# Detecting Non-crashing Functional Bugs in Android Apps via Deep-State Differential Analysis

Jue Wang
juewang591@gmail.com
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China

Yanyan Jiang*
jyy@nju.edu.cn
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China

Ting Su
tsu@sei.ecnu.edu.cn
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, China

Shaohua Li
shaoli@ethz.ch
Department of Computer Science, ETH Zurich, Switzerland

Chang Xu
changxu@nju.edu.cn
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China

Jian Lu
lj@nju.edu.cn
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China

Zhendong Su
zhendong.su@inf.ethz.ch
Department of Computer Science, ETH Zurich, Switzerland

## ABSTRACT

Non-crashing functional bugs of Android apps can seriously affect user experience. Often buried in rare program paths, such bugs are difficult to detect but lead to severe consequences. Unfortunately, very few automatic functional bug oracles for Android apps exist, and they are all specific to limited types of bugs. In this paper, we introduce a novel technique named deep-state differential analysis, which brings the classical "bugs as deviant behaviors" oracle to Android apps as a generic automatic test oracle. Our oracle utilizes the observations on the execution of automatically generated test inputs that (1) there can be a large number of traces reaching internal app states with similar GUI layouts, and only a small portion of them would reach an erroneous app state, and (2) when performing the same sequence of actions on similar GUI layouts, the outcomes will be limited. Therefore, for each set of test inputs terminating at similar GUI layouts, we manifest comparable app behaviors by appending the same events to these inputs, cluster the manifested behaviors, and identify minorities as possible anomalies. We also calibrate the distribution of these test inputs by a novel input calibration procedure, to ensure the distribution of these test inputs is balanced with rare bug occurrences.

We implemented the deep-state differential analysis algorithm as an exploratory prototype Odin and evaluated it against 17 popular real-world Android apps. Odin successfully identified 28 non-crashing functional bugs (five of which were previously unknown) of various root causes with reasonable precision. Detailed comparisons and analyses show that a large fraction (11/28) of these bugs cannot be detected by state-of-the-art techniques.

## CCS CONCEPTS

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

## KEYWORDS

Software testing, mobile apps, non-crashing functional bugs

**ACM Reference Format:**
Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting Non-crashing Functional Bugs in Android Apps via Deep-State Differential Analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549170

## 1 INTRODUCTION

**Background and Motivation.** Non-crashing functional bugs [36] of Android apps caused by *program logic errors* seriously affect user experience [41]. Being buried in rare program paths, such bugs may not be captured in the quality assurance (testing) procedure and may lead to severe consequences [38].

Despite the rapid development of automatic test input generation for Android apps [2, 4, 8, 14, 18, 23, 24, 34, 35, 39], very few automatic functional bug oracles for Android apps exist [36, 37].

*Corresponding author.

*ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore*
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9413-0/22/11...$15.00
https://doi.org/10.1145/3540250.3549170

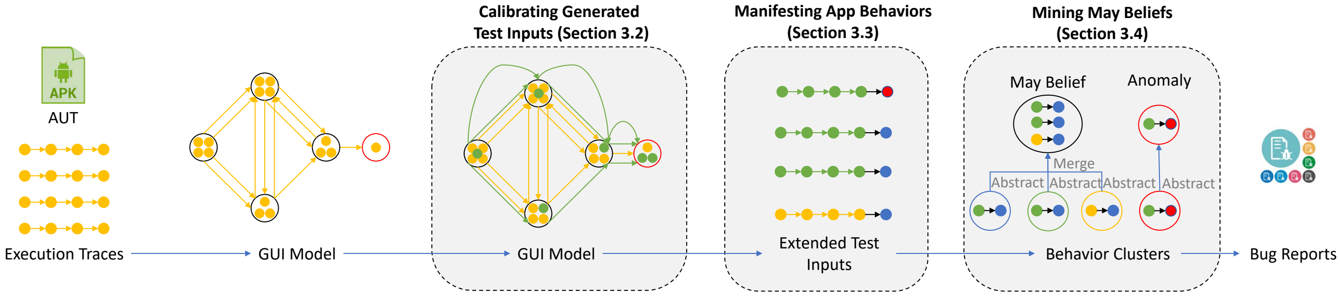

**Figure 1: Workflow of Odin**

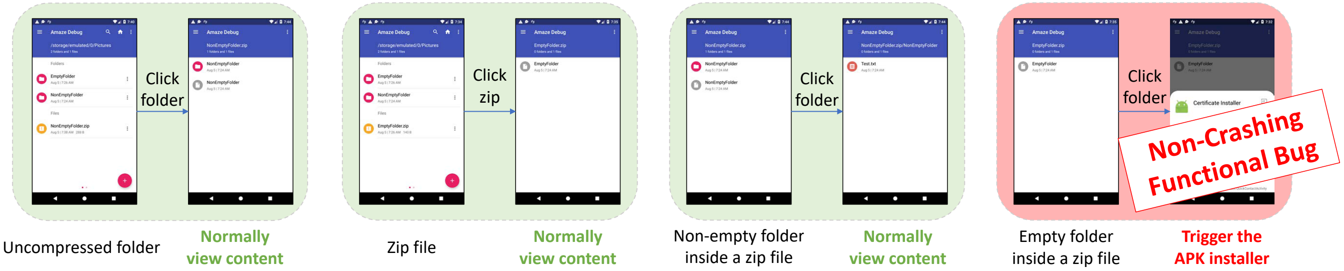

**Figure 2: A motivating bug example**

Specifically, metamorphic relations [7] can be established by comparing independent executions (Genie [36]) or with injected neutral UI/system events (Thor [1] and SetDroid [37]). However, these metamorphic relations all place a strong emphasize on the independence of two event fragments, and are fundamentally limited in identifying programming errors that occurred in dependent event fragments. For instance, Genie's authors acknowledged that only 29.5% of the non-crashing functional bugs in an empirical study falls into the scope of Genie [36].

A natural question then arises: can we exploit the massive, automatically generated test inputs to expose potential logic errors? This paper demonstrates that *it is possible to mine behavioral specifications for detecting non-crashing functional bugs without supervision*. Specifically, we observed that automatically generated test inputs can reach comparable states of an app, from which we can differentiate its behaviors to find non-crashing, functional bugs.

**Mining May-beliefs as Test Oracle.** This paper introduces *deep-state differential analysis*, a novel, generic, and automatic oracle for Android apps that brings the classical "bugs as deviant behaviors" [11] oracle to automatically generated GUI test inputs of Android apps. Specifically, we mine *may beliefs*[1] (expected app behavior specifications) from execution traces and use such beliefs to identify anomalies as deviant behaviors likely caused by bugs.

Our mined beliefs are based on the following two observations:

(1) There can be a large number of traces (test inputs) that end up with a similar GUI layout. Suppose that the app is mostly functionally correct, then only a small portion of them could possibly end up with an erroneous state since developers would already notice bugs on frequent program paths.

