+1}, *i.e.*, $[\ell_1, \dots, \ell_i]$), and then select the event with the largest weight on layout ℓ_{i+1}:

$$\Upsilon_{i+1}(e) = \frac{\Upsilon_i(e) + \sum \Upsilon_i(new\_events(e))}{exec\_times(e)^2} \tag{1}$$

where, $new\_events(\cdot)$ denotes the set of events on the layout due to the execution of e in the history, $exec\_times(\cdot)$ records the executed times of e during mining ($i \geq 1$). Initially, $\forall e \in E. \Upsilon_1(e) = 100, exec\_times(e) = 1$, and $new\_events(e) = \emptyset$.

---

[4]Our implementation associates each view's type $w.type$ with its resource-id and content-description from the corresponding GUI tree node, but omits text and other attribute values (*e.g.*, Clickable).

- *Random event selection.* This strategy randomly selects events according to their types with predefined probabilities (our implementation sets 60% for touch events, 35% for long-touch events, and 5% for navigation events). It reduces the possibility of exploration being trapped on a specific GUI page (*e.g.*, repeatedly selecting events from a long ListView with similar items).

During model mining, we interleave these two strategies to combine their effectiveness like hybrid symbolic execution [Majumdar and Sen 2007]. Indeed, we first apply systematic event selection and attempt to cover as many app states as possible. Upon saturation after a certain number of rounds, we apply the random strategy to discover possible new states, and then switch back to systematic event selection.

Example. Fig. 1(c) shows a partial transitional model of *ActivityDiary*. The nodes denote the abstract states, while the solid edges are the events from Scenario I (Fig. 1(a)) and the dotted edges are the events from other explorations.

3.3 Inferring Independent Views from Seed Tests

Given a seed test, this step aims to infer independent views on each layout and maintain their active view information along the execution trace. This information will be leveraged to manufacture property-preserving mutant tests in the next step (Section 3.4). The key of this analysis is, given a GUI layout, to identify which views are independent from each other and which views are active according to previous event executions. To this end, we define the Independent_from relation (denoted by ⊥⊥) and the concept of *active views*.

DEFINITION 2. **Independent_from** *(⊥⊥). Conceptually, a GUI layout ℓ can be partitioned into non-overlapping regions that are rooted by a set of group views $G = \{w_{g_1}, w_{g_2}, \ldots, w_{g_n}\}$, where $w_{g_i} \in \ell$ is a non-leaf layout view of* group view *type.[5] These regions serve different functional purposes. Group(w) denotes the group view of w. We define view $w_1 \in \ell$ and $w_2 \in \ell$ to be independent, i.e., $w_1 \perp\!\!\!\perp w_2$, if*

(1) w_1 and w_2 are the children of two different group views in G, i.e., Group(w_i) ≠ Group(w_j).
(2) w_1 and w_2 are the children of the same group view, i.e., Group(w_i) = Group(w_j), and satisfy that w_1 and w_2 are siblings[6] and w_1.type = w_2.type.

Note the views that are not within any group view are assumed independent from those views in group views. We say any views that are independent from others are independent views.

Example. Fig. 1(d) shows the GUI tree of ℓ_1, where w_1 (*Cinema*), w_2 (*Sleeping*) and w_3 (*Cleaning*) are independent views and rooted by the group view w_0 (RecyclerView). Obviously, clicking one of them will not affect the states of others. w_6 (*No Activity*) $\perp\!\!\!\perp w_1$ holds because w_6 is in another group view w_5 (ViewGroup). w_4 (*Camera*) $\perp\!\!\!\perp w_1$ also holds because w_4 is not within a group view. Fig. 1(e) shows the GUI tree of ℓ_4', where w_1 (*i.e.*, *Cleaning* and its picture) $\perp\!\!\!\perp w_4$ (*Cinema* and its picture) holds because they are rooted by the group view w_0 (ListView). Obviously, clicking *Cinema*'s picture will not affect the state of *Cleaning*'s picture.

DEFINITION 3. *Active View. Let $G = \{w_{g_1}, w_{g_2}, \ldots, w_{g_n}\}$ be the set of group views of layout ℓ and $h_\ell \in H$ be the map that maintains the active view information of ℓ, we define the* active view *within each group view as follows:*

(1) A view w_i becomes active *when an event e, satisfying e.r(l) = w_i, was executed on ℓ. The active view information of ℓ is updated as h_ℓ[Group(w_i)] = w_i.*

---

[5] According to GUI design guidelines [Material Design 2020], the typical group view types include RecyclerView, ListView, GridView, ViewGroup, RadioGroup, LinearLayout, GridLayout, *etc.*
[6] To ease explanation, we assume w_1 and w_2 are leaf views. When w_1 and w_2 are non-leaf views, their respective children are independent from each other. Our implementation considers both cases.

(2) A view w_i becomes inactive *when an event e, satisfying $e.r(l) = w_j \wedge \text{Group}(w_i) = \text{Group}(w_j)$, was executed on ℓ. The active view of ℓ is updated as $h_\ell[\text{Group}(w_j)] = w_j$.*
(3) A view w_i stays active *or* inactive*, when an event e, satisfying $e.r(l) = w_j \wedge \text{Group}(w_i) \neq \text{Group}(w_j)$, was executed on ℓ. The active view of ℓ is updated as $h_\ell[\text{Group}(w_j)] = w_j$.*

Note that those views not within any group view are always assumed *inactive*. More importantly, we maintain the active view information along the execution trace, *i.e.*, update this information of ℓ to one subsequent layout ℓ' if similar views can be located on ℓ'. This enables us to leverage the independent view property at any layout of a seed test informed by previous event executions.

Example. Assume Scenario II in Fig. 1(b) denotes a seed test. *Cinema* will become *active* when it is clicked on layout ℓ'_1. *Sleeping* and *Cleaning* are by default *inactive*. *Cleaning* will become *active* when it is clicked on ℓ'_3, while *Cinema* will become *inactive*. Because *Cinema* and *Cleaning* are in the same group view. During event executions, this active view information will be updated from ℓ'_1 to ℓ'_2 (*i.e.*, *Cinema* is active on ℓ'_2), and from ℓ'_3 to u_1 and u_2 (*i.e.*, *Cleaning* is active on u_1 and u_2) since we can locate the similar view *Cinema* on ℓ'_2 and *Cleaning* on u_1 and u_2, respectively. But it will not be updated on ℓ'_4 because *Cleaning* does not exist on ℓ'_4. On the other hand, clicking *Camera* on ℓ'_2 and u_1 will not affect the active view information of *Cinema* or *Cleaning* as they are in different group views. *Camera* itself is always *inactive* because it is not within any group views.

To maintain the active view information, we define the Similar_with relation (denoted by $\sim$) for locating similar views between two GUI layouts. It is also used for mutant execution in Section 3.4.

DEFINITION 4. $\textbf{Similar\_with}$ $(\sim)$. *Let $\ell = \{w_1, \ldots, w_{|\ell|}\}$ be the layout on which event e is executed, and $\ell' = \{w'_1, \ldots, w'_{|\ell'|}\}$ be another layout. Obviously, there exists a receiver view $w_i \in \ell$ satisfying $e.r(\ell) = w_i$. If there exists a "similar" view $w'_j \in \ell'$ such that w'_j should be the receiver view of e on ℓ', i.e., $e.r(\ell') = w'_j$, we say $\text{Similar\_with}(w_i, w'_j)$, i.e., $w_i \sim w'_j$.*

Locating similar views. Specifically, to locate the similar view $w'_j \in \ell'$ w.r.t. $w_i \in \ell$, we first convert the type information of w_i (including its text if it exists) into a string. Then, we use string equality to check whether there exists at least one exactly matched view on ℓ'. If no such views exist on ℓ', we conclude no similar views can be found. Otherwise, we use the type information of w_i (including all its children views) to find the "most similar" view $w'_j \in \ell'$ in terms of string edit distance.

Specifically, we update the active view information for an independent view from ℓ to ℓ' only if we can locate a similar view, otherwise we abandon that update.

Inferring independent views from seed tests. Algorithm 1 gives the process of inferring independent views and maintaining their active view information during the execution of a seed test E. The seed test was randomly generated in our approach (detailed in Section 4.1). Recall that, a seed test E's execution trace is $\Pi_P(E) = \langle L, I, H \rangle$, where L is the sequence of layouts, I and H record the independent views and the active view information for each layout, respectively. The algorithm annotates the receiver view of e as active (if it is within a group view) on the current layout (Line 4). Then, it sends event e to the app, which transitions to a new layout (Lines 5-6). Next, it updates the active view information from prior GUI layouts to the new layout (Line 7). The function $\text{Update\_active\_views}$ internally updates the active view information from the most recent similar GUI layout to the current one (Lines 11-16). $\text{Similar\_layout\_type}$ (Line 12) checks whether two layouts are of similar type (*i.e.*, denoting similar functional page), which is a coarse version of Equivalent_with. In practice, $\text{Similar\_layout\_type}$ leverages the layout information to characterize the page type. For example, in Fig. 1(a), ℓ_2 and ℓ_3 have the similar layout type because they denote the same *main* page (although they are not structurally equivalent). Thus, we can

---

Algorithm 1: Maintaining independent views and their active view information during the execution of a seed test.

1 **Function** $\text{Execute}_P(\textit{Seed test } E)$:
2 　　$\ell \leftarrow \text{GetGUI}(); L \leftarrow [\ell]; I \leftarrow []; I \leftarrow I :: \text{Get\_independent\_views}(\ell); H \leftarrow []$
3 　　**foreach** $e \in E$ **do**
4 　　　　$h[\text{Group}(e.r(l))] \leftarrow e.r(l); H \leftarrow H :: [h]$
5 　　　　$\text{SendEventToApp}(e.t, e.r(\ell), e.o)$
6 　　　　$\ell \leftarrow \text{GetGUI}()$
7 　　　　$h \leftarrow \text{Update\_active\_views}(\ell, \langle L, I, H \rangle)$
8 　　　　$L \leftarrow L :: [\ell]; I \leftarrow I :: \text{Get\_independent\_views}(\ell);$
9 　　**return** $\langle L, I, H \rangle$
10 **Function** $\text{Update\_active\_views}(\textit{layout } \ell, \textit{trace } \langle L, I, H \rangle)$:
　　　　// update the active view info from the prior layouts
11 　　**foreach** $\langle \ell_i, h_i \rangle \in \langle L, H \rangle.\text{reverse}()$ **do**
12 　　　　**if** $\text{Similar\_layout\_type}(\ell_i, \ell)$ **then**
　　　　　　// Group(w_a)=w_g, and w_a is active
13 　　　　　　**foreach** $\langle w_g, w_a \rangle \in h_i$ **do**
14 　　　　　　　　**if** $(\exists w_g' \in \ell.w_g \sim w_g') \wedge (\exists w_a' \in \ell.w_a \sim w_a')$ **then**
　　　　　　　　　　// update the active view info
15 　　　　　　　　　　$h[w_g'] \leftarrow w_a'$
16 　　　　　　**return** h

---

update the active information from ℓ_2 to ℓ_3. The active view information is updated according to the Similar_with relation (Lines 13-15).

