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.

Additional Key Words and Phrases: GUI testing, Android apps, Non-crashing functional bugs, Logic bugs

*This work was mainly conducted when Ting Su was a postdoc at ETH Zurich.
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-deployement, 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 2021; 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 \(\ell_1\) 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 one of its basic functionality: a user can start an activity (e.g., Cinema in the list on \(\ell_1\)) and attach

\(^1\)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.
pictures to this activity to record information (e.g., the movie the user watched in a cinema on \(\ell_3\)). 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 \(\ell_3\), and thus switching to another activity (e.g., Cleaning) on page \(\ell_3\) 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 \(\ell_6\)): when the user switches to Cleaning (\(\ell_3' \Rightarrow u_1\)) and takes a picture for it (\(u_1 \Rightarrow u_2\), Cleaning’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 \(\ell_5'\) and \(\ell_6'\).

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 \(\ell_4'\). 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], AntennaPod [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 these apps are popular (with 1K+ stars on GitHub and 1M+ installations on Google Play), diverse
1. Mine GUI Transitional Model
2. Inferring Independent Views from Seed Tests
3. Mutant Test Generation and Execution
4. Property Validation via GUI-based Oracle Checking

Fig. 2. Workflow of our approach.

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 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 $\ell_1$ and $\ell_2$ in Scenario I are mapped to $s_0$ because of their structural similarity, while $\ell_3$ 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 $\ell_1$, view Cinema, Sleeping, and Cleaning are independent views. Because from Fig. 1(d), $\ell_1$’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 $\ell_1$, 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 $\ell_2$ and $\ell_3$, but will not be updated on $\ell_4$ (because Cinema does not exist on $\ell_4$). On $\ell_4$, picture will be annotated as active when it is clicked. Note that on $\ell_2$, 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 \( \ell_5 \) (in Fig. 1(a)) as the pivot layout, based on the active view information, we know that \textit{Cleaning} is inactive and independent from \textit{Cinema}. Then, we start from \textit{Cleaning}, and query the transitional model (see Fig. 1(c)) to obtain any possible event traces that can return back to layout \( \ell_5 \). Here, one possible independent event trace is “click \textit{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 \( \Delta \)) 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 \( \Lambda \) of the seed test is not contained in \( \Lambda' \) 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 \( \ell_4 \) and \( \ell_6 \), the GUI effect of the event trace between \( \ell_4 \) and \( \ell_6 \) can be characterized by the deletion of a picture under \textit{Cinema} (denoted as \( \Delta \)). Similarly, in Fig. 1(b), we can know the GUI effect of the event trace between \( \ell_4' \) and \( u_6 \) can be characterized by the deletion of a picture under \textit{Cleaning} (denoted as \( \Delta' \)). 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 \( \ell \), \( i.e., \) 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 (\texttt{Button}) or a text field (\texttt{EditText}). A non-leaf view’s \( w.type \) can be a layout view, \( e.g., \) \texttt{ListView}, \texttt{LinearLayout}, which arranges or displays other views in specific orders and appearances. When \( P \) is not in the foreground, we define \( \ell = \perp \).

A GUI event \( e = (t, r, o) \) is a tuple, where \( e.t \) denotes \( e \)’s event type (\( e.g., \) \texttt{click}, \texttt{edit}, \texttt{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 \texttt{edit}).

A GUI test is a sequence of events \( E = [e_1, e_2, ..., e_n] \). \( E \) 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 \( L = [\ell_1, \ell_2, ..., \ell_{n+1}] \), \( I \) 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 \( \ell_{i+1} \) is the layout due to the execution of \( e_i \) on \( \ell_i \). 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 triple $M = (S, \Sigma, \delta)$, where $S$ denotes a set of abstract states $\{s\}$, $s$ is a subset of layouts in $L$, $\Sigma$ the set of events that $P$ accepts, and $\delta : S \times \Sigma \rightarrow S$ the transition function with each transition $(s, e, s') \in \delta$ 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 $\ell$ and $\ell'$ are structurally equivalent. If $\ell$ and $\ell'$ 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 Equivalence-with relation (denoted by $\equiv$) below.