(2) Android apps are designed with the "least surprise" principle [30] that performing the same sequence of actions on similar GUI layouts should trigger only a few limited behaviors of the app.

Therefore, if we append the same event sequence to all these test inputs (ending up with similar GUI layouts), their triggered app behaviors should fall into only a few behavior clusters–the majority becomes our belief (oracle). A corollary is thus *any small cluster should be considered potentially buggy*.

There are two main challenges to porting this idea to Android. First, beliefs (bugs as deviant behaviors) should be established over "balanced" traces of rare bug occurrences. However, existing coverage-directed test input generators often fail to provide such traces, which are necessary for mining reliable beliefs. Our approach addresses this challenge by a novel input calibration procedure that generates additional test inputs by repeatedly approximating random walks on a mined GUI model of the app, so that for each GUI model state representing a group of similar GUI layouts, there are a set of sufficiently balanced test inputs reaching it, with which we can safely mine reliable beliefs.

Second, the may-beliefs are extracted from the clustered "majority behaviors". Android apps are GUI-centered, and thus using the corresponding GUI layout sequence as a representation of the app's behavior is a common practice [2, 4, 14, 18, 34]. However, GUI layouts often contain rich but redundant, or even non-deterministic information [2, 14, 18, 23, 34] which hinders precise clustering and majority extraction. To mitigate this challenge, our approach performs a GUI-abstraction-based hierarchical clustering [9] on the GUI layout sequences. Specifically, starting with each sequence in its own cluster with no abstraction on the GUI layout, our approach iteratively (1) selects one more rule from a pool of GUI abstraction rules commonly adopted by existing work, (2) further abstracts each GUI layout accordingly, and (3) merge clusters that contain

[1] The original paper [11] also defines a set of "must beliefs" of formal specifications a system must satisfy. A few existing techniques [1, 15] proposed manual must beliefs (assertions concerning specific system behaviors) and are out of our scope.

similar abstracted GUI layout sequences. It attempts to detect outliers, which are accordingly identified as anomaly behaviors while others as may-beliefs, by calculating and comparing z-scores [13] of the clusters' size after each round of merging.

**The Odin Prototype Tool**. We implemented the deep state differential analysis algorithm as an exploratory prototype Odin and evaluated it against 17 popular real-world Android apps. Odin successfully identified 28 non-crashing functional bugs, five of which were previously unknown. We reported these unknown bugs to the developers, and all have been confirmed. Detailed comparisons and analyses show that (1) a large fraction (11/28) of these bugs cannot be detected by the state-of-the-art technique Genie [36], (2) the input calibration and GUI-abstraction-based hierarchical clustering can indeed improve Odin's effectiveness, and (3) Odin can identify non-crashing functional bugs of various root causes with reasonable precision.

In Summary, this paper makes the following key contributions:

- We proposed deep-state differential analysis and brought the "bug as deviant behaviors" idea to Android apps as a novel, generic, and automatic test oracle.
- We implemented the prototype tool Odin and will make it public[2].
- We evaluated the tool against real-world Android apps, and the results are encouraging that Odin well complements state-of-the-art techniques.

The rest of this paper is organized as follows. Section 2 provides an overview of our approach with a motivating example. Details of our approach are discussed in Section 3. In Section 4 we introduce the implementation of Odin and in Section 5 the evaluation is conducted. Related work is discussed in Section 6, and finally Section 7 concludes the paper.

## 2 OVERVIEW

It is non-trivial to port the "bugs as deviant behaviors" idea to Android despite its simplicity. This section describes the overview of our approach illustrated in Figure 1, and discusses the challenges and their mitigation.

Our approach takes the app under test and a set of its GUI execution traces (a GUI execution trace is an event sequence combined with the GUI layout sequence obtained by sending the event sequence to the app) as input, and outputs bug reports. First, to provide balanced GUI execution traces of rare bug occurrences with which we can mine reliable beliefs, our approach constructs a GUI model from the traces and preforms an input calibration procedure for each GUI model state representing a set of similar GUI layouts, and then extends calibrated inputs ending up at the GUI state for manifesting normal and anomaly app behaviors. Next, to find a deliberate abstraction that is simultaneously effective in distinguishing anomaly behaviors and resistant to noises, it adopts a GUI-abstraction-based hierarchical clustering algorithm to cluster the manifested behaviors, mines may beliefs, and detects anomalies from the behavior clusters. Anomalies are reported as potential non-crashing functional bugs.

We motivate our approach by a previously unknown bug (found by our deep-state differential analysis) in the Amaze file manager app (Figure 2). A user can use Amaze to view folders on the device and select a folder to view its content by clicking its icon. There is a subtle bug that Amaze sometimes erroneously identified an empty folder inside a zip file as an APK. Clicking such a folder triggers the system's installer, while the expected behavior is displaying an empty folder. This is a typical non-crashing functional bug.

State-of-the-art automated test input generators [14, 34, 39] can occasionally trigger this erroneous behavior. We explain below how do we establish the belief that "clicking an folder should not open an app installer".

### 2.1 Calibrating Generated Test Inputs

May-beliefs assume that (1) traces are sufficiently diverse, i.e., each (internal) app state has a considerable portion of traces reaching it, and (2) buggy traces are rare. However, automatically generated test inputs are drawn from a highly skewed probability distribution in which dominant inputs can only reach shallow app states with limited diversity.

This is due to the coverage-directed nature of automatic test input generators [4, 8, 14, 24, 34]. First, to maximize covered (internal) states or app code within a limited time, automatic test input generators tend to stick to a profitable trace for long-term exploration, producing a skewed distribution of test inputs. If such a profitable trace is accidentally erroneous, it cannot be identified as rare and buggy. Furthermore, deeper internal states are generally more difficult to reach. Consequently, test inputs that reach deep states are also rare and deep states lack sufficient traces for deriving strong beliefs on majority behaviors.

**Challenge 1**: How to effectively generate massive, balanced test inputs that provide all explored app GUIs (shallow or deep) with sufficiently many traces of behavioral diversity?