3.4 Mutant Test Generation and Execution

This step inserts independent event traces at a given pivot layout of a seed test to manufacture mutant tests. Formally, let $E = [e_1, e_2, \ldots, e_n]$ be a seed test, and $L = [\ell_1, \ell_2, \ldots, \ell_{n+1}]$ be the runtime layouts yielded by E, we can create a mutant test $E' = [e_1, \ldots, e_{i-1}, e_1', e_2', \ldots, e_m', e_i, \ldots, e_n]$ by inserting an event trace $\tau = [e_1', e_2', \ldots, e_m']$ at a given pivot layout $\ell_i \in L$ ($\ell_i = e_{i-1}(\ell_{i-1})$). Importantly, τ should satisfy (1) independent from subsequent events; and (2) connecting with the event traces before and after the insertion position.

> **DEFINITION 5.** ***Independent event trace.*** *In our context, a valid inserted trace τ should satisfy: $e_1'.r(\ell_i)$ is inactive, i.e., the receiver view of τ's first event e_1' on ℓ_i is independent from any active views on ℓ_i. In this case, we say τ is an independent event trace. In other words, τ is independent from subsequent events in the seed test. Here, we leverage the active view information maintained by $\Pi_P(E) = \langle L, I, H \rangle$ in Section 3.3. Section 5.4 validates the generality of this assumption.*

> **DEFINITION 6.** Connect_with ($\rightsquigarrow$). *Let $\tau_1 = [e_1, e_2, \ldots, e_{|\tau_1|}]$ and $\tau_2 = [e_1', e_2', \ldots, e_{|\tau_2|}']$ be two arbitrary event traces. We say τ_1 can connect with τ_2 (denoted by $\tau_1 \rightsquigarrow \tau_2$) if τ_2's first event e_1' can locate a similar receiver view on τ_1's last GUI layout $\ell_{|\tau_1|+1}$, i.e., $e_1'.r(\ell_{|\tau_1|+1}) \neq \bot$.*

Thus, a valid inserted trace τ should satisfy $[e_1, \ldots, e_{i-1}] \rightsquigarrow \tau \rightsquigarrow [e_i, \ldots, e_n]$, i.e., τ can connect the event traces before and after the insertion position, i.e., $e_1'.r(\ell_i) \neq \bot \wedge e_i.r(\ell_{m+1}') \neq \bot$. Specifically, we will use the Similar_with relation defined in Section 3.3 to locate similar views.

Example. Let Scenario I in Fig. 1(a) be a seed test. Assume that we select ℓ_3 as a pivot layout to insert independent traces. The layout ℓ_3 can be mapped to an abstract state s_1 in the transitional

---

Algorithm 2: Generate mutant tests from a seed test

---

1   $\langle L, I, H \rangle \leftarrow \text{Execute}_P(E)$; *mutants* $\leftarrow []$

2   **Function** Generate_mutants(*Seed test E, Model M*)**:**

3   **foreach** $\langle \ell_i, h_i \rangle \in \langle L, H \rangle$ **do**

4          $\ell_{abs} \leftarrow \text{Find\_equivalent\_state}(M, \ell)$

5   *independent_traces* $\leftarrow \text{Search\_traces}(i, \ell_{abs}, M)$

6   *tests* $\leftarrow \text{Assemble}(E, \ell, independent\_traces)$

7   *mutants* $\leftarrow$ *mutants* :: *tests*

8   **return** *mutants*

9   **Function** Search_traces(*insert position i, abstract layout ℓ_{start}, Model M*)**:**

10   *candidate_traces* $\leftarrow []$; *stack* $\leftarrow \{\}$;

11   **foreach** $\langle \ell_{start}, e, \ell_{next} \rangle \in M[\ell_{start}]$ **do**

         // Get the transitions starting from $\ell_{start} \in M$

12   **if** $e.r(L[i]) \in \text{Get\_independent\_views}(L[i]) \wedge \text{Is\_inactive}(e)$ **then**

             // Pick τ's the starting event from inactive views on ℓ_i, *i.e.*, $L[i]$

13   *stack*.Push$\{[e]\}$

14   **while** *stack* **do**

15          $\tau \leftarrow$ *stack*.Pop() // Pop τ in breadth-first random

16          $\ell_{\tau-last} \leftarrow \text{Get\_last\_layout}(\tau)$

17   **if** $(\ell_{start} \simeq \ell_{\tau-last}) \wedge ([e_0, \ldots, e_{i-1}] \rightsquigarrow \tau \rightsquigarrow [e_i, \ldots, e_n]) \wedge |\tau| < \text{MAX\_LENGTH}$ **then**

             // Form a loop trace satisfying Connect_with.

18   *candidate_traces* $\leftarrow$ *candidate_traces* :: $[\tau]$

19   **else**

20   **foreach** $\langle \ell_{\tau-last}, e, \ell'_{next} \rangle \in M[\ell_{\tau-last}]$ **do**

21   *stack*.Push(τ :: $[e]$)

22   **if** $|candidate\_traces| > \text{MAX\_NUM}$ **then**

23   break

24   **return** *candidate_traces*

---

model (Fig. 1(c)). From ℓ_3's layout information, we know that *Cinema*, *Clearning* and *Sleeping* are independent views and *Cinema* is active. Thus, from ℓ_3, a valid event trace τ could start from *Cleaning* or *Sleeping*. If τ's maximum length is set as 2, we can generate at least two valid independent event traces from the model (denoted by the loop traces in red lines), *i.e.*, $\tau_1 = \langle Cleaning, Camera \rangle$, and $\tau_2 = \langle Sleeping, Camera \rangle$. Note that τ_1 and τ_2 satisfy the Connect_with relation. Take τ_1 as an example, τ_1's first event, *i.e.*, clicking *Cleaning*, can find a similar receiver view *Cleaning* on the runtime layout ℓ_3 (τ_1's starting layout), while the original event clicking *Nav.* on ℓ_3 (following τ_1) can find a similar receiver view *Nav.* on the abstract layout s_1 (*i.e.*, τ_1's last layout). Thus, τ_1 forms a valid independent loop trace that can start from ℓ_3 and return back to ℓ_3.

Generate mutant tests from a seed test. Algorithm 2 gives how our approach generates independent event traces to manufacture property-preserving mutants. The algorithm iteratively selects each GUI layout of a seed test to generate mutants (Lines 3-7). It first maps the pivot layout into an abstract state ℓ_{abs} on the transitional model M (Line 4), from which it starts searching independent traces (Line 5). In function Search_traces, the search of valid independent event traces is conducted on the transitional model M in a breadth-first random manner (Lines 14-23). The search process is detailed in Section 4.2. Specifically, a valid trace τ should (1) be independent from

subsequent events (Lines 11-13), and (2) form a loop trace at ℓ_{abs} that satisfies the Connect_with relation (Lines 17-18).

3.5 GUI-based Oracle Checking

Now, we turn to the difficult challenge of test oracles and automated oracle checking. Our key intuition is that, because the inserted event trace is independent, it would add, but not remove GUI effects from the execution of the seed test. If any GUI effect has been removed, it indicates a likely property violation. We leverage this intuition to achieve automated oracle checking.

Specifically, we define *GUI effect* as the GUI changes between two GUI layouts with similar types, which characterizes the effect of executing the events in between.

DEFINITION 7. **GUI Effects**. *A GUI test E can yield a sequence of runtime GUI layouts* $L = [\ell_1, \ldots, \ell_{n+1}]$. *Given any two GUI layouts,* ℓ_i *and* ℓ_j $(i < j)$, *satisfying* Similar_layout_type(ℓ_i, ℓ_j), *their GUI effect, denoted by* $\Delta(\ell_i, \ell_j)$, *is equivalent to the differences of GUI views between* ℓ_i *and* ℓ_j. $\Delta(\ell_i, \ell_j)$ *characterizes the effect (i.e., changes of views) of executing the event trace* $[e_i, \ldots, e_{j-1}]$ *between* ℓ_i *and* ℓ_j. *Formally,* $\Delta(\ell_i, \ell_j)$ *can be represented as:*

$$\Delta(\ell_i, \ell_j) \equiv (\ell_i \setminus \ell_j) \uplus (\ell_j \setminus \ell_i) \tag{2}$$

where $\setminus$ *denotes set difference and* $\uplus$ *disjoint union. We use* $(\ell_i \setminus \ell_j)$ *and* $(\ell_j \setminus \ell_i)$ *to model view deletions and additions, respectively. Therefore,* $\Delta(\ell_i, \ell_j)$ *captures both deleted and added views from* ℓ_i *to* ℓ_j, *thus the GUI effects of executing the event trace* $[e_i, \ldots, e_{j-1}]$ *between* ℓ_i *and* ℓ_j.

In practice, to efficiently compute $\Delta(\ell_i, \ell_j)$, we adapt the classic tree edit distance algorithms [Bille 2005; Pawlik and Augsten 2016; Zhang and Shasha 1989] to compute the minimal edit operations between the two ordered GUI trees T_i and T_j (corresponding to ℓ_i and ℓ_j). By analyzing these edit operations, we can quickly identify the changes of views, *i.e.*, which views are deleted, added or changed. More specifically, $\Delta(\ell_i, \ell_j)$ contains three types of tuples: (1) $(w, \perp) \in \Delta(\ell_i, \ell_j)$ for $w \in \ell_i$ denotes a view deletion, (2) $(\perp, w) \in \Delta(\ell_i, \ell_j)$ for $w \in \ell_j$ denotes a view addition, and (3) (w, w') for $w \in \ell_i$, $w' \in \ell_j$ and $w \neq w'$ denotes a view change.