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

**Example.** In Fig. 1(a), $\ell_1 \equiv \ell_2$ because their structures are equivalent (both are similar to Fig. 1(d)). $\ell_2 \not\equiv \ell_3$ because $\ell_3$ has a picture view. For $\ell_4$ in Fig. 1(a) and $\ell'_4$ in Fig. 1(b), $\ell_4 \equiv \ell'_4$ 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 $\ell'_4$. 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 $\ell$, we select and execute an event $e$ on its receiver view $e.r(\ell)$. Then, $e$ takes us to a new layout $\ell'$, from which we continue to select the next event. During this process, a self-loop transition $(s, e, s')$ is added into $M$ if $\ell_{abs} \equiv \ell'_{abs}$ (i.e., Equivalence-with($\ell, \ell'$)). Otherwise, a transition connecting the two (different) states is added. Here, $s$ and $s'$ are the states in $M$ corresponding to the abstract layouts $\ell_{abs}$ and $\ell'_{abs}$, 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 $Y(e)$ to capture these two pieces of information. Assume we are at the $i$th round of event execution, i.e., $\ell_{i+1} = e_i(\ell_i)$, we first recursively update the weight $Y(\cdot)$ 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 $\ell_{i+1}$, i.e., $[\ell_1, \ldots, \ell_{i+1}]$), and then select the event with the largest weight on layout $\ell_{i+1}$:

$$Y_{i+1}(e) = \frac{Y_i(e) + \sum Y_i(new\_events(e))}{exec\_times(e)^2} \quad (1)$$

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, $\forall e \in E, Y_1(e) = 100$, $exec\_times(e) = 1$, and $new\_events(e) = \emptyset$.

- **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

---

4Our 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).
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 \( \perp \)) and the concept of active views.

**Definition 2.** Independent_from (\( \perp \)). Conceptually, a GUI layout \( \ell \) can be partitioned into non-overlapping regions that are rooted by a set of group views \( G = \{ w_1, w_2, \ldots, w_n \} \), where \( w_i \in \ell \) 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., \( w_1 \perp w_2 \), if
(1) \( w_1 \) and \( w_2 \) are the children of two different group views in \( G \), i.e., \( \text{Group}(w_1) \neq \text{Group}(w_2) \).
(2) \( w_1 \) and \( w_2 \) are the children of the same group view, i.e., \( \text{Group}(w_1) = \text{Group}(w_2) \), and satisfy that \( w_1 \) and \( w_2 \) are siblings\(^6\) and \( w_1\_\text{type} = w_2\_\text{type} \).

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 \( \ell_1 \), 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. \( w_4 \) (No Activity) \( \perp w_1 \) holds because \( w_4 \) is in another group view \( w_5 \) (ViewPager). \( w_3 \) (Camera) \( \perp w_1 \) also holds because \( w_4 \) is not within a group view. Fig. 1(e) shows the GUI tree of \( \ell_4 \), where \( w_1 \) (i.e., Cleaning and its picture) \( \perp w_4 \) (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 \( G = \{ w_1, w_2, \ldots, w_n \} \) be the set of group views of layout \( \ell \) and \( h_{\ell} \in H \) be the map that maintains the active view information of \( \ell \), 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 \( \ell \). The active view information of \( \ell \) is updated as \( h_{\ell}[\text{Group}(w_i)] = w_i \).
2. A view \( w_i \) becomes inactive when an event \( e \), satisfying \( e.r(l) = w_j \land \text{Group}(w_i) = \text{Group}(w_j) \), was executed on \( \ell \). The active view of \( \ell \) is updated as \( h_{\ell}[\text{Group}(w_j)] = w_j \).

---

\(^5\)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.
(3) A view \( w_i \) stays active or inactive, when an event \( e \), satisfying \( e.r(l) = w_j \wedge \text{Group}(w_j) \neq \text{Group}(w_i) \), was executed on \( l \). The active view of \( l \) is updated as \( h_l[\text{Group}(w_j)] = w_j \).

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 \( l'_2 \), 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 \( l'_2 \) 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 \( l'_3 \) because Cleaning does not exist on \( l'_3 \). On the other hand, clicking Camera on \( l'_1 \) 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 \texttt{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.** \texttt{Similar_with} \( (\sim) \). Let \( l = \{w_1, \ldots, w_{|l|}\} \) be the layout on which event \( e \) is executed, and \( l' = \{w'_1, \ldots, w'_{|l'|}\} \) be another layout. Obviously, there exists a receiver view \( w_i \in l \) satisfying \( e.r(l) = w_i \). If there exists a “similar” view \( w'_j \in l' \) such that \( w'_j \) should be the receiver view of \( e \) on \( l' \), i.e., \( e.r(l') = w'_j \), 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 \( w'_j \in l' \) w.r.t. \( w_i \in l \), 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 \( w'_j \in l' \) 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 \texttt{Update_active_views} internally updates the active view information from the most recent similar GUI layout to the current one (Lines 11-16). \texttt{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 \texttt{Equivalent_with}. In practice, \texttt{Similar_layout_type} leverages the layout information to characterize the page type. For example, in Fig. 1(a), \( l_2 \) and \( l_5 \) have the similar layout type because they denote the same main page (although they are not structurally equivalent). Thus, we can update the active information from \( l_2 \) to \( l_5 \). The active view information is updated according to the \texttt{Similar_with} relation (Lines 13-15).