To mitigate this challenge, our approach *calibrates* the generated test inputs by approximating a random walk on a mined GUI model of the app. First, our approach mines the app's GUI model from the GUI execution traces of massive, automatically generated test inputs by grouping similar GUI layouts as GUI model states[3] and adding transitions according to the input event between each pair of GUI layouts in the execution traces. Note that our approach ensures that no non-deterministic transitions (i.e., an event can trigger two different transitions from the same GUI model state) in the GUI model. Given any GUI model state $\sigma$, no matter shallow or deep, our approach simulates a *random walk* on the GUI model, i.e., finding a random path terminating at $\sigma$. The random walk forces all outgoing transitions from the same GUI model state the same probability. Specifically, suppose that the random walk is at GUI model state $\sigma_i$, and there are transitions $(\sigma_i, \sigma_j)$ in the GUI model ($\sigma_j \in \Sigma$, $\sigma_i \neq \sigma_j$). Then, our approach selects each $\sigma_j \in \Sigma$ with probability $\frac{1}{|\Sigma|+1}$ and $\sigma_i$ (terminating at the same state) with $\frac{1}{|\Sigma|+1}$, to obtain the next state $\sigma_i'$ in the random walk. In each step of the random walk, our approach also randomly selects an event that can manifest the transition $(\sigma_i, \sigma_i')$, yielding a test input (event

[2] https://automatedoracleforandroid.github.io/Odin/

[3] We consider two GUI layouts similar if they handle the same set of events, i.e., events are interchangeable for both states.

sequence) that can potentially terminates with GUI model state $\sigma$. By repeating this procedure, our approach generates sufficiently many and diverse test inputs for each GUI model state. Note that the goal of our calibration is to generate, for each GUI model state $\sigma$ (no matter whether it is shallow or deep) that has been covered by some given test inputs, a set of diverse traces reaching $\sigma$. It does not guarantee to cover all deep states of the app, which is a challenge for any test input generation approach [4, 8, 14, 34, 39].

GUI modeling inevitably losses information, and test input obtained from a random walk may not terminate at a designated $\sigma$ in real execution. To enable a practical approximation, when such an inconsistency occurs, our approach re-mines the GUI model from the original execution traces and the newly obtained ones causing inconsistency.

In the motivating example, automatic test input generators can provide massive execution traces reaching the same GUI model state with folders listed on similar GUI layouts (the first GUI page of each GUI page pair in Figure 2). However, the generators are likely to stick to an erroneous one (reaching a GUI page displaying the empty folder inside a zip file) and extend it for long-term exploration because they can cover extra pieces of code that are incorrectly executed to display an "apk file", leading to a skewed distribution in which the erroneous traces are no longer rare. Nonetheless, our input calibration procedure is able to calibrate the skewed distribution by generating additional test inputs for the GUI model state, most of which reach the correct app states.

## 2.2 Manifesting App Behaviors

Given sufficiently balanced test inputs that reach a GUI model state, we can reasonably assume that only a small fraction of the inputs terminate with an erroneous internal state. However, it is difficult to directly cluster internal states, which consist of low-level representation of data like serialized heap objects.

Alternatively, we leverage the observation that an internal state $s$ can be characterized by its future behaviors. Specifically, for a test input whose execution terminates with an internal state $s$, we can extend the test input by appending various event sequences to the test input, and all observable triggered behaviors (GUI layouts) of the appended events depend on $s$. Manifested anomaly GUI layouts indicate a buggy $s$. Unfortunately, appending all inputs with exhaustively enumerated event sequences yields an intractable search space. Therefore, our second challenge concerns efficient manifestation of diverse behaviors:

**Challenge 2**. How to efficiently extend inputs to manifest both normal and deviant app behaviors for establishing beliefs?

This challenge is mitigated by the observation that appending *only one* event to the inputs suffices for manifesting abnormal GUI layouts and establishing beliefs. Suppose that an input terminating with an internal state $s_0$ is extended to yield a state transition sequence of $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_{k-1}} s_{k-1} \xrightarrow{e_k} s_k$, in which $s_k$ displays an anomaly GUI layout. Then, our calibrated inputs should contain sufficiently many test inputs that terminate with a similar GUI layout with $s_{k-1}$, and only a minority of them hit an erroneous state. Thus, appending only one event $e_k$ to such inputs (of similar GUI layout, and thus $e_k$ can be applied to all of them) will establish a may-belief that the less frequently occurred GUI layout of $s_k$ indicates a bug.

In the motivating example, for the GUI model state representing app states in which a folde can be selected to view its content, our approach appends different single events to the inputs reaching it. When appending one click event on the folder, our approach manifests both normal and anomaly GUI layouts illustrated in Figure 2 for belief mining.

## 2.3 Mining May-beliefs as Test Oracle

Finally, the may-beliefs are extracted from the "majority behaviors" of GUI layouts. The one event we append to each input yields an internal state transition $s \xrightarrow{e} s'$ connecting a pair of GUI layouts $(\ell, \ell')$ that can be clustered. However, GUI layouts often contain rich but redundant, or even non-deterministic information [2, 14, 18, 23, 34] like dynamic Web contents. It is a challenge to find a deliberate abstraction that is simultaneously effective in distinguishing anomaly behaviors and resistant to noises:

**Challenge 3**: How to cluster GUI layout pairs for establishing correlations between anomaly behaviors and minority?

To mitigate this challenge, a GUI-abstraction-based agglomerative hierarchical clustering [9] is performed by our approach. Based on the common abstraction criteria adopted by existing techniques [5, 14, 34], we design a set of abstraction rules (e.g., abstracting away all texts in the GUI layout) that can be applied individually or combined. With each GUI layout pair in its own cluster with no abstraction, our approach iteratively selects one more abstraction rule and further abstracts all GUI layouts. After applying a new rule, our approach merges similar clusters. Our approach measures the similarity between two clusters by comparing the fingerprints of their contained GUI layout pairs. Specifically, it uses the *differential* between the abstracted GUI layouts in each layout pair as its fingerprint (denote by the tree editing distance [45] of abstracted GUI layouts), which enables the clustering algorithm to focus on the "instantaneous rate of change" (e.g., a newly added button or a piece of unchanged text) and ignore the accumulated non-determinism over test input execution, like the first-order differential of continuous functions. After each round of abstraction and cluster merging, our approach conducts a z-score-based [13] may-beliefs mining on the merged clusters in which outliers (if any) are considered anomalies while others may-beliefs. Such an iteration terminates when all abstraction rules are applied or any anomaly is found (and thus may-beliefs are mined).

In the motivating example in Figure 2, for the majority of GUI layout pairs, the second layout displays a list of different files. However, for the erroneous ones obtained by clicking an empty folder inside a zip file, the second layout displays a dialog asking permissions to "install" the folder. With a deliberate abstraction, the majority is grouped in one cluster (the may beliefs) while the erroneous ones in another (the anomalies), and the non-crashing functional bug is detected.