Example. In Fig. 1(a), the GUI effect between ℓ_4 and ℓ_6, *i.e.*, $\Delta(\ell_4, \ell_6)$, is the deletion of a picture under *Cinema*, which characterizes the effect of executing the event trace ["*Pic.*", "*Yes*"] in between. When we use tree edit distance to do computation, $\Delta(\ell_4, \ell_6)$ can be represented as $\{(pic_{Cinema}, \perp)\}$. Note that ℓ_4 and ℓ_6 is comparable because they are of same layout type (*i.e.*, the *Diary* page), while ℓ_3 and ℓ_4 is not comparable because they are of different layout types (ℓ_3 denotes the *main* page of *ActivityDiary*). Similarly, in Fig. 1(b), the GUI effect between ℓ_4' and ℓ_6' (in Fig. 1(b)) is $\Delta(\ell_4', \ell_6') = \{(pic_{Cinema}, \perp)\}$, while the GUI effect between ℓ_4' and u_6 is $\Delta'(\ell_4', u_6) = \{(pic_{Cleaning}, \perp)\}$.

Next, we introduce our method for oracle checking based on the definition of GUI effects.

DEFINITION 8. **Oracle Violation**. *Given a GUI seed test E with its GUI layouts* $L = [\ell_1, \ell_2, \ldots, \ell_{n+1}]$, *and one of its mutants E′ with GUI layouts* $L' = [\ell_1', \ell_2', \ldots, \ell_{n+1}']$, *if there exists* $(\ell_i, \ell_j) \in L \times L$ *such that* $\Delta(\ell_i, \ell_j) \not\subseteq \Delta(\ell_i', \ell_j')$, *i.e., an oracle violation is detected.*

Example. Fig. 3 shows the oracle checking between Scenario I in Fig. 1(a) and Scenario II in Fig. 1(b) (actual behavior). The red rectangle annotates the inserted independent events. Note that we only need to compare the layouts affected by the inserted traces. Thus, the only instance of oracle checking in this case is between $\Delta(\ell_4, \ell_6)$ from Scenario I and $\Delta'(\ell_4', u_6)$ from Scenario II because ℓ_4 and ℓ_6 are of same layout type. Obviously, due to $\{(pic_{Cinema}, \perp)\} \not\subseteq \{(pic_{Cleaning}, \perp)\}$, *i.e.*, $\Delta(\ell_4, \ell_6) \not\subseteq \Delta'(\ell_4', u_6)$, a likely bug was reported by Scenario II (actual behavior). In another case, if we do oracle checking between Scenario I in Fig. 1(a) and Scenario II in Fig. 1(b) (expected behavior),
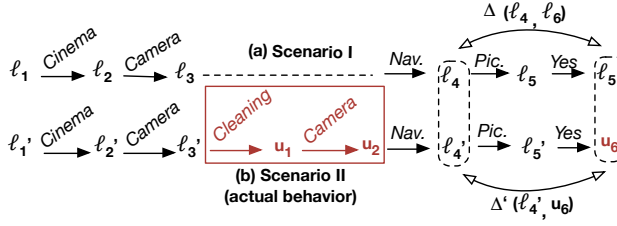
Fig. 3. Oracle checking between Scenario I in Fig. 1(a) and Scenario II (actual behavior) in Fig. 1(b).

Algorithm 3: GUI-based Oracle Checking

---

1 **Function** Oracle_Checking(*Seed test E, Mutant test E′*):
2      $L \leftarrow [\ell_1, \ldots, \ell_k, \ell_{k+1}, \ldots \ell_{n+1}]; L' \leftarrow [\ell_1, \ldots, \ell_k] :: L_\tau :: [\ell'_{k+1}, \ldots \ell'_{n+1}];$
3      $O \leftarrow \text{True};$
4   **foreach** $(\ell_i, \ell_j) \in L \times L$ **do**
5   **if** $(i \leq k \wedge k + 1 \leq j) \vee (k + 1 \leq i < j)$ **then**
6   **if** Similar_layout_type(ℓ_i, ℓ_j) **then**
7   **if** $\Delta(\ell_i, \ell_j) \not\subset \Delta(\ell'_i, \ell'_j)$ **then**
8                      $O \leftarrow \text{False};$

9   **return** O;

---

no oracle violation exists. For example, if we compare $\Delta(\ell_4, \ell_6)$ and $\Delta(\ell'_4, \ell'_6)$, we can easily know $\Delta(\ell_4, \ell_6) \subseteq \Delta(\ell'_4, \ell'_6)$. Thus, Scenario II in Fig. 1(b) (actual behavior) will not report any bug.

GUI-based Oracle Checking. Algorithm 3 gives our GUI-based oracle checking procedure. L and L' denote the sequences of layouts of a seed test E and one of its mutant test E', respectively. L_τ is the sequence of layouts due to the inserted trace τ. Specially, we only need to compare the layouts affected by the inserted traces to improve checking performance (Line 5), and focus on the layouts with similar types to improve checking precision (Line 6).

4 IMPLEMENTATION

We implemented GENIE as an automated, end-to-end functional fuzzing tool for Android apps. Given an app as input, it outputs any likely distinct bug reports. GENIE was built upon DroidBot [Li et al. 2017] and significantly extended Stoat's model mining algorithm [Su et al. 2017]. It was written in Python and JavaScript (5,671 lines of Python for the core algorithms; 1,035 lines of Python for parallel fuzzing, and 386 lines of JavaScript for bug report visualization). Specifically, it uses AccessibilityService [Accessibility 2020] to collect GUI layouts, the APTED algorithm [Pimentel 2020] to compute tree edit distance, ADB [ADB 2020] to send GUI events, including touch, long-touch, edit, and navigation (*e.g.*, back, scroll). We have made GENIE publicly available at *https://github.com/functional-fuzzing-android-apps/home*.

4.1 Seed Test Generation

We generate random seed tests as the basis of property validation, which are expected to be much more diverse, practical and scalable to obtain. To improve the diversity of seeds, we adapted the state coverage-optimized mining algorithm in Section 3.2 for seed generation. Specifically, we keep the weight information of events during seed test generation, which will automatically drive each seed test to reach different GUI pages. Meanwhile, we exploit the motif events [Jue et al. 2020; Mao et al. 2016] to improve the chance of generating meaningful seed tests (*e.g.*, more likely clicking

"OK" than "Cancel" after filling the text fields in a dialog; more likely taking a picture than clicking "Cancel" after opening the camera).

4.2 Mutant Generation and Execution

Algorithm 2 uses `Search_traces` to search independent event traces and create mutants. However, we face the path explosion problem. To this end, we did some optimizations. (1) Algorithm 2 searches the independent traces in a *breadth-first random search* order. We observe it outperforms depth-first or pure random search in improving the diversity of mutants according to our preliminary study. (2) To avoid path explosion, one independent trace selects the same self-loop event *at most twice* and *at most three* independent views from the same group view. (3) The transitional model removes redundant events between any two abstract states to reduce the search space. Considering the generated mutants are not guaranteedly replayable, we detect unreplayable mutants at runtime, and use their trace prefixes to skip similar unreplayable mutants to save testing time.

4.3 Duplicated errors and false positives reduction

In practice, GENIE may report many of suspicious non-crashing functional errors, most of which are actually duplicated or false positives. To reduce *duplicated errors*, we implemented a two-step reduction algorithm. First, GENIE converts each suspicious functional error into a string, which encodes the witness of oracle violation, *i.e.*, the differences of GUI views between the seed and its mutant. By string equality comparison, we remove any redundant functional errors, and only keep distinct ones. This step *will not* incur any false negatives. Second, inspired by Engler *et al.*'s "*bugs as deviant behavior*" idea [Engler et al. 2001], GENIE ranks these distinct errors according to their occurrences. The basic idea is that suspicious errors with fewer occurrences are more likely to be true errors. Specifically, when GENIE identifies distinct errors, it counts the occurrences of them, and then ranks these errors by their occurrences. In practice, GENIE only keeps the mutants with *1-occurrence* distinct errors (of course, this may unavoidably incur false negatives).

On the other hand, we observed that many trivial false positives were caused by the app features and the state abstraction of GUI model. To this end, we adopted two strategies to remove these trivial false positives. (1) We replay a seed test twice to identify those GUI views which will automatically change by themselves (*e.g.*, time-sensitive text labels, dynamic or non-deterministic app behaviors), and exclude these GUI views from oracle checking. (2) We use a heuristic to decide whether an independent event trace indeed returns back to the pivot layout — if the pivot layout and the layout of ending state of the independent trace have more than 50% differences in the texts, we assume these two layouts are likely different. In other words, this independent event trace fails to return back to the pivot layout and the corresponding mutant will be excluded. Section 5.2 shows the reduction effectiveness and Section 5.4 discusses the remaining false positives.

We also implemented a bug report visualization tool to ease manual inspection by highlighting critical GUI inconsistencies. It automatically aligns the seed test and its mutants side-by-side, and annotates any critical GUI inconsistencies between different groups of GUI pages.

4.4 Challenges, Trade-offs and Limitations

Due to the diversity and complexity of apps in practice, we have to balance between generalizability and precision of our approach. For example, specific app features or GUI designs may affect the precision of inferring independent views and oracle checking. Thus, GENIE adopts some trade-offs when implementing `Equivalent_with`, `Independent_from`, `Similar_with`, `Connect_with`, `Search_traces` and `Similar_layout_type`. These trade-offs may lead to imprecise results, and thus incur false positives or negatives. The false positive analysis in Section 5.4 also reflects the impacts of these trade-offs. Additionally, GENIE's effectiveness is also limited by the seed test quality

Table 1. Challenges faced by GENIE, the strategies/heuristics adopted by GENIE to tackle the corresponding challenges, and related algorithm pseudocode and functions.

Challenges	Strategies/Heuristics	Pseudocode	Related Functions
Inferring independent views	group view-based inference (Section 3.3)	Algorithm 1	`Independent_from`, `Similar_with`, `Similar_layout_type`
Generate independent event traces (trace space explosion)	model-based independent event trace query (Section 3.4, 4.2)	Algorithm 2	`Equivalent_with`, `Connect_with`, `Search_traces`
Precise identification of property violation	GUI tree-based oracle computation (Section 3.5, 4.3)	Algorithm 3	`Similar_layout_type`
High-quality, meaningful seed tests	random seed generation with motif-events (Section 4.1)	-	-

and diversity of inserted traces. Section 5.7 discusses the quality of seed and mutant tests generated by GENIE in finding non-crashing bugs. Table 1 summarizes the challenges faced by GENIE, the strategies or heuristics adopted by GENIE to tackle the corresponding challenges, and the related algorithm pseudocode and functions.

5 EVALUATION

Our evaluation aims to answer four research questions:

- **RQ1 (Bug Finding)**: Can GENIE automatically find non-crashing functional bugs in real apps? Can existing fully automated GUI testing tools find such non-crashing bugs?
- **RQ2 (Code Coverage)**: Can GENIE improve code coverage via functional fuzzing?
- **RQ3 (Oracle Precision)**: How is GENIE's precision in finding non-crashing functional bugs?
- **RQ4 (Bug Types)**: What types and characteristics of non-crashing bugs can GENIE find?