<table>
<thead>
<tr>
<th>Algorithm 1: Maintaining independent views and their active view information during the execution of a seed test.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function Executevie(Seed test E):</td>
</tr>
<tr>
<td>1. $\ell \leftarrow \text{GetGUI}(); L \leftarrow {\ell}; I \leftarrow []; I \leftarrow I \cup \text{Get_independent_views}(\ell); H \leftarrow []$</td>
</tr>
<tr>
<td>foreach $e \in E$ do</td>
</tr>
<tr>
<td>2. $h[\text{Group}(e.r(l))] \leftarrow e.r(l); H \leftarrow H \cup {h}$</td>
</tr>
<tr>
<td>3. $\ell \leftarrow \text{GetGUI}();$</td>
</tr>
<tr>
<td>4. $h \leftarrow \text{Update_active_views}(\ell, (L, I, H))$</td>
</tr>
<tr>
<td>5. $L \leftarrow L \cup {\ell}; I \leftarrow I \cup \text{Get_independent_views}(\ell);$</td>
</tr>
<tr>
<td>return $(L, I, H)$</td>
</tr>
</tbody>
</table>

Function Update_active_views(layout $\ell$, trace $(L, I, H)$):

foreach $(\ell_i, h_i) \in (L, H).\text{reverse()}$ do

if Similar_layout_type($\ell_i, \ell$) then

// Group($w_a$)=w_g, and $w_a$ is active
foreach $(w_g, w_a) \in h_i$ do

if $(\exists w_g' \in L.w_g \sim w_g'^{'} \wedge (\exists w_a' \in L.w_a \sim w_a'^{'}))$ then

// update the active view info
$h[w_g'] \leftarrow w_a'$

return $h$


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 $E' = [e_1, \ldots, e_{l-1}, e'_1, e'_2, \ldots, e'_{\text{m}}, e_{l}, \ldots, e_n]$ by inserting an event trace $\tau = [e'_1, e'_2, \ldots, e'_{\text{m}}]$ at a given pivot layout $\ell_i \in L (\ell_i = e_{l-1}(\ell_{l-1}))$. Importantly, $\tau$ 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 $\tau$ should satisfy: $e'_1.r(\ell_i)$ is inactive, i.e., the receiver view of $\tau$’s first event $e'_1$ on $\ell_i$ is independent from any active views on $\ell_i$. In this case, we say $\tau$ is an independent event trace. In other words, $\tau$ is independent from subsequent events in the seed test. Here, we leverage the active view information maintained by $\Pi_P(E) = (L, I, H)$ in Section 3.3. Section 5.4 validates the generality of this assumption.

**Definition 6.** Connect_with ($\sim\sim$). 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 $\tau_1$ can connect with $\tau_2$ (denoted by $\tau_1 \sim\sim \tau_2$) if $\tau_2$’s first event $e'_1$ can locate a similar receiver view on $\tau_1$’s last GUI layout $\ell_{|\tau_1|+1}$, i.e., $e'_1.r(\ell_{|\tau_1|+1}) \neq \bot$.