**Algorithm 1:** GUI model mining

```
Function MineModel($T_\ell = \{L_1, L_2, \ldots, L_n\}$)
    $V \leftarrow \emptyset; E \leftarrow \emptyset; \delta \leftarrow \emptyset$;
    for each $L = \langle \ell_0 \xrightarrow{e_1} \ell_1 \xrightarrow{e_2} \ldots \xrightarrow{e_m} \ell_m \rangle \in T_\ell$ do
        $V \leftarrow V \cup \{\{\ell_i\} | 0 \le i \le m\}$; // initially, no state is merged
        $E \leftarrow E \cup \{e_i | 1 \le i \le m\}$;
        $\delta \leftarrow \delta \cup \{\langle \{\ell_{i-1}\}, e_i, \{\ell_i\} \rangle | 0 < i \le m\}$;
    for each $(\sigma_i, \sigma_j) \in V \times V$ and $\sigma_i \neq \sigma_j$ do // in the BlueFringe ordering [17]
        $\langle V', \delta' \rangle \leftarrow$ merge-recursive($\sigma_i, \sigma_j, V, \delta$);
        if $\langle V', \delta' \rangle \neq \bot$ then
            $\langle V, \delta \rangle \leftarrow \langle V', \delta' \rangle$;
    return $\langle V, E, \delta \rangle$
Function merge-recursive($\sigma_1, \sigma_2, V, \delta$)
    if $\forall \ell_1 \in \sigma_1, \ell_2 \in \sigma_2$.similar($\ell_1, \ell_2$) then
        $V' \leftarrow V \backslash \{\sigma_1, \sigma_2\} \cup \{\sigma_1 \cup \sigma_2\}$; $\delta' \leftarrow \delta[\sigma_1/\sigma_2]$;
        for each $\langle \sigma_1, e, \sigma_k \rangle, \langle \sigma_1, e, \sigma_t \rangle \in \delta'$ and $\sigma_k \neq \sigma_t$ do
            $\langle V', \delta' \rangle \leftarrow$ merge-recursive($\sigma_k, \sigma_t, V', \delta'$);
            if $\langle V', \delta' \rangle = \bot$ then
                return $\bot$ // merge failed
        return $\langle V', \delta' \rangle \neq \bot$
    return $\bot$ // merge failed
```

## 3 DEEP-STATE DIFFERENTIAL ANALYSIS

### 3.1 Notations and Definitions

Android apps are GUI-centered and event-driven. At runtime, the GUI layout (snapshot) of the app $P$'s current (internal) state $s$, $\ell = L(s)$, is represented as a tree in which each node $\omega \in \ell$ is a GUI widget (e.g., a button or a text field object). A set of attributes are associated with each node, for instance $\omega.type$ refers to $\omega$'s widget type (e.g., a button or a text field) and $\omega.text$ refers to $\omega$'s displayed text ($\omega.text = \bot$ if no text is displayed). When $P$ is inactive (closed or paused to background), no GUI layout exists and $\ell = \bot$.

An GUI event $e = \langle t, r \rangle$ is a record in which $e.t$ and $e.r$ denote $e$'s event type and receiver widget, respectively. An event type can either be *click*, *long-click*, or *swipe* [4], and the receiver $r(\ell) = \omega$ denotes the widget $\omega \in \ell$ to which $e$ can be delivered ($r(\ell) = \bot$ if this event cannot be delivered to any widget of $\ell$).

Executing $P$ with an event sequence (i.e., a test input) $[e_1, e_2, \ldots, e_n]$ yields an *execution trace* $\tau = \langle s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} s_n \rangle$, in which $s_0$ is the initial (internal) app state, and sending event $e_{i+1}$ to state $s_i$ yields a new state $s_{i+1}$ ($0 \le i < n$). Its corresponding *GUI execution trace* is $L = \langle \ell_0 \xrightarrow{e_1} \ell_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} \ell_n \rangle$, where $\ell_i = L(s_i)$ ($0 \le i \le n$).

### 3.2 Calibrating Generated Test Inputs

We calibrate the automatically generated test inputs by simulating a random walk on an automatically mined GUI model.

**Mining a GUI Model.** The GUI model is mined from the GUI execution traces of massive, automatically generated test inputs.

**Algorithm 2:** Calibrating over GUI model $G(V, E, \delta)$

```
1  Function Calibrate($T$)
2      $T' \leftarrow \varnothing$;
3      repeat
4          $\sigma^\star \leftarrow \arg\min_{\sigma \in V} \sum_{\tau \in T}$ [reach($\tau^\star, \sigma$)];
5          $p \leftarrow$ random-walk($\langle \sigma_0 \rangle, \sigma^\star$);
6          if $p \neq \bot$ then
7              $T' \leftarrow T' \cup \{$to-input($p$)$\}$;
8      until sufficiently many traces are collected;
9      return $T \cup T'$;
10 Function random-walk($p = \langle \sigma_0, \sigma_1, \ldots, \sigma_i \rangle, \sigma^\star$)
11     if $|p| >$MAX_LIMIT then
12         return $\bot$
13     $\Sigma \leftarrow \{\sigma | \exists e \in E. \langle \sigma_i, e, \sigma \rangle \in \delta\}$;
14     if $\sigma_i = \sigma^\star$ then
15         $\Sigma \leftarrow \Sigma \cup \{\sigma_i\}$; // terminate at the designated GUI model state
16     for each $\sigma_{i+1} \in$ shuffle($\Sigma$) do
17         if $\sigma_{i+1} = \sigma_i$ and $\sigma_i = \sigma^\star$ then
18             $p' \leftarrow p$;
19         else
20             $p' \leftarrow$ random-walk($\langle \sigma_0, \sigma_1, \ldots, \sigma_1, \sigma_{i+1} \rangle, \sigma^\star$);
21         if $p' \neq \bot$ then
22             return $p'$
23     return $\bot$
```

Given the execution trace set $T = \{\tau_1, \tau_2, \ldots, \tau_n\}$, whose corresponding GUI execution trace set is $T_\ell = \{L_1, L_2, \ldots, L_n\}$, its corresponding GUI model is a tuple $G(V, E, \delta)$, in which $V$ is a set of GUI model states ($\{\sigma | \sigma \in V\}$ is a partition of all GUI layouts in $\bigcup_{L \in T_\ell} \{\ell | \ell \in L\}$), $E$ is the set of events sent to the app on some $L \in T_\ell$, and $\delta : V \times E \to V$ are the transitions in $G$.

To mine a minimal GUI model, we adopt the existing algorithm (Algorithm 1) in SwiftHand [8] that groups similar GUI layouts together and ensures transitions in the model are deterministic [5].

We consider two GUI layouts $\ell_1$ and $\ell_2$ are similar if and only if they can handle the same set of events, i.e., $\forall e \in E. e.r(\ell_1) \neq \bot \leftrightarrow e.r(\ell_2) \neq \bot$. Specifically, if we have witnessed in $T_\ell$ that an event $e$ is sent to a widget $\omega \in \ell_1$, we compute the *tree editing distance* between $\ell_1$ and $\ell_2$ using the classic Zhang-Shasha algorithm [45], and find the shortest editing operation sequence (each editing operation inserts, removes, or modifies a widget) that transforms $\ell_1$ to $\ell_2$. If $\omega$ is not removed during the transformation, there must exist a unique correspondence $\omega' \in \ell_2$. We thus let $e.r(\ell_2) = \omega'$. Otherwise, $e.r(\ell_2) = \bot$ and we consider $\ell_1$ and $\ell_2$ not similar. We discuss the editing operation sequence in more details in Section 3.4.

**Random Walk Simulation.** With the GUI model $G(V, E, \delta)$, we calibrate automatically generated test inputs by a random walk simulation with the algorithm presented in Algorithm 2.