5.1 Evaluation Setup

Evaluation Environment. GENIE runs on a 64-bit Ubuntu 18.04 machine (64 cores, AMD 2990WX CPU, and 128GB RAM) to do functional fuzzing. It runs tests on Android emulators. Each emulator is configured as a Google Nexus 7 device with 2GB RAM, X86 ABI image (KVM powered), and the Marshmallow version (SDK 6.0, API level 23). Different types of external files (including 2 PNGs/2 MP3s/2 PDFs/2 TXTs/1 DOCX) are stored in the SDCard to facilitate file access from apps.

App Subjects. We chose open-source apps in the wild as the subjects to evaluate GENIE because we can directly communicate with app developers for bug confirmation. Specifically, we scrawled all the apps from F-Droid [F-Droid 2020], the largest open-source Android app market, and also collected the subjects from recent research on GUI testing of Android [Choudhary et al. 2015; Fazzini et al. 2018; Kowalczyk et al. 2018; Lu et al. 2019]. Among these apps, we selected the apps on GitHub, by which we can submit bug reports. We obtained 1,752 candidates, and further filtered apps following the following criteria: (1) *Real-world apps* — The app is officially released on Google Play [Google Play 2020] for users to download and install — 807 candidates remained; (2) *Popular apps* — The app has 10,000+ installations on Google Play *and* 100+ stars on GitHub — 348 apps remained; (3) *Well-maintained* — The app is active and has been maintained for a long time (*e.g.*, two years) since its first release; we assume such apps are usually stable and well-tested by their developers (and users) — 115 candidates remained; and (4) *Self-contained apps* — Since Android apps are diverse, for an objective evaluation of GENIE, we focus on "self-contained" apps, whose functionalities can be automatically explored without specific human inputs or participation, because GENIE uses random seed tests. Thus, we filtered apps that require valid user account (*e.g.*, email/social media

Table 2. App subjects used in our evaluation. #L. #Br., #M., #Cl. represent the numbers of code lines, branches, methods and classes, respectively (K=1,000, M=1,000,000).

App Name	Feature Description	Version	First Release	#Google Play Installations	#GitHub Stars	#L.	#Br.	#M.	#Cl.
ActivityDiary	Personal Diary	1.4.0	Oct. 2017	1K~5K	56	4,238	1,169	611	157
Transistor	Radio Program Player	3.2.2	Oct. 2015	10K~50K	254	3,024	1,003	551	137
Tasks	Task Manager & Reminder	6.6.5	Dec. 2008	100K~500K	1.3K	31,757	8,386	6,187	984
UnitConverter	Unit Converter	5.5.1	Jun. 2014	1M~5M	144	1,525	304	300	88
RadioDroid	Radio Streaming Player	0.79	Jan. 2013	50K~100K	323	8,085	2,397	1,426	309
SkyTube	YouTube Video Player	2.97	Nov. 2015	100K~500K	843	8,469	2,531	1,889	318
SimpleTask	Cloudless To-do Lists	10.3.0	Mar. 2013	10K~50K	390	2,512	1,022	403	83
Fosdem	Conference Scheduler Browser	1.6.2	Dec. 2013	10K~50K	111	5,563	1,419	1,106	290
Markor	Markdown Note Editor	2.0.1	Dec. 2014	50K~100K	1.2K	8,149	3,801	1,516	232
AnkiDroid	Advanced Flashcard Learning	2.9.6	Jun. 2009	5M~10M	3K	29,548	11,047	4,157	543
AnyMemo	Language & Vocabulary Learning	10.11.4	Jan. 2010	100K~500K	122	13,003	3,060	2,809	649
MyExpense	Personal Financial Assistant	3.0.9.1	Mar. 2011	500K~1M	278	34,549	13,466	5,935	836

clients), interact with other devices (*e.g.*, messaging and IoT apps), require specific human inputs or gestures (*e.g.*, navigation and gaming apps). At the end, we obtained 26 apps that meet these criteria. *Note that, in principle, GENIE is not limited to these apps if provided with proper information (e.g., valid user account or text inputs), which is an orthogonal issue for any automated testing tools.*

To further ensure the diversity of apps, we classified these 26 apps plus the app *ActivityDiary* into four different categories according to their main features (*e.g.*, personal management, multimedia), and then randomly selected 3 apps from each category. Thus, we focus on the 12 selected apps (including *ActivityDiary*). Table 2 gives the details of these subjects. *Feature Description* briefs their functionalities. *Version* gives the versions of their *latest releases* on Google Play at the time of our study, which our evaluation targets. *First Release* gives the time of their first releases. *#L., #Br., #M.* and *#Cl.* give their numbers of code lines, branches, methods and classes, respectively.

Evaluation Setup. GENIE was configured with 1 hour for mining the GUI transitional model and 20 random seed tests (each with up to 15 events) for fuzzing. At each insertion position, GENIE generates up to 300 mutants (one seed test has up to 15 insertion positions); and each inserted event trace contains up to 8 events. To scale the testing, GENIE runs mutants on 16 emulators (the maximum number of emulators allowed by Android system) in parallel on one machine. Under these settings, it took around 25~35 machine hours to finish testing one app (including the four core steps and bug report reduction). The whole evaluation took roughly 18 machine days in total for the twelve app subjects. Note that the fuzzing cost can be significantly reduced by using more machines, Android devices, and parallel threads. In addition, we selected two state-of-the-art/practice fully automated GUI testing tools for Android, MONKEY [Monkey 2020] and APE [Gu et al. 2019], to compare them with GENIE and verify whether they can find non-crashing bugs. We allocated 30 hours for the two tools to test each of the twelve app subjects, and inspected the found bugs.

Table 3. Bugs that were found by GENIE in the evaluated app subjects.

Bug Type	#Reported	#Confirmed	#Fixed
#Non-crashing Bugs	34	34	22
#Crash Bugs	26	24	14
#Total	60	58	36

Table 4. Comparison between GENIE, MONKEY and APE on detecting non-crashing bugs.

Bug Type	GENIE	MONKEY	APE
#Non-crashing Bugs	**34**	**0**	**0**
#Crash Bugs	26	25	36
#Total	60	25	36

Table 5. Detailed statistics of functional fuzzing on the app subjects. D. E. (Distinct Error), 1-O. D. E. (1-Occurrence Distinct Error), T. P. (True Positive).

App Name	#Generated Mutants	#Executed Mutants	#Error Mutants	#D. E. Mutants	#1-O. D. E. Mutants	#1-O. D. E. Mutants (Refined)	#T. P. Mutants	#Distinct Non-crashing Bugs	#Distinct Crash Bugs
ActivityDiary	45,258	27,433	13,822	1,264	249	69	31	7	3
Transistor	84,029	6,740	2,497	194	46	16	12	5	0
Tasks	37,339	33,786	7,367	396	87	17	12	2	2
UnitConverter	73,203	21,511	14,864	4,015	229	106	50	2	1
RadioDroid	50,421	28,954	10,538	819	95	62	19	6	5
SkyTube	73,853	12,525	7,810	2,584	558	58	27	1	8
SimpleTask	59,081	13,526	3,277	363	75	16	13	1	0
Fosdem	36,594	31,990	6,243	845	192	71	30	1	1
Markor	57,065	20,516	7,978	665	132	73	33	4	1
AnkiDroid	59,740	38,475	3,759	449	58	23	5	1	2
AnyMemo	39,288	19,552	5,255	448	95	83	22	3	3
MyExpense	68,565	40,925	9,262	2,815	378	41	6	1	0
Total	684,436	295,934	92,672	14,857	2,194	635	260	34	26

5.2 RQ1: Bug Finding

Table 3 shows the statistics of bugs found by GENIE. GENIE found 34 non-crashing functional bugs, all of which were confirmed and 22 were already fixed. As a by-product of our testing, 26 crash bugs were also discovered with 24 confirmed and 14 fixed. In total, GENIE found 60 bugs with 58 confirmed and 36 fixed. All these bugs were discovered in the latest app releases at the time of our study and previously unknown. Section 5.5 discusses the types of the 34 non-crashing bugs.

Table 4 shows the comparison results. We can see that MONKEY and APE *cannot* detect any of the 34 functional bugs because they are limited to finding crash bugs. Specifically, we find GENIE and MONKEY have no common crash bugs, while GENIE and APE have 6 common crash bugs. These results clearly show GENIE complements existing fully automated testing for crash bugs.

Table 5 gives the detailed statistics. *#Generated Mutants* and *#Executed Mutants* denote the total number of generated and executed mutants, respectively. *#Error Mutants*, *#D. E. Mutants*, and *#1-O. D. E. Mutants* denote the total number of error mutants (*i.e.*, suspicious functional errors), the distinct error mutants (including 1-occurrence, 2-occurrence, 3-occurrence, *etc*), the 1-occurrence distinct error mutants (only appear once), respectively. *#1-O. D. E. Mutants (refined)* denotes the number of 1-occurrence distinct error mutants after automatically removing trivial false positives (discussed in Section 4.3), which are the error mutants for manual inspection. *#T. P. Mutants* denotes the total number of true positives (*i.e.*, real functional bugs). *#Distinct Non-crashing Bugs* and *#Distinct Crash Bugs* denote the number of distinct non-crashing bugs and crash bugs, respectively.

Table 5 shows GENIE is able to discover non-crashing functional bugs in all the subjects. Specifically, for several highly popular apps, *i.e.*, *Tasks*, *UnitConverter*, *Skytube*, *Markor* and *AnkiDroid*, although they have many developer-written tests and oracles (released with their source code) and large user installations (1M~10M)/high number of stars (843~3K), GENIE is still able to find 24 bugs (10 non-crashing bugs and 14 crash bugs) in them. It shows GENIE's effectiveness.

From Table 5, we can see 295,934 out of 684,436 (43.2%) mutants were executable. The reason is that GUI tests are event order sensitive. Any mutation-based testing technique like ours may generate unreplayable tests. Because the inserted event traces may lead to the GUI states that cannot connect with subsequent events. For example, *Transistor* has many unreplayable mutants

Table 6. Code coverage of seed tests and their corresponding mutant tests generated by GENIE for the 12 app subjects. Line, Branch, Method represent line, branch and method coverage, respectively. The numbers in the brackets represent the coverage improvement of the mutant tests *w.r.t.* the seed tests in terms of line, branch and method coverage, respectively.