Thus, a valid inserted trace $\tau$ should satisfy $[e_1, \ldots, e_{l-1}] \sim\sim \tau \sim\sim [e_l, \ldots, e_n]$, i.e., $\tau$ can connect the event traces before and after the insertion position, i.e., $e'_1.r(\ell_i) \neq \bot \wedge e_i.r(\ell_{|\tau_1|+1}) \neq \bot$. 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 $\ell_3$ as a pivot layout to insert independent traces. The layout $\ell_3$ can be mapped to an abstract state $s_i$ in the transitional model (Fig. 1(c)). From $\ell_3$’s layout information, we know that Cinema, Cleaning and Sleeping are independent views and Cinema is active. Thus, from $\ell_3$, a valid event trace $\tau$ could start from Cleaning.
Algorithm 2: Generate mutant tests from a seed test

```
1 \langle L, I, H \rangle \leftarrow \text{Execute}(E); \text{mutants} \leftarrow []
2 \text{Function Generate_mutants(Seed test } E, \text{ Model } M):$
3 \text{foreach } \langle \ell_i, h_i \rangle \in \langle L, H \rangle \text{ do}
4 \quad \ell_{abs} \leftarrow \text{Find_equivalent_state}(M, \ell)
5 \quad \text{independent_traces} \leftarrow \text{Search_traces}(i, \ell_{abs}, M)
6 \quad \text{tests} \leftarrow \text{Assemble}(E, \ell, \text{independent_traces})
7 \quad \text{mutants} \leftarrow \text{mutants} :: \text{tests}
8 \text{return mutants}
9 \text{Function Search_traces(insert position } i, \text{ abstract layout } \ell_{start}, \text{ Model } M):$
10 \quad \text{candidate_traces} \leftarrow []; \text{stack} \leftarrow {};
11 \quad \text{foreach } \langle \ell_{start}, e, \ell_{next} \rangle \in M[\ell_{start}] \text{ do}
12 \quad \quad \text{// Get the transitions starting from } \ell_{start} \in M
13 \quad \quad \text{if } e.r(L[i]) \in \text{Get_independent_views}(L[i]) \land \text{Is_inactive}(e) \text{ then}
14 \quad \quad \quad \text{// Pick } r \text{'s the starting event from inactive views on } \ell_i, \text{i.e. } L[i]
15 \quad \quad \quad \quad \text{stack.Push}\{\lfloor e \rfloor\}
16 \quad \text{while } \text{stack do}
17 \quad \quad \tau \leftarrow \text{stack.Pop}() \text{// Pop } r \text{ in breadth-first random}
18 \quad \quad \ell_{\tau-\text{last}} \leftarrow \text{Get_last_layout}(\tau)
19 \quad \quad \text{if } (\ell_{\tau} \neq \ell_{\tau-\text{last}}) \land \left(\left[ e_0, \ldots, e_{i-1} \right] \leadsto \tau \leadsto \left[ e_i, \ldots, e_n \right]\right) \land |r| < \text{MAX_LENGTH} \text{ then}
20 \quad \quad \quad \text{// Form a loop trace satisfying Connect_with}
21 \quad \quad \quad \text{candidate_traces} \leftarrow \text{candidate_traces} :: [r]
22 \quad \quad \text{else}
23 \quad \quad \quad \text{foreach } \langle \ell_{\tau-\text{last}}, e, \ell'_{\text{next}} \rangle \in M[\ell_{\tau-\text{last}}] \text{ do}
24 \quad \quad \quad \quad \text{stack.Push}(\tau :: [e])
25 \quad \quad \text{if } |\text{candidate_traces}| > \text{MAX_NUM} \text{ then}
26 \quad \quad \quad \text{break}
27 \quad \text{return candidate_traces}
```

or Sleeping. If \( r \)'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., \( \tau_1 = \langle \text{Cleaning, Camera} \rangle \), and \( \tau_2 = \langle \text{Sleeping, Camera} \rangle \). Note that \( \tau_1 \) and \( \tau_2 \) satisfy the Connect_with relation. Take \( \tau_1 \) as an example, \( \tau_1 \)'s first event, i.e., clicking Cleaning, can find a similar receiver view Cleaning on the runtime layout \( \ell_1 \) (\( \tau_1 \)'s starting layout), while the original event clicking Nav on \( \ell_3 \) (following \( \tau_1 \)) can find a similar receiver view Nav on the abstract layout \( s_1 \) (i.e., \( \tau_1 \)'s last layout). Thus, \( \tau_1 \) forms a valid independent loop trace that can start from \( \ell_3 \) and return back to \( \ell_3 \).

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 \( \ell_{abs} \) on the transitional model \( M \) (Line 4), from which it starts searching independent traces (Line 5). In function \text{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 \( \tau \) should (1) be independent from subsequent events (Lines 11-13), and (2) form a loop trace at \( \ell_{abs} \) 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, $\ell_i$ and $\ell_j$ (i < j), satisfying $\text{Similar}_\text{layout}_\text{type}(\ell_i, \ell_j)$, their GUI effect, denoted by $\Delta(\ell_i, \ell_j)$, is equivalent to the differences of GUI views between $\ell_i$ and $\ell_j$. $\Delta(\ell_i, \ell_j)$ characterizes the effect (i.e., changes of views) of executing the event trace $[e_i, \ldots, e_{j-1}]$ between $\ell_i$ and $\ell_j$. Formally, $\Delta(\ell_i, \ell_j)$ can be represented as:

$$\Delta(\ell_i, \ell_j) = (\ell_i \setminus \ell_j) \cup (\ell_j \setminus \ell_i)$$

where \setminus denotes set difference and $\cup$ 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 $\ell_i$ to $\ell_j$, thus the GUI effects of executing the event trace $[e_i, \ldots, e_{j-1}]$ between $\ell_i$ and $\ell_j$.