Given a set of traces $T$, we select the least balanced GUI model state $\sigma^\star \in V$, i.e., $\sigma^\star$ has the fewest traces reaching it (Line 4). “Reaching” a GUI model state $\sigma$ is defined by visiting $\sigma$ one or more

[4] Our approach does not limit (and assume) the event types in the given test inputs. GUI events, such as text input or pinch, can be modeled as combinations of these three types of events. For example, a text input event can be modeled as a series of click events on the soft keyboard.

[5] All transitions in a GUI model $G(V, E, \delta)$ are deterministic if and only if for each $\sigma \in V$, $\nexists \langle \sigma, e, \sigma_1 \rangle, \langle \sigma, e, \sigma_2 \rangle \in \delta$ such that $\sigma_1 \neq \sigma_2$.

times:

$$\texttt{reach}(\tau, \sigma) \Leftrightarrow \left|\{s \in \tau \mid L(s) \in \sigma\}\right| \geq 1$$

Then, we try to find a path $p$ on the model reaching the least balanced $\sigma^\star$ via a simulated random walk starting at the initial GUI model state $\sigma_0$ (Line 5), and obtain the corresponding test input (Line 6-7). The initial GUI model state $\sigma_0$ is the state containing GUI layout of the app at the initial app state. Such a procedure is repeated until sufficiently many traces are obtained (Lines 3–7).

During a simulated random walk, when at a GUI model state $\sigma_i$, we first obtain $\Sigma$, the set of GUI model states that an outer transition from $\sigma_i$ can reach (Line 13). Moreover, if $\sigma_i$ is the target GUI model state $\sigma^\star$, we also add it to $\Sigma$ (Lines 14–15). We then iteratively select each $\sigma_{i+1} \in \Sigma$ with a uniform probability $\frac{1}{|\Sigma|}$ in turn (Line 16), and continue the random walk on $\sigma_{i+1}$ (Lines 17–20). Such an iteration terminates when all $\sigma_{i+1} \in \Sigma$ have been selected or a transition path to $\sigma^\star$ is found (Lines 16–22). To reduce the search space, we also limit the number of steps in one random walk (Lines 11–12).

**The Feedback Loop**. GUI modeling inevitably losses information [14, 34], and the test input we obtained by a simulated random walk may not actually yield the same transitional path in real execution. To reduce the occurrence of such inconsistencies, after we obtain a transitional path $p = \langle \sigma_0, \sigma_1, \dots, \sigma_{|p|}\rangle$ and its corresponding test input, we send the input to the app and record the actual execution trace $\tau$ and its corresponding transitional path $p' = \langle \sigma_0', \sigma_1', \dots, \sigma_{|p'|}'\rangle$ in the GUI model. If an inconsistency occurs, i.e., for a transition $\langle \sigma_i, \sigma_{i+1}\rangle$ $(0 \leq i < |p|)$ in $p$, the corresponding transition $\langle \sigma_i', \sigma_{i+1}'\rangle$ in $p'$ has $\sigma_i = \sigma_i'$ but $\sigma_{i+1} \neq \sigma_{i+1}'$, we add $\tau$'s GUI execution trace to $T_\ell$ from which the GUI model $G(V, E, \delta)$ is mined, and re-mine a new minimal GUI model $G'(V', E, \delta')$, which will be used for future calibration and belief mining.

## 3.3 Manifesting App Behaviors

Given any test input (trace) that terminates with GUI model state $\sigma$ (can be obtained by selecting a trace reaching $\sigma$ and removing all subsequent events after reaching $\sigma$), we extend it by exactly one event to manifest potentially buggy behaviors. Specifically, given a GUI model state $\sigma = \{L(s)\}$, all $s$ respond to the same set of events due to our similarity criteria. Therefore, the belief mining is conducted on the per-event basis. Suppose that all states s, where $L(s) \in \sigma$, responds to $e$. We enumerate all inputs $[e_1, e_2, \dots, e_n]$ reaching $\sigma$ and append $e$ to yield a new execution trace

$$\tau^+ = \langle s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} \dots \xrightarrow{e_n} s_n \xrightarrow{e} s_{n+1}\rangle,$$

and the layout pair of the last state transition $\langle L(s_n), L(s_{n+1})\rangle$ represents the manifested app behavior from which we mine may-beliefs.

## 3.4 Mining May Beliefs

**Hierarchical Behavior Clustering**. Given a set of GUI layout pairs $B = \{\langle \ell_1, \ell_1'\rangle, \langle \ell_2, \ell_2'\rangle, \dots, \langle \ell_n, \ell_n'\rangle\}$ obtained by sending a same event to inputs terminating with a same GUI model state, we conduct a agglomerative hierarchical clustering [9]. As shown in Algorithm 3, initially each layout pair starts with no abstraction in its own cluster (Lines 2-3). Next, we iteratively select one more abstraction rule to apply (Line 5). We study existing start-of-the-art techniques concerning GUI layout abstraction [5, 14, 34], and design a set of abstraction rules that can be applied individually or combined. There are three abstraction rules that can be applied, namely (1) sets the value of a specific attribute *attr* of all widgets (e.g., $\omega.text$) in a GUI layout $\ell$ to $\bot$, i.e., for all $\omega \in \ell$, set $\omega.attr = \bot$, (2) removes all widgets in a GUI layout $\ell$ that are not the receiver of any event, i.e., remove each widget $\omega \in \ell$ if $\forall e \in E.e.r(\ell) \neq \omega$, and (3) removes duplicate sub-trees of each widget $\omega$ in a GUI layout $\ell$ if it displays a list on the GUI, e.g, if $\omega.type = \texttt{ListView}$.

**Algorithm 3:** Hierarchical Behavior Clustering

```
1  Function Cluster($B = \{\langle \ell_1, \ell_1' \rangle, \langle \ell_2, \ell_2' \rangle, \dots, \langle \ell_n, \ell_n' \rangle\}$)
2      $R \leftarrow \emptyset$; // initially, no abstraction rule applied
3      $C \leftarrow \{\{\langle \ell, \ell' \rangle\} | \langle \ell, \ell' \rangle \in B\}$; // each layout pair in its own cluster
4      while not all abstraction rules are applied and $|C| > 1$ do
5          $r \leftarrow$ select-one-rule($C$); $R \leftarrow R \cup \{r\}$;
6          for each $\langle C_1, C_2 \rangle \in C \times C$ and $C_1 \neq C_2$ do
7              $\Delta_1 \leftarrow$ fingerprint($C_1, R$);
8              $\Delta_2 \leftarrow$ fingerprint($C_2, R$);
9              if $\Delta_1 = \Delta_2$ then
10                 $C \leftarrow C \backslash \{C_1, C_2\} \cup \{C_1 \cup C_2\}$;
11         $C_{error} \leftarrow$ detect-anomaly($C$);
12         if $C_{error} \neq \bot$ then
13             return $\langle C, C_{error} \rangle$
14     return $\bot$
15 Function fingerprint($C = \{\langle \ell_1, \ell_2 \rangle\}, R$)
16     $\langle \ell_1, \ell_2 \rangle \leftarrow$ random-choice($C$);
17     $\ell_1' \leftarrow$ abstract($\ell_1, R$);
18     $\ell_2' \leftarrow$ abstract($\ell_2, R$);
19     $\Delta \leftarrow$ tree-edit($\ell_1', \ell_2'$);
20     return $\Delta$
```

For each iteration, we select one more rule that (1) has not been applied, and (2) leads to minimal cluster merging, i.e., fewest clusters can be merged after applying this rule, and add it to the rule set $R$ (Line 5). With $R$, we enumerate each cluster pair $\langle C_1, C_2\rangle$ to determine whether they can be merged by comparing their new fingerprints (Line 6–10). After merging all clusters with identical fingerprint, we try to detect anomalies (and mine may-beliefs) on the merged clusters (Line 11). Such an iteration terminates when all abstraction rules are applied, only one cluster remains, or we have found an anomaly (Lines 4–13).