App Name	Line (%) (seeds)	Line (%) (mutants)	Branch (%) (seeds)	Branch (%) (mutants)	Method (%) (seeds)	Method (%) (mutants)
ActivityDiary	45.9	65.0 (+19.1)	33.8	48.2 (+14.4)	51.7	70.0 (+18.3)
Transistor	58.5	63.1 (+4.6)	39.1	43.9 (+4.8)	61.0	65.5 (+4.5)
Tasks	28.6	33.9 (+5.3)	16.8	23.3 (+6.5)	36.4	40.9 (+4.5)
UnitConverter	56.9	71.6 (+14.7)	32.5	61.5 (+29)	63.3	73.7 (+10.4)
RadioDroid	46.6	53.2 (+6.6)	31.9	40.0 (+8.1)	48.8	55.2 (+6.4)
SkyTube	44.7	51.9 (+7.2)	30.5	37.3 (+6.8)	50.0	55.9 (+5.9)
SimpleTask	31.3	33.7 (+2.4)	15.3	19.3 (+4.0)	31.0	34.7 (+3.7)
Fosdem	60.0	62.0 (+2.0)	39.7	44.4 (+4.7)	69.0	70.3 (+1.3)
Markor	49.7	57.7 (+8.0)	34.6	43.8 (+9.2)	54.4	60.0 (+5.6)
AnkiDroid	32.7	42.1 (+9.4)	20.8	28.0 (+7.2)	41.0	51.4 (+10.4)
AnyMemo	49.3	53.5 (+4.2)	32.1	38.7 (+6.6)	55.6	58.3 (+2.7)
MyExpense	25.1	29.3 (+3.5)	15.6	19.2 (+3.6)	32.3	36.6 (+4.3)
Average	**44.1**	**53.0 (+8.9)**	**28.6**	**38.6 (+10.0)**	**49.5**	**56.0 (+6.5)**

due to this reason. Additionally, we can see GENIE's duplicated errors reduction is effective, which reduces 92,672 error mutants to 14,857 distinct ones (6.2X reduction rate); by ranking these distinct error mutants and automatically reducing trivial false positives, GENIE further constrains our focus on 635 mutants for manual inspection. Section 5.4 will discuss the false positives and show that the manual inspection cost is acceptable.

5.3 RQ2: Code Coverage

Table 6 shows the code coverage of all subjects achieved by the 20 random seed tests and their corresponding mutant tests, respectively. The seed tests and the mutant tests are all generated by GENIE. The main purpose of this measurement is to investigate whether GENIE can improve code coverage via functional fuzzing. Here, the code coverage data is measured by Jacoco [Jacoco Team 2020] at runtime. For example, for *ActivityDiary*, the mutant tests generated by GENIE achieves 19.1% and 14.4% higher line and branch coverage than the seed tests, respectively. We can see that, GENIE on average improved line coverage by 8.9%, branch coverage by 10.0%, and method coverage by 6.5% by generating the mutant tests from the seed tests. This coverage improvement is significant, considering achieving high code coverage of Android apps is challenging [Zheng et al. 2017]. These results also explain GENIE's effectiveness in finding functional bugs. For example, in 8 apps, the improvements of branch coverage are even higher than those of line coverage. The improvement of branch coverage indicates the control-flows or even data-flows were more stress-tested by the mutants than the seeds.
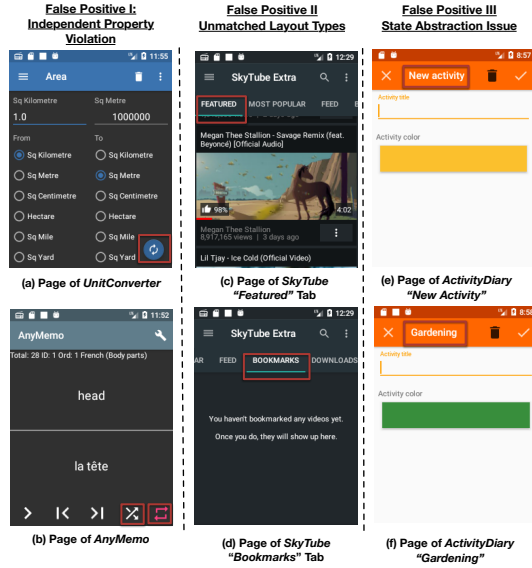
5.4 RQ3: Oracle Precision

As shown in Table 5, GENIE may report false positives (FPs). To this end, we carefully examined all the reported FPs to identify the root causes and understand the oracle precision (and other possible issues). Table 7 gives the detailed FP analysis results. *#F. P. Mutants* denotes the total number of FP mutants (computed by *#1-O. D. E. Mutants (refined)* and *#T. P. Mutants* in Table 5). Specifically, we identified three classes of FPs.

One class of FPs is related to the violation of independent view property. Some FPs (see *#I* in Table 7) appear because the inserted event traces do not satisfy the independent view property. The main reason is that GENIE only leverages GUI layouts to automatically infer independent views, which

Fig. 4. Examples of different false positive types. The red boxes annotate the root causes.

Table 7. Detailed analysis results on false positives (FPs).

App Name	#F. P. Mutants	#I	#L	#S
ActivityDiary	38	31	5	2
Transistor	4	4	0	0
Tasks	5	5	0	0
UnitConverter	56	50	6	0
RadioDroid	43	8	10	25
SkyTube	31	16	15	0
SimpleTask	3	2	1	0
Fosdem	41	0	22	19
Markor	40	7	5	28
AnkiDroid	18	16	0	2
AnyMemo	61	34	12	15
MyExpense	35	0	0	35
Total	**375**	**173**	**76**	**126**

(a) Page of *UnitConverter*

(b) Page of *AnyMemo*

(c) Page of *SkyTube* "*Featured*" Tab

(d) Page of *SkyTube* "*Bookmarks*" Tab

(e) Page of *ActivityDiary* "*New Activity*"

(f) Page of *ActivityDiary* "*Gardening*"

sometimes may not be precise. Fig. 4 (a) shows an example of *UnitConverter*. GENIE infers the FAB button (annotated by a red box) is independent from other views, and thus generates a number of independent event traces that start with a click on this button. However, clicking this button will exchange two units between the columns "From" and "To", which breaks the independent property and affects subsequent events. Fig. 4 (b) shows another example of *AnyMemo*, where GENIE assumes the *shuffle* button (the left red box) and the *repeat* button (the right red box) are independent within their group view. However, they affect each other when both are enabled.

The other two classes of FPs are caused by some open technical challenges. Some FPs (see *#L*) appear because GENIE sometimes fails to precisely infer layout types. GENIE takes two comparable GUI layouts (with similar functionality) and computes GUI effects. However, automatically inferring layout types is difficult. For example, GENIE cannot differ *SkyTube*'s page "Featured" (see Fig. 4 (c)) and page "Bookmarks" (see Fig. 4 (d)), although we humans can easily tell they are different, which should not be compared. *Fosdem* and *RadioDroid* have similar FPs.

Some FPs (see *#S*) are caused by *model state abstraction*. GENIE queries independent event traces via the abstract GUI model. This model allows us to generate more interesting mutants because similar states are abstracted to form a lot of loop traces. However, this may incur invalid independent traces in practice, *i.e.*, the layout of ending state of an independent trace and the pivot layout cannot form a *connected, loop trace* at concrete state level. For example, in *ActivityDiary*, an invalid trace may start from page Fig. 4 (e) and ends at page Fig. 4 (f) because these two pages are similar at the abstraction level (because all the texts are ignored in the abstracted model). The second heuristic we used in Section 4.3 to reduce false positives cannot exclude this invalid event trace because these two pages are only different at the page title ("New Activity" *v.s.* "Gardening"). In fact, eliminating such cases depends on the precision of inferring layout types at the concrete state level.

Discussions. According to Table 7, GENIE's true positive rate is 40.9% (260/635). It is worth noting that this is the result in a very challenging problem setting (*i.e.*, finding non-crashing functional bugs without any human-provided tests and oracles). Existing fully automated tools cannot find
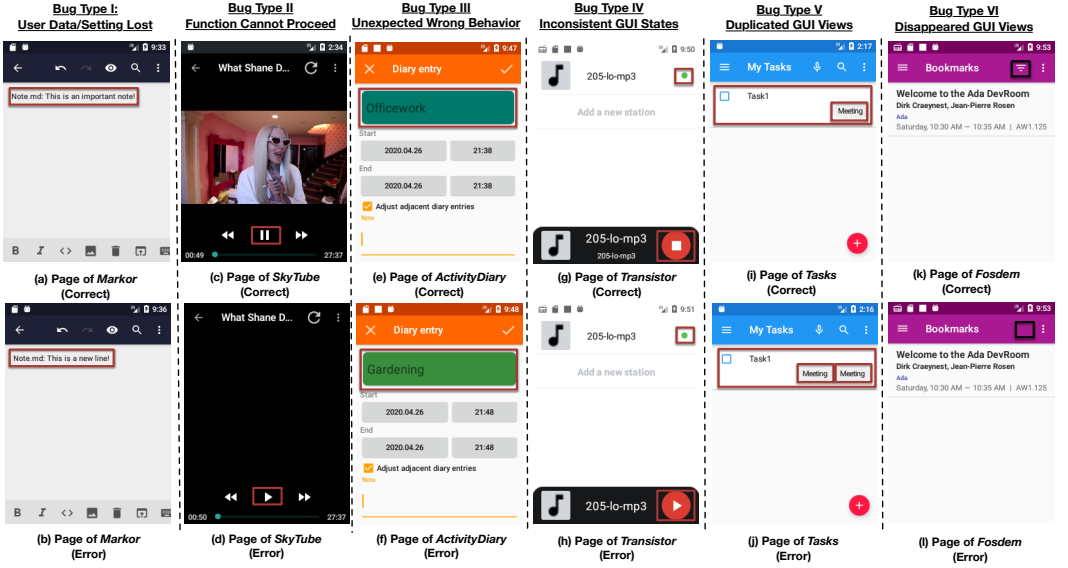
Fig. 5. Examples of different functional bug types. In each group, the page at top shows the correct behavior, while the page at bottom shows the erroneous behavior, and the red boxes indicate the clues of each issue.

the 34 non-crashing bugs. To put things into perspective, high-impact, well-tuned industrial bug finding tools like Facebook's INFER was found to have a 26.3% true positive rate for detecting null pointer dereferences, an extensively studied problem which Infer targets, by a recent study [Wang et al. 2020]. Also, the FP analysis shows some FPs could be further automatically removed with light developer input (*e.g.*, providing information on layout types, independent views and tab types). Thus, the current result is already strong for the problem setting. On the other hand, the benefits of GENIE in finding non-crashing bugs can outweigh the manual cost of inspecting these FPs. First, manual validation (*e.g.*, writing tests and oracles) is the only alternative to discover these bugs, but is more costly, small-scale, and often ineffective in finding bugs. For example, 8 out of the 12 apps in fact have manually-written tests and oracles, which did not reveal these bugs however. Second, our experience shows that manually checking one suspicious mutant takes roughly half a minute. Thus, the manual cost is less than one hour for each app (see *#1-O. D. E. Mutants (refined)* in Table 5). This is a low, affordable cost since it is much more expensive to manually write and maintain GUI tests. Moreover, in the context of classic static analysis, to check whether an alert is indeed a bug or not, one typically has to look at the code and its context and try to decide, which is tough. While in our context, one only needs to look at the GUI pages on the test executions, which is much easier.