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 $\ell_i$ and $\ell_j$). 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) $(w, \bot) \in \Delta(\ell_i, \ell_j)$ for $w \in \ell_i$ denotes a view deletion, (2) $(\bot, w) \in \Delta(\ell_i, \ell_j)$ 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 $\ell_4$ and $\ell_6$, i.e., $\Delta(\ell_4, \ell_6)$, is the deletion of a picture under Camera, 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 $\{(\text{pic}_\text{Camera}, \bot)\}$. Note that $\ell_4$ and $\ell_6$ is comparable because they are of same layout type (i.e., the Diary page), while $\ell_4$ and $\ell_6$ is not comparable because they are of different layout types ($\ell_4$ denotes the main page of ActivityDiary). Similarly, in Fig. 1(b), the GUI effect between $\ell'_4$ and $\ell'_6$ (in Fig. 1(b)) is $\Delta(\ell'_4, \ell'_6) = \{(\text{pic}_\text{Camera}, \bot)\}$, while the GUI effect between $\ell'_4$ and $\text{u_6}$ is $\Delta'(\ell'_4, \text{u_6}) = \{(\text{pic}_\text{Cleaning}, \bot)\}$.

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 $\Delta(\ell_i, \ell_j) \notin \Delta'(\ell'_i, \ell'_j)$ i.e., an oracle violation is detected.

![Fig. 3. Oracle checking between Scenario I in Fig. 1(a) and Scenario II (actual behavior) in Fig. 1(b).](attachment:image.png)

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

Algorithm 3: GUI-based Oracle Checking

Function Oracle_Checking(Seed test E, Mutant test E'):

1. $L \leftarrow [\ell_1, \ldots, \ell_k, \ell_{k+1}, \ldots, \ell_{n+1}]$; $L' \leftarrow [\ell_1, \ldots, \ell_k]$; $L_T \leftarrow [\ell'_1, \ldots, \ell'_{k+1}, \ldots, \ell'_{n+1}]$;
2. $O \leftarrow$ True;
3. foreach $(i, j) \in L \times L'$ do
4.   if $(i \leq k \wedge k + 1 \leq j) \vee (k + 1 \leq i < j)$ then
5.     if Similar_layout_type($\ell_i, \ell_j$) then
6.       if $\Delta(\ell_i, \ell_j) \not\subseteq \Delta(\ell'_i, \ell'_j)$ then
7.         $O \leftarrow$ False;
8.   return $O$;

checking in this case is between $\Delta(\ell_4, \ell_6)$ from Scenario I and $\Delta'(\ell'_4, u_6)$ from Scenario II because $\ell_4$ and $\ell_6$ are of same layout type. Obviously, due to $\{(pic_{\text{Cinema}}, \bot)\} \not\subseteq \{(pic_{\text{Cleaning}}, \bot)\}$, 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), 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. $L_T$ is the sequence of layouts due to the inserted trace $T$. 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 AP TED 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 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
Table 1. Challenges faced by Genie, the strategies/heuristics adopted by Genie to tackle the corresponding challenges, and related algorithm pseudocode and functions.

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

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 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
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).

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

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.

### 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 (843K), 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 unrepeatable tests. Because the inserted event traces may lead to the GUI states that cannot connect with subsequent events. For example, Transistor has many unrepeatable mutants 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...
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.

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

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 #1 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 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
Table 7. Detailed analysis results on false positives (FPs).

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

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

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 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...
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.

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

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

e 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 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 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.,
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.

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], CHIMP Check [Lam et al. 2017], APPFlow [Hu et al. 2018], ACAT [Rosenfeld et al. 2018], AppTestMigrator [Lin et al. 2019], CraftDroid [Lin et al. 2019]), linear-time temporal logic (LTL) formulas (e.g., FARLEAD-Android [Köroglu and Sen 2021]), 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

We thank the anonymous SPLASH/OOPSLA reviewers for their valuable feedback. This work is partially supported by NSFC Project No. 62072178. Zhendong Su and Ting Su were partially supported by a Google Faculty Research Award. Ting Su was also partially supported by a Swiss NSF Spark Award CRSK-2_190302. Jue Wang was partially supported by NSFC Project No. 61690204.

A APPENDIX

A.1 Prevalence of the Independent 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 [AmazeFileManager 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 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).
Table 10. Detailed statistics of investigating the prevalence of independent view property for finding non-crashing functional bugs.

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

**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**


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


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