**Fingerprint Extraction**. As Algorithm 3 shows, to extract fingerprint of a cluster $C$, we randomly select one layout pair in $\langle \ell_1, \ell_2\rangle \in C$ (Line 16), apply the currently selected abstraction rules on $\ell_1$ and $\ell_2$, and calculate the *differential* between the abstracted layouts $\ell_1'$ and $\ell_2'$ as $C$'s fingerprint (Lines 17–20).

We denote the differential of two (abstracted) GUI layouts $\ell$ and $\ell'$ as a tree editing operation sequence [45] $\Delta$ transforming $\ell$ to $\ell'$. Each operation is a tuple $o = \langle t, \omega, \omega'\rangle$, where $t$ is its type (addition, deletion, or modification), $\omega$ is the target of $o$, and $\omega'$ is the widget after the operation is applied. For an adding operation, we add $\omega'$ as $\omega$'s leftmost child. For an deleting or modification operation, we replace $\omega$ with $\omega'$ (for deleting operations $\omega' = \bot$). Specifically for a modification operation $o = \langle t, \omega, \omega'\rangle$, if for an widget attribute *attr* (e.g., *text*) we have $\omega.attr = \omega'.attr$, we set $\omega.attr$ and $\omega'.attr$ to $\bot$ to reduce noises.

We randomly select one layout pair in $C$ and use its $\Delta$ as $C$'s fingerprint because (1) all layout pairs start in their own clusters, (2) we only merge clusters with identical fingerprints, and (3) for any two layout pairs in one cluster, applying one more abstraction rule may only remove identical operations in their $\Delta$s. Therefore, we can safely assume that all layout pairs in a cluster share the same $\Delta$.

**Anomaly Detection and May-Beliefs Mining**. Given a set of clusters $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$, we conduct a z-score-based analysis to identify anomaly clusters. Ideally, the z-score [13] of a cluster $C \in \mathcal{C}$ is

$$z_C = \frac{|C| - \mu_{\mathcal{C}}}{\mathbf{s}_{\mathcal{C}}},$$

where $\mu_{\mathcal{C}}$ is the average sizes of clusters in $\mathcal{C}$ and $\mathbf{s}_{\mathcal{C}}$ is the standard deviation. However, $|\mathcal{C}|$ can be quite small, and the anomaly clusters can largely affect $\mu_{\mathcal{C}}$ and $\mathbf{s}_{\mathcal{C}}$. Therefore, we replace $\mu_{\mathcal{C}}$ and $\mathbf{s}_{\mathcal{C}}$ with $\mu_{\mathcal{C}'}$ and $\mathbf{s}_{\mathcal{C}'}$, respectively, where the majority subset $\mathcal{C}' \subseteq \mathcal{C}$ is the smallest subset such that

$$\sum_{C' \in \mathcal{C}'} |C'| \geq \frac{3}{4} \sum_{C \in \mathcal{C}} |C|.$$

Following the common practice, a cluster $C \in \mathcal{C}$ is considered anomaly if $z_C \geq 3$ and $|C| < \mu_{\mathcal{C}'}$. Other clusters are accordingly considered as may beliefs.

## 4 IMPLEMENTATION

We implemented the deep state differential analysis algorithm as a prototype tool ODIN consisting of 14,362 lines of Kotlin code. We extensively used open-source tools in the implementation, and ODIN is also open-source available: bootstrapping automatically generated test inputs (traces) are obtained using APE [14] and ComboDroid [39]. Such a mixed bootstrapping is also inspired by existing work [36]. APE is also used to execute test inputs and obtain execution traces (GUI layout dumps at app states regarded quiescent by APE after launching and sending each event).

All implementation is consistent with the descriptions in Section 3. For performance considerations, during the calibration procedure (Algorithm 2) , if the trace of a generated input for $\sigma^\star$ does not reach $\sigma^\star$, ODIN still keeps it in the calibrated input set but does not count it when checking the number of test inputs for termination. Moreover, for each GUI model state and appended event pair, may beliefs are mined and multiple anomalies can be detected. ODIN outputs a single report containing all corresponding execution traces and clusters for further manual examination.

## 5 EVALUATION

Our evaluation aims to answer the following research questions:

- **RQ1 (Bug Finding, Section 5.2)**: How effective does ODIN automatically find non-crashing functional bugs in real Android apps comparing with state-of-the-art techniques?
- **RQ2 (Test Input Calibration, Section 5.3)**: How beneficial is input calibration in establishing beliefs and finding non-crashing functional bugs?
- **RQ3 (May-Belief Mining, Section 5.4)**: How beneficial is our may-belief mining algorithm in identifying non-crashing functional bugs?
- **RQ4 (False Positives, Section 5.5)**: How precise does ODIN report non-crashing functional bugs?
- **RQ5 (Bug Types, Section 5.6)**: What types and characteristics of non-crashing functional bugs can ODIN find?

### 5.1 Experimental Subjects and Setup

**Evaluated Apps**. We first collected 11 apps (latest version) used in the evaluation of existing Android testing/oracle work [14, 34, 39] as Group *Randoms*. We selected the top three largest (in LoC) among all available subjects: WIKIPEDIA, ANTENNAPOD, and ANKIDROID, and eight random subjects with at least 10K downloads and 4,000 LoC. These subjects are listed as the first group in Table 1.

To conduct a full comparison with the state-of-the-art work Genie [36], we also include *all* experimental subjects in the evaluation of Genie, excluding two non-functional apps (RADIODROID and SKUTUBE) due to unavailable Web services. For ANKIDROID, ANYMEMO, MARKOR, and TRANSISTOR, their latest versions are included in the first group of subjects. The remaining six subjects are listed as the second group named Group *Comparisons* in Table 1.

If an app's major functionalities cannot be accessed without a proper initial setup (e.g., user login), we provided the app a script to complete the setup. All evaluated techniques received exactly the same script, which runs automatically once the initial setup GUI is reached, to ensure a fair comparison. This is a common practice in Android testing [4, 5, 8, 14, 34, 39]. We did not mock any further functionality other than the initial setup script.