5.5 RQ4: Bug Types and Assorted Samples

The non-crashing functional bugs found by GENIE are diverse. According to the bug manifestations and consequences, we are able to categorize the 34 non-crashing bugs into six main types (summarized in Table 8). We illustrate these six bug types with some assorted bug samples.

T1: User data/setting lost. This bug type leads to user data or setting lost. Sometimes, this bug type can bring severe consequences and critical user complaints. For example, *Markor*, a markdown note editor, ever had a data loss issue. One user critically complains "*Markor just went from hero to zero, in my books, as a file I had been working on just disappeared, without a trace! [...] I don't feel like I can currently trust markor with anything important, like the file I was just working on.*". GENIE

Table 8. Different types of non-crashing functional bugs found by Genie.

Type ID	Bug Type	#Bugs
T1	user data/setting lost	4
T2	function cannot proceed	7
T3	unexpected wrong behavior	7
T4	inconsistent GUI states	6
T5	incorrectly displayed GUI views	7
T6	GUI design issue	3

uncovered a data loss issue in *Markor*, a markdown note editor. The original content of a note file will be silently overwritten if the user creates another note file with the same name. Fig. 5(a) and Fig. 5(b), respectively, show the main page where a user can create two notes with the same name and the latter note's content overwrites that of the former one. The developer quickly confirmed and fixed this bug.

T2: Function cannot proceed. This bug type means one specific app function that works well before suddenly cannot proceed anymore and loses effect. Genie found such an issue in *SkyTube*, a YouTube video player. Originally, a user can arbitrarily pause and play any video at his/her will. However, if the user opens the menu, selects an option that plays the video in a browser, and returns back, he/she cannot play the video anymore — clicking the play button cannot load the video; the play button cannot change to a pause button as well. After we reported this issue, the developer quickly confirmed and fixed it with positive feedback "*Thanks for the excellent bug report, it was very helpful to fix the bug!*".

T3: Unexpected wrong behavior. This bug type means the specific functionality shows wrong behavior *w.r.t.* its previous correct behavior, *e.g.*, an incorrect GUI page was erroneously opened, a deleted GUI view was suddenly restored. Genie found such an issue in *ActivityDiary*. Fig. 5(e) shows the history detail page of *Officework*, which was correctly opened after the user (1) starts *Officework* from the main page, and (2) clicks its statistics region. However, if the user does three additional actions between (1) and (2) — rename *Officework* to another arbitrary name (say *Shopping*); start another activity (say *Gardening*); and undo *Gardening* (*i.e.*, return back to *Shopping*). *ActivityDiary* will erroneously open the history page of *Gardening* rather than that of *Shopping* (show in Fig. 5(f)). If at this time the user modifies the history data of *Gardening*, it will break data consistency. The developer commented "*(This is) definitely a bug, but I have to dig deeper in the code to get an idea how could it go wrong*".

T4: Inconsistent GUI states. This bug type means the GUI states are inconsistent for specific functionality, which counters users' intuition on app function. Genie found such an issue in *Transistor*, a radio program player. In the normal scenario, Fig. 5(g) shows the page where a radio program is playing, and meanwhile a green playing indicator is shown near the program name. When the user stops the program, this green indicator will disappear. However, if the user renames the program when it is playing, and then stops it. The playing button will correctly change to a pause button, but the green playing indicator keeps there and indicates the program is playing (see Fig. 5(h)). This issue confuses users' intuition on the play-stop function. In fact, at the same time, the new program name will get lost. The developer was quite annoyed by this issue and commented "*This [...] related to the way Transistor handles stations internally. The root (cause) is so deep in the app's architecture, [...], it needs to wait for the big Kotlin re-write. Without a rewrite this would end in a bunch of nasty hacks.*". Finally, the developer re-wrote the architecture, and fixed this issue.

T5: Incorrectly displayed GUI views. Some GUI views are erroneously duplicated. Genie found such an issue in *Tasks*, a task manager & reminder. Fig. 5(i) shows its main page when *Task 1* was successfully created with a tag *Meeting*. However, when this task was additionally inserted with
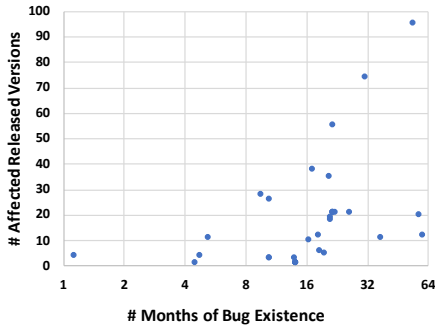
Fig. 6. Statistics of 34 confirmed functional bugs: (1) #months they have resided (x-axis, in logarithmic scale); (2) #affected releases (y-axis), before they were uncovered.

Table 9. Quality of seed and mutant tests generated by Genie. 1-O. D. E. (1-Occurrence Distinct Error), T. P. (True Positive), Insert. Pos. (Insert Position).

App Name	#1-O. D. E. Seeds (refined) (A)	#T. P. Seeds (B)	B/A	T. P. Insert. Pos. (%)
ActivityDiary	15	8	53.3%	30.4%
Transistor	13	5	38.5%	73.3%
Tasks	10	7	70.0%	45.7%
UnitConverter	8	2	25.0%	35.0%
RadioDroid	11	5	45.0%	63.9%
SkyTube	15	2	13.3%	23.0%
SimpleTask	7	1	14.3%	25.0%
Fosdem	9	1	11.1%	60.0%
Markor	11	4	36.6%	52.5%
AnkiDroid	14	1	7.1%	20.4%
AnyMemo	12	4	33.3%	48.7%
MyExpense	15	2	13.3%	18.6%
Total	140	42	-	-

two attachments (*e.g.*, two pictures). *Task 1* will be erroneously associated with two duplicated *Meeting* tags (see Fig. 5(j)).

Some GUI views inadvertently disappear. Genie found such an issue in *Fosdem*, a conference schedule browser. Fig. 5(k) shows the correct *Bookmarks* page. However, if the user (1) opens the menu (the three dots on the top right corner), (2) does two consecutive selections on the option "Search events", and (3) presses back, the filter option on the tool bar inadvertently disappears (Fig. 5(l)). Only restarting the app or reopening the page can recover this option view.

Some views are incorrectly displayed. For example, Genie found such an issue in *RadioDroid*. Due to this issue, all page titles are displayed as "Setting" if the user changes an irrelevant setting option. This option only alters the location of navigation menus between the page bottom and the navigation drawer.

T6: GUI design issues. This bug type is related to the issue of functional GUI design, which could confuse users and disrupt user experience. For example, Genie found such an issue in *RadioDroid*. The issue manifests itself as the *history* page fails to record some ever played radio programs. After the discussion with developers, we know that *RadioDroid* maintains two types of history data, *i.e.*, *history* and *track history*. The *track history* only records those radio programs with track information. However, *RadioDroid* has never clearly differed these two types of history data on its GUIs (use the word *history* for both functional pages). Finally, developers confirmed this issue and reworked the GUI design to avoid user confusion.

5.6 Discussion

The non-crashing functional bugs found by Genie are nontrivial and long latent. Fig. 6 characterizes the 34 non-crashing bugs. Each point denotes one bug, which is characterized by (1) #months it has resided in the app and (2) #affected releases on Google Play before uncovered by Genie (some points are overlapped). We can see 26 out of the 34 bugs (78.8%) were unnoticed for more than one year, and 19 out of the 34 bugs (57.6%) affected more than ten releases. For example, Genie uncovered a user setting lost issue in *UnitConverter* (1M~5M installations), which leads to an incorrect group separator shown in the converted values. This issue was unnoticed for more than four years and its manifestation requires a minimal 10-event GUI test. Genie also found an inconsistent GUI states issue in *Markor* (1.2K GitHub Stars), which leads to Undo and Redo behave inconsistently when editing different lengths of texts. This issue was introduced by another bug fix three years ago, and resides in 74 releases. Now both bugs were fixed. These results show many

found bugs are non-trivial and long latent. Also, we received positive feedback from developers, *e.g.*, "*This is an outstanding report, thank you so much!*", "*Excellent bug report!*", "*Thanks for the excellent bug report, it was very helpful to fix the bug!*".

Quality of seed/mutant tests generated by Genie. We further examined the quality of seed and mutant tests generated by Genie. Table 9 gives the detailed analysis on the quality of seeds/mutants from Genie. *#1-O. D. E. Seeds (refined)* and *#T. P. Seeds* denote the number of seed tests that can generate 1-occurrence distinct error mutants (refined) and true positive mutants, respectively. Thus, *B/A* indicates how many seed tests (in percentage) can generate true positive mutants. We can see, on average, 30.0% (=42/140) of seed tests can be successfully mutated to uncover non-crashing bugs. In Table 9, *T. P. Insert. Pos.* denotes, among those seed tests that can generate true positive mutants, on average how many insertion positions (in percentage) inserted with independent event traces can successfully create true positive mutants. We can see the success rate ranges from 18.6~73.3% on these app subjects. Thus, these results show Genie has reasonable mutation ability to create true positive mutants at both seed test and insertion position levels without any human-provided tests. We also observe Genie's bug detection ability could be further improved by feeding high-quality seed tests that cover more meaningful app functionalities. For example, Genie's random seed generation produces only one meaningful seed test for finding the bugs in *SimpleTask*, *Fosdem* and *AnkiDroid*, respectively.

Threats to validity. Our work may suffer from some threats to validity. First, our technique is only evaluated on 12 apps. To counter this, we select these apps with different features to ensure the generalizability of our evaluation. Table 2 shows these apps are diverse. Second, the false positive analysis is manually conducted which may incur imprecision. To counter this, three co-authors of this paper participate in the analysis and do cross-checking to ensure the correctness. In the future, we will evaluate more apps.

5.7 Open Challenges

Automatically validating functional correctness of Android apps without explicit oracles is very challenging. Our technique is the first to tackle this problem via independent view fuzzing. We face some open challenges. First, although the independent view property captures a substantial class of functional bugs, it cannot detect all types of non-crashing functional bugs. In principle, Genie can only find those non-crashing functional bugs that lead to GUI-level inconsistencies by comparing a seed test and its corresponding mutant tests from the same app version. Thus, Genie cannot detect the non-crashing regression functional bugs (*i.e.*, the app functionality does not work in the current tested app version, but worked in some prior app versions). One way is to explore other invariant properties of apps or testing techniques (*e.g.*, using differential testing to detect those regression bugs). In addition, Genie cannot detect just-failing functional bugs (*e.g.*, the back-end server is down, so the app cannot function) or those ad-hoc GUI issues (*e.g.*, color contrast/brightness, text trucked/misalignment issues).

Second, our technique only uses random seed tests for functional validation. Such seed tests are easy to obtain, but may be unable to cover complicated app functionalities that require human participation. This constrains the current implementation of Genie to test "self-contained" apps (as we discussed in the evaluation setup). One solution is to leverage human-provided tests to cover more complicated functionalities. In the future, we will also explore the possibility of synergistically combining human knowledge with Genie to further improve the testing effectiveness and precision. We believe this is a promising direction to explore.

6 RELATED WORK

Finding crash bugs of Android apps. Many *fully automated* GUI testing techniques have been proposed in the literature [Kong et al. 2019; Su et al. 2021; Tramontana et al. 2019], including random testing [Hu et al. 2014; Machiry et al. 2013; Monkey 2020], evolutionary algorithms [Mahmood et al. 2014; Mao et al. 2016], symbolic execution [Anand et al. 2012; Mirzaei et al. 2012; van der Merwe et al. 2012], model-based testing [Amalfitano et al. 2015; Choi et al. 2013; Gu et al. 2019; Su et al. 2017; Yang et al. 2013], systematic exploration [Azim and Neamtiu 2013; Dong et al. 2020; Mirzaei et al. 2016; Song et al. 2017], and combinations thereof [Fan et al. 2018a; Jensen et al. 2013]. But all of them can only detect crashes because they use *app exceptions* as the implicit test oracle [Fan et al. 2018b; Su et al. 2020]. They cannot fully automatically uncover non-crashing functional bugs without human-provided test oracles.

Finding non-crashing functional bugs of Android apps. Finding non-crashing functional bugs depends on oracles. The current practices heavily rely on manual validation, which incurs substantial cost. On the other hand, the current research can be classified into three categories according to the source of oracles. The first category leverages human-provided oracles to find non-crashing bugs in different strategies. These oracles are usually encoded in assertions (*e.g.*, THOR [Adamsen et al. 2015], CHIMPCHECK [Lam et al. 2017], APPFLOW [Hu et al. 2018], ACAT [Rosenfeld et al. 2018], AppTestMigrator [Behrang and Orso 2019], CraftDroid [Lin et al. 2019]), linear-time temporal logic (LTL) formulas (*e.g.*, FARLEAD-Android [Köroglu and Sen 2019]), or semantic models in Alloy (*e.g.*, AUGUSTO [Mariani et al. 2018] which targets web applications). One special case in this category is the oracles derived from human-written app specifications. For example, Memon *et al.* [Memon et al. 2003, 2000; Xie and Memon 2007] derives oracles (*i.e.*, the expected GUI state) for desktop GUI applications from an initial GUI state of a GUI test based on human-provided GUI specifications. RENALART [Malik et al. 2021] derives oracles from app specifications written in natural language. Different from these work, GENIE do not require direct or indirect human-provided oracles.

The second category uses differential testing to overcome the oracle problem. For example, SPAG-C [Lin et al. 2014] and DIFFDROID [Fazzini and Orso 2017] compare the GUI images or screenshots (explored by the same GUI test) between two different mobile devices or platform versions to identify GUI issues. However, they can only detect device- or platform-specific bugs.

The third category generates automated oracles for a specific class of user interactions. For example, QUANTUM [Zaeem et al. 2014] derives the automated oracles only for app-agnostic interaction features. QUANTUM assumes an app's GUI page should not change after double-screen-rotations, killing-and-restarting or pausing-and-resuming (THOR [Adamsen et al. 2015] injects similar "neutral" events into existing developer tests to do functional testing); or should keep partial GUI elements after zooming-in, zooming out or scrolling. Due to QUANTUM [Zaeem et al. 2014] (also THOR [Adamsen et al. 2015]) can only validate specific app-agnostic features, it cannot detect the 34 non-crashing bugs found by GENIE because all these bugs are related to app-specific features. SetDroid [Sun et al. 2021] can generate automated test oracles for system setting-related, non-crashing defects of Android apps. But it cannot detect the 34 non-crashing bugs found by GENIE because all these bugs are not related to system settings. ACETON [Jabbarvand et al. 2020] uses deep learning to automatically construct test oracles for detecting energy bugs (which are non-functional bugs), while GENIE targets functional bugs.

Mutation-based testing for Android apps. Our approach generates mutant tests by inserting independent event traces, which is quite different from prior mutation-based testing techniques. Sapienz [Mao et al. 2016] shuffles the orders of events between two seed tests to create new tests. EvoDroid [Mahmood et al. 2014] uses evolutionary algorithms to inject, swap or remove events based on given test cases. Specifically, it uses a static UI interface model and program call graphs to

do event dependency analysis to ensure the test connectivity. Quantum [Zaeem et al. 2014] and Thor [Adamsen et al. 2015] injects specific neutral events (*e.g.*, double-screen-rotations, pausing-and-resuming) to create new tests. Amalfitano *et al.* [Amalfitano et al. 2018] inject screen rotations to stress test Android apps. TCM [Köroglu and Sen 2018] uses six different mutation operators (*e.g.*, repeat existing event sequence, remove event delays, change texts) to mutate a seed test. Other tools, like KREfinder [Shan et al. 2016] and APEChecker [Fan et al. 2018a], injects specific events (*e.g.*, app pausing-and-resuming, quickly start and exit of a GUI page) to help reproduce certain types of errors. In contrast, the mutation strategy in Genie is able to insert generic GUI events via querying the GUI transitional model, which is more flexible and general.

Metamorphic Testing. Our approach leverages the idea of metamorphic testing [Chen et al. 1998, 2018; Segura et al. 2016] at the high-level. The key idea of metamorphic testing is to detect violations of domain-specific metamorphic relations by comparing the outputs between a seed test and its corresponding mutant tests. It has been widely used in testing many other specification-less software systems, *e.g.*, compilers [Le et al. 2014; Sun et al. 2016], debuggers [Tolksdorf et al. 2019], and machine learning-based systems [Tian et al. 2018; Xie et al. 2011]. To the best of our knowledge, we are the first to apply the idea of metamorphic testing to automatically detect non-crashing functional bugs, based on our key observation of independent view property.

7 CONCLUSION

We have introduced independent view fuzzing, a novel, fully automated approach to detecting non-crashing functional bugs of Android apps. The key idea is to leverage the independent view property to manufacture property-preserving mutant tests from a set of seed tests. It overcomes the difficulty of functional testing due to lack of test oracles by checking whether the GUI effects of a seed test are preserved in each of its mutants. The realization of our idea, Genie, has successfully discovered 34 non-crashing bugs from twelve real-world apps, all of which have been confirmed and 22 were already fixed. All these bugs were previously unknown and most are long latent. Our work complements and enhances existing manual testing and fully automated testing for crashes.

ACKNOWLEDGMENTS

A APPENDIX

A.1 Prevalence of the Indpendent View Property

To validate the prevalence of the independent view property, we investigated the non-crashing functional bugs of five popular apps, *i.e.*, *Ankidroid* [AnkiDroid Team 2020], *AmazeFileManager* [Amaze-FileManager Team 2020], *AntennaPod* [AntennaPod Team 2020], *k-9mail* [K-9 Mail Team 2020], *TeamNewPipe* [NewPipe Team 2020]. We focused on all their reported bugs within the recent one-year duration (Sep. 2019–Sep. 2020). This investigation took the authors more than one-person month. We detail the process and the results below.

Apps. We selected these five apps since they are popular, diverse and well-maintained. They are released on Google Play and their issue repositories are open on GitHub. *Ankidroid* is an advanced flashcard learning app (3K stars, 5M~10M installations and first released since Jun. 09); *AmazeFileManager* is a file manager app (3.1K stars, 1M~5M installations and first released since Jul. 14); *AntennaPod* is a podcast manager app (3.1K stars, 500K~1M installations and first released

Table 10. Detailed statistics of investigating the prevalence of independent view property for finding non-crashing functional bugs.

App Name	#V	#T	#N	#F	#R	#O	#P (=#O/#F)
Ankidroid	25	215	90	46	28	13	28.3%
AmazeFileManager	10	70	39	19	14	4	21.1%
AntennaPod	7	53	42	23	15	6	26.1%
k-9mail	23	91	67	27	15	9	33.3%
TeamNewPipe	23	78	44	14	9	6	42.8%
Total	88	507	282	129	81	38	29.5%

since Dec. 11); *k-9mail* is an email client app (5.3K stars, 5M~10M installations and first released since Oct. 08); and *TeamNewPipe* is a video streaming app (11.4K stars, 10M~50M installations and first released since Sep. 15).

Investigation process. For each app, we first collected all the bugs reported between Sep. 2019 and Sep. 2020 with the issue labels assigned by developers themselves (*e.g.*, "bug", "Issue-Bug"). We chose one-year duration in order to balance the representativeness of this investigation and the cost of manual analysis. Table 10 shows the number of app release versions (denoted by **#V**) and the number of total bugs (denoted by **#T**) within this one-year duration. We then manually identified and excluded those crash or exception bugs, which are not the focus of this paper. Next, for the remaining non-crashing bugs, we manually read their bug reports to check whether they are valid non-crashing functional bugs. For example, we excluded the non-functional bugs (*e.g.*, performance or energy issues), usability bugs (*e.g.*, accessibility issues, color contrast issues, misaligned GUI views, truncated texts), feature issues (problematic feature design) and compatibility issues (only happened on specific devices). Last, we tried our best to manually reproduce those valid functional bugs by following the reproducing steps in their bug reports if exist. For each reproducible functional bug, we manually analyzed whether it violated the independent view property and could be detected by the independent view fuzzing approach.