**Experimental Setup**. To answer **RQ1**, we compared ODIN with the state-of-the-art automated oracle Genie [36] on all 17 selected apps. Genie was configured with its default settings (same in its evaluation): one hour for mining a GUI transitional model and generating 20 initial test inputs with up to 15 events. Then, it mutated the test inputs (at most 4,500 mutated test inputs from one initial test input), executed them on 16 parallel Android emulators, and detected non-crash functional bugs. The exploration is terminated if accumulated wall-clock time exceeds 48 hours.

ODIN is given a same 48-hour time limit. Since ODIN requires massive test inputs to establish beliefs, we divide the 48-hour into 12 hours of test input generation (6 hours each for APE [14] and ComboDroid [39], each test input is 60-event long) and 36 hours for test input calibration and behavior manifestation on 16 parallel emulators. Runtime cost for anomaly detection is negligible. The calibration process terminates when ODIN obtains the same number of test inputs as the automatically generated ones, and the steps of a simulated random walk is also limited to 60. For each GUI model state $\sigma$, ODIN keeps 200 random inputs reaching $\sigma$. We compared the numbers of bug reports, true positive ones, and detected distinct non-crashing functional bugs of ODIN and Genie. We manually analyzed the GUI execution traces and related code of true bugs to determine their distinctness.

To answer **RQ2** and **RQ3**, we compared ODIN with two of its variants, namely ODIN-NOCALIB and ODIN-SIMPLE on the subjects of the *Randoms*. ODIN-NOCALIB does not conduct test input calibration, but directly samples 200 inputs for each GUI model state from automatically generated test inputs. On the other hand, ODIN-SIMPLE adopts a simple clustering and anomaly detection strategy. Specifically, for clustering it uses a fixed abstraction criteria that

**Table 1: Experimental subjects and comparison results with Genie**

| ID | Subject (Version, #Downloads, LoC) | Genie [36] | | Odin | | | | Comparison | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | #TP/#Report | 🐞 | #Input | #State (Cov) | #TP/#Report | 🐞 | Genie | Common | Odin |
| **Group Random: Random apps, latest version** | | | | | | | | | | |
| 1 | Wikipedia (2.7.50366, 10M–50M, 93404) | 35/112 (31.2%) | 1 | 5,015 | 261 (8.0%) | 136/381 (35.7%) | 1 | 0 | 1 | 0 |
| 2 | AntennaPod (2.0.0, 100K–500K, 262460) | 35/106 (33.0%) | 8 | 4,856 | 96 (24.0%) | 151/447 (33.8%) | 6 | 4 | 4 | 2 |
| 3 | AnkiDroid (2.15.4, 5M–10M, 66513) | 6/19 (31.6%) | 1 | 4,876 | 124 (20.2%) | 101/256 (39.5%) | 1 | 0 | 1 | 0 |
| 4 | Amaze (3.6.0, 100K–500K, 66126) | 72/152 (47.4%) | 1 | 5,621 | 103 (21.4%) | 231/683 (33.8%) | 2 | 1 | 0 | 2 |
| 5 | And-Bible (beta-539.3, 100K–500K, 20301) | 22/45 (48.9%) | 2 | 4,339 | 63 (34.9%) | 144/394 (36.5%) | 2 | 1 | 1 | 1 |
| 6 | AnyMemo (10.11.6, 100K–500K, 40152) | 20/91 (22.0%) | 1 | 5,022 | 147 (14.3%) | 62/203 (30.5%) | 1 | 0 | 1 | 0 |
| 7 | Markor (2.6.0, 50K–100K, 8356) | 31/69 (44.9%) | 1 | 5,388 | 87 (21.8%) | 138/403 (34.2%) | 1 | 1 | 0 | 1 |
| 8 | materialistic (3.3, 50K–100K, 38468) | 35/78 (44.9%) | 1 | 4,852 | 196 (8.7%) | 247/735 (33.6%) | 1 | 0 | 1 | 0 |
| 9 | Transistor (4.0.15, 10K–50K, 4925) | 15/21 (71.4%) | 1 | 4,362 | 83 (50.6%) | 86/238 (36.1%) | 2 | 0 | 1 | 1 |
| 10 | SkyTube (2.987, 100K–500K, 9615) | 16/38 (42.1%) | 2 | 4,961 | 235 (6.8%) | 216/807 (26.8%) | 2 | 1 | 1 | 1 |
| 11 | Aard2 (0.46, 10K–50K, 9622) | 8/23 (34.8%) | 1 | 4,471 | 68 (51.5%) | 47/145 (32.4%) | 1 | 1 | 0 | 1 |
| **Group Comparisons: Other Genie evaluated apps, Genie's evaluated version** | | | | | | | | | | |
| 12 | ActivityDiary (1.4.0, 1K–5K, 6966) | 31/69 (44.9%) | 7 | 4,335 | 77 (24.7%) | 115/310 (37.1%) | 3 | 4 | 3 | 0 |
| 13 | Tasks (6.6.5, 100K–500K, 46828) | 12/17 (70.6%) | 2 | 4,516 | 133 (16.5%) | 71/203 (35.0%) | 2 | 1 | 1 | 1 |
| 14 | UnitConverter (5.5.1, 1M–5M, 4167) | 50/106 (47.2%) | 2 | 5,013 | 179 (11.2%) | 126/246 (51.2%) | 2 | 1 | 1 | 1 |
| 15 | SimpleTask (10.3.0, 10K–50K, 3767) | 13/16 (81.2%) | 1 | 4,562 | 169 (13.6%) | 0/153 (0.0%) | 0 | 1 | 0 | 0 |
| 16 | Fosdem (1.6.2, 10K–50K, 8188) | 30/71 (42.3%) | 1 | 5,012 | 207 (9.2%) | 142/299 (47.5%) | 1 | 0 | 1 | 0 |
| 17 | MyExpense (3.0.9.1, 500K–1M, 77155) | 6/41 (14.6%) | 1 | 4,912 | 205 (8.8%) | 0/291 (0.0%) | 0 | 1 | 0 | 0 |
| **Summary** | | **26/63 (41.3%)** | **34** | **4,830** | **143 (18.2%)** | **118/364(32.4%)** | **28** | 17 | 17 | 11 |

[1] Column **Subjects** lists the information of each subject. For Genie and Odin, column **#TP/#Reports** displays the numbers of true positive/all reports for each subject and the TP rate in the brackets, and column 🐞 displays the numbers of detected distinct bugs. For Odin, the number of automatically generated test inputs and the number of GUI states in the mined model along with the state coverage (a state is considered covered if Odin finishes mining may beliefs for it) are displayed in column **#Inputs** and **#State (Cov)**, respectively. Finally, column Comparison plots the venn diagrams of the sets of bugs detected by Genie alone, Odin alone, and both, respectively.

removes all non-interactive widgets and sets the values of all attributes except *type* to ⊥ for all remaining widgets in the GUI layout. This is a common abstraction criteria adopted by existing techniques [5, 34, 36]. Moreover, for anomaly detection Odin-Simple directly uses the mean and standard deviation of all clusters instead of the ones of the majority subset to calculate z-scores. For Odin-NoCalib, we let it use the same automatically generated test inputs as Odin, and gave it 36 hours for behavior manifesting and anomaly detection. For Odin-Simple, we let it mine may beliefs and detect anomalies on all the GUI layout pairs produced by Odin with no time limitation. We compare the reports and detected distinct non-crashing bugs of these variants on the 11 randomly selected subjects of Group *Randoms*.

To answer **RQ4** and **RQ5**, we further analyzed Odin's reports and detected bugs, the corresponding execution traces, and the related code of the apps, to determine the root causes of false reports and the bugs, respectively. All experiments were conducted on a server running Ubuntu 18.04 LTS with 32-core AMD Ryzen 2990WX CPU, 128G RAM, and 16 Android 7.1.1 emulators.

## 5.2 Evaluation Results: Bug Finding

**Bugs Found**. The overall results in Table 1 show that Odin complements Genie in finding non-crashing functional bugs. Considering that only ~5,000 traces are used in the experiments due to the time limit, Odin would have potential to reveal even more non-crashing functional bugs given truly massive traces.

After eliminating false and duplicated reports, we found that Odin and Genie reported 28 and 34 distinct non-crashing functional bugs, respectively. 17 of the bugs are overlapping. The Venn diagrams in the last column of Table 1 displays the detailed results.

In the subjects of the *Randoms* and the *Comparisons*, Genie reported 20 and 14 functional bugs, respectively, while Odin reported 20 and 8, respectively. On average, Genie output 63 bug reports for each app, 26 (41.3%) of which were true, while Odin output 364 reports for each app, 118 (32.4%) of which were true. Odin reported more duplicated reports because there can be multiple GUI model states representing semantically similar GUI layouts that respond to (slightly) different sets of events, and from these GUI model states Odin identifies the same non-crashing functional bugs and reports each individually.

**Previously Unknown Bugs**. Odin and Genie are also complementary to each other in detecting previously unknowns non-crashing functional bugs. Among all true-positive bugs reported by Odin in the apps of the *Randoms* group, we reported five previous unknown bugs (listed in column **New** in Table 2) by excluding bugs already in the issue tracking system. Genie reported four previously unknown bugs (two in Amaze and two in SkyTube), two of which were also reported by Odin. Developers confirmed all these previously unknown bugs.

**Discussions**. For **RQ1** (bug finding), we argue that Odin complements Genie, and the major limitation of Odin is the unavailability of truly massive traces that thoroughly manifests app behaviors for all GUI model states.

For the 17 bugs detected by both Odin and Genie, we found that they indeed fall into the scopes of both tools. Specifically, we found

**Table 2: Comparison results between different variants of ODIN on the *Randoms***

| ID | Subject | Inputs | | ODIN | | | | ODIN-NOCALIB | | | ODIN-SIMPLE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | #Input | #State | Cov | #TP/#Report | 🐞 | New | Cov | #TP/#Report | 🐞 | #TP/#Report | 🐞 |
| 1 | WIKIPEDIA | 5,015 | 261 | 8.0% | 136/381 (35.7%) | 1 | | 7.3% | 99/290 (34.1%) | 1 | 0/142 (0.0%) | 0 (-1) |
| 2 | ANTENNAPOD | 4,856 | 96 | 24.0% | 151/447 (33.8%) | 6 | | 19.8% | 52/226 (23.0%) | 3 (-3) | 16/115 (13.9%) | 2 (-4) |
| 3 | ANKIDROID | 4,876 | 124 | 20.2% | 101/256 (39.5%) | 1 | 🐞 | 20.2% | 93/194 (47.9%) | 1 | 29/385 (7.5%) | 1 |
| 4 | AMAZE | 5,621 | 103 | 21.4% | 231/683 (33.8%) | 2 | 🐞 | 19.4% | 151/493 (30.6%) | 1 (-1) | 163/672 (24.3%) | 1 (-1) |
| 5 | AND-BIBLE | 4,339 | 63 | 34.9% | 144/394 (36.5%) | 2 | | 33.3% | 94/264 (35.6%) | 1 (-1) | 81/272 (29.8%) | 1 (-1) |
| 6 | ANYMEMO | 5,022 | 147 | 14.3% | 62/203 (30.5%) | 1 | | 16.3% | 52/164 (31.7%) | 1 | 0/181 (0.0%) | 0 (-1) |
| 7 | MARKOR | 5,388 | 87 | 21.8% | 138/403 (34.2%) | 1 | | 23.0% | 103/342 (30.1%) | 1 | 30/178 (16.9%) | 1 |
| 8 | MATERIALISTIC | 4,852 | 196 | 8.7% | 247/735 (33.6%) | 1 | | 6.6% | 147/644 (22.8%) | 1 | 0/103 (0.0%) | 0 (-1) |
| 9 | TRANSISTOR | 4,362 | 83 | 50.6% | 86/238 (36.1%) | 2 | 🐞🐞 | 53.0% | 75/191 (39.3%) | 2 | 31/108 (28.7%) | 1 (-1) |
| 10 | SKYTUBE | 4,961 | 235 | 6.8% | 216/807 (26.8%) | 2 | 🐞 | 5.5% | 168/651 (25.8%) | 1 (-1) | 0/203 (0.0%) | 0 (-2) |
| 11 | AARD2 | 4,471 | 68 | 51.5% | 47/145 (32.4%) | 1 | | 55.9% | 34/98 (34.7%) | 1 | 23/87 (26.4%) | 1 |
| **Summary** | | **4,888** | **133** | **18.0%** | **142/427 (33.3%)** | 20 | 5 | 17.3% | **97/323 (30.0%)** | 14 (-6) | **34/222 (15.3%)** | 8 (-12) |

[1] Column **ID** lists the id of each subject in Table 1. The numbers of generated test inputs and GUI model states in the mined model are displayed in column **#Input** and **#State**, respectively. For the three variants ODIN (with calibration), ODIN-NOCALIB, and ODIN-SIMPLE, column **Cov** lists the state coverage of the GUI model (ODIN-SIMPLE shares the same number as ODIN), column **#TP/#Report** displays the numbers of true positive/all reports for each subject and the TP rate in the brackets, and column 🐞 displays the numbers of detected distinct bugs. For the standard version of ODIN, column **New** additionally lists the number of detected previously unknown bugs.

that these bugs are rare of occurrences and lead to violations of metamorphic relations utilized by Genie.

For the 11 bugs detected by ODIN but not Genie, we found most (8/11, 73%) of them are beyond Genie's independent assumption. Two comparable GUI layout sequences can be obtained by (1) Executing a single functionality of the app, and (2) executing it sequentially after executing another functionality independent from it. Genie designs metamorphic relations between these sequences, and identifies violations of these relations caused by the incorrect inference between executing the