Results. Table 10 gives the detailed statistics of this investigation. **#T**, **#N**, **#F**, **#R** and **#O** denote the number of reported bugs in total, non-crashing bugs, valid functional bugs, reproducible functional bugs and the functional bugs that are reproducible and indeed violate the independent view property, respectively. **#P** denotes the percentage of functional bugs that violate the independent view property (=#O/#F). Note that we have to reproduce a functional bug for confirmation because the independent view property is GUI-based (only reading the textual bug reports is not enough for analysis). Many functional bugs cannot be reproduced because their bug reports have not provided clear and enough reproducing information (*e.g.*, steps to reproduce). We can see, among all the non-crashing functional bugs, the percentages of violating independent view property range from 21.1~42.8% individually. Out of 129 non-crashing functional bugs, we identified in these five apps, 38 bugs (29.5%≈38/129) violated the independent view property. Note that these percentages indicate the lower bounds because many functional bugs cannot be analyzed (they are not reproducible). *Thus, it shows validating the independent view property is applicable to find many non-crashing functional bugs.*

Further, we find these five apps contain developer-written regression tests (and oracles), but these tests still miss many non-crashing functional bugs (as we can see in Table 10). Specifically, *AnkiDroid* has 431 unit and 59 GUI tests; *AmazeFileManager* has 211 unit and 37 GUI tests; *AntennaPod* has 197 unit and 55 GUI tests; *k-9mail* has 585 unit tests; and *TeamNewPipe* has 30 unit tests. *Thus, human-written tests and oracles are usually inadequate to detect non-crashing functional bugs.*

REFERENCES

Accessibility. 2020. *Accessibility overview.* Retrieved 2019-11 from https://developer.android.com/guide/topics/ui/accessibility

ActivityDiary. 2020a. *Activity Diary.* Retrieved 2020-5 from https://play.google.com/store/apps/details?id=de.rampro.activitydiary

ActivityDiary. 2020b. *Activity Diary's issue #118.* Retrieved 2020-5 from https://github.com/ramack/ActivityDiary/issues/118

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 (ISSTA)*. 83–93.

ADB. 2020. *Android adb.* Retrieved 2020-5 from https://developer.android.com/studio/command-line/adb

Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59.

Domenico Amalfitano, Vincenzo Riccio, Ana C. R. Paiva, and Anna Rita Fasolino. 2018. Why does the orientation change mess up my Android application? From GUI failures to code faults. *Softw. Test., Verif. Reliab.* 28, 1 (2018).

AmazeFileManager Team. 2020. *AmazeFileManager.* Retrieved 2020-11 from https://github.com/TeamAmaze/AmazeFileManager

Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*. 59.

AnkiDroid Team. 2020. *AnkiDroid.* Retrieved 2020-11 from https://github.com/ankidroid/Anki-Android

AntennaPod Team. 2020. *AntennaPod.* Retrieved 2020-11 from https://github.com/AntennaPod/AntennaPod

Appium. 2020. *Appium: Mobile App Automation Made Awesome.* Retrieved 2020-5 from http://appium.io/

Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 641–660.

Young Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–249.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Trans. Software Eng.* 41, 5 (2015), 507–525.

Farnaz Behrang and Alessandro Orso. 2019. Test Migration Between Mobile Apps with Similar Functionality. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 54–65.

Philip Bille. 2005. A survey on tree edit distance and related problems. *Theor. Comput. Sci.* 337, 1-3 (2005), 217–239.

Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 1998. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. HKUST-CS98-01, Hong Kong University of Science and Technology.

Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1 (2018), 4:1–4:27.

Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. 623–640.

Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 429–440.

Compuware. 2013. *Users Have Low Tolerance For Buggy Apps - Only 16% Will Try A Failing App More Than Twice.* Retrieved 2020-5 from https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/

Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 481–492.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*. 57–72.

Espresso. 2020. *Espresso.* Retrieved 2020-5 from https://developer.android.com/training/testing/espresso/

F-Droid. 2020. *F-Droid.* Retrieved 2020-5 from https://f-droid.org/en/

Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, and Geguang Pu. 2018a. Efficiently manifesting asynchronous programming errors in Android apps. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 486–497.

Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018b. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 408–419.

Amin Milani Fard, Mehdi MirzaAghaei, and Ali Mesbah. 2014. Leveraging existing tests in automated test generation for web applications. In *International Conference on Automated Software Engineering (ASE)*. 67–78.

Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 308–318.

Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 141–152.

FindBugs. 2020. *FindBugs*. Retrieved 2020-5 from http://findbugs.sourceforge.net/

Google Play. 2020. *Google Play Store*. Retrieved 2020-5 from https://play.google.com/store/apps

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 *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 269–280.

Gang Hu, Xinhao Yuan, Yang Tang, and Junfeng Yang. 2014. Efficiently, effectively detecting mobile app bugs with AppDoctor. In *Ninth Eurosys Conference (EuroSys)*. 18:1–18:15.

Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 269–282.

Infer. 2020. *Infer*. Retrieved 2020-5 from https://fbinfer.com/

Reyhaneh Jabbarvand, Forough Mehralian, and Sam Malek. 2020. Automated construction of energy test oracles for Android. In *28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 927–938.

Jacoco Team. 2020. *Jacoco*. Retrieved 2020-5 from https://www.eclemma.org/jacoco/

Casper Svenning Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis (ISSTA)*. 67–77.

Wang Jue, Jiang Yanyan, Xu Chang, Cao Chun, Ma Xiaoxing, and Lu Jian. 2020. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 469–480.

K-9 Mail Team. 2020. *K-9 Mail*. Retrieved 2020-11 from https://github.com/k9mail/k-9/

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

Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Trans. Reliability* 68, 1 (2019), 45–66.

Yavuz Köroglu and Alper Sen. 2018. TCM: Test Case Mutation to Improve Crash Detection in Android. In *21st International Conference on Fundamental Approaches to Software Engineering (FASE)*. 264–280.

Yavuz Köroglu and Alper Sen. 2019. Reinforcement Learning-Driven Test Generation for Android GUI Applications using Formal Specifications. *CoRR* abs/1911.05403 (2019). http://arxiv.org/abs/1911.05403

Emily Kowalczyk, Myra B. Cohen, and Atif M. Memon. 2018. Configurations in Android testing: they matter. In *1st International Workshop on Advances in Mobile App Analysis (A-Mobile)*. 1–6.

Edmund S. L. Lam, Peilun Zhang, and Bor-Yuh Evan Chang. 2017. ChimpCheck: property-based randomized test generation for interactive apps. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 58–77.

Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 216–226.

Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. 23–26.

Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer Across Mobile Apps Through Semantic Mapping. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 42–53.

Ying-Dar Lin, José F. Rojas, Edward T.-H. Chu, and Yuan-Cheng Lai. 2014. On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. *IEEE Trans. Software Eng.* 40, 10 (2014), 957–970.

Lint. 2020. *Improve your code with lint checks*. Retrieved 2020-5 from https://developer.android.com/studio/write/lint.html

Localytics. 2019. *25% of Users Abandon Apps After One Use*. Retrieved 2020-5 from http://info.localytics.com/blog/25-of-users-abandon-apps-after-one-use

Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. 2019. Preference-wise testing for Android applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 268–278.

Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 224–234.

Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 599–609.

Rupak Majumdar and Koushik Sen. 2007. Hybrid Concolic Testing. In *29th International Conference on Software Engineering (ICSE)*. 416–426.

Maryam Imtiaz Malik, Muddassar Azam Sindhu, Akmal Saeed Khattak, Rabeeh Ayaz Abbasi, and Khalid Saleem. 2021. Automating test oracles from restricted natural language agile requirements. *Expert Syst. J. Knowl. Eng.* 38, 1 (2021).

Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *International Symposium on Software Testing and Analysis (ISSTA)*. 94–105.

Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: exploiting popular functionalities for the generation of semantic GUI tests with Oracles. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 280–290.

Material Design. 2020. *Material Design*. Retrieved 2020-5 from https://material.io/design

Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. What Test Oracle Should I Use for Effective GUI Testing?. In *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. 164–173.

Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. 2000. Automated test oracles for GUIs. In *ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. 30–39.

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 (ICSE)*. 559–570.

Nariman Mirzaei, Sam Malek, Corina S. Pasareanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing Android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

Monkey. 2020. *Monkey*. Retrieved 2020-5 from http://developer.android.com/tools/help/monkey.html

Motherboard. 2020. *Here's the Shadow Inc. App That Failed in Iowa Last Night*. Retrieved 2020-5 from https://www.vice.com/en_us/article/y3m33x/heres-the-shadow-inc-app-that-failed-in-iowa-last-night

NewPipe Team. 2020. *NewPipe*. Retrieved 2020-11 from https://github.com/TeamNewPipe/NewPipe

Mateusz Pawlik and Nikolaus Augsten. 2016. Tree edit distance: Robust and memory-efficient. *Inf. Syst.* 56 (2016), 157–173.

Joao Felipe Pimentel. 2020. *Python APTED algorithm for the Tree Edit Distance*. Retrieved 2020-5 from https://github.com/JoaoFelipe/apted

Robotium. 2020. *Robotium: User scenario testing for Android*. Retrieved 2020-5 from http://www.robotium.org

Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2018. Automation of Android applications functional testing using machine learning activities classification. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 122–132.

Konstantin Rubinov and Luciano Baresi. 2018. What Are We Missing When Testing Our Android Apps? *Computer* 51, 4 (2018), 60–68.

Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. 2016. A Survey on Metamorphic Testing. *IEEE Trans. Software Eng.* 42, 9 (2016), 805–824.

Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA)*. 864–880.

Sixth Tone. 2019. *E-Commerce App Loses 'Tens of Millions' From Coupon Glitches.* Retrieved 2020-5 from https://www. sixthtone.com/news/1003483/e-commerce-app-loses-tens-of-millions-from-coupon-glitches

Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI Testing for Android Applications. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 27–37.

Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why My App Crashes Understanding and Benchmarking Framework-specific Exceptions of Android apps. *IEEE Transactions on Software Engineering* (2020).

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.

Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 119–130.

Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding compiler bugs via live code mutation. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. 849–863.

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 *30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 204–215.

Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 303–314.

Sandro Tolksdorf, Daniel Lehmann, and Michael Pradel. 2019. Interactive metamorphic testing of debuggers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 273–283.

Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal* 27, 1 (2019), 149–201.

Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying Android Applications Using Java PathFinder. *SIGSOFT Softw. Eng. Notes* 37, 6 (Nov. 2012), 1–5.

Mario Linares Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017a. How do Developers Test Android Applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 613–622.

Mario Linares Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017b. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 399–410.

Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 738–748.

Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 137:1–137:27.

Qing Xie and Atif M. Memon. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Trans. Softw. Eng. Methodol.* 16, 1 (2007), 4.

Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4 (2011), 544–558.

Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE)*. 250–265.

Xun Yuan and Atif M. Memon. 2007. Using GUI Run-Time State as Feedback to Generate Test Cases. In *29th International Conference on Software Engineering (ICSE)*. 396–405.

Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *International Conference on Software Testing, Verification and Validation (ICST)*. 183–192.

Kaizhong Zhang and Dennis E. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262.

Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 253–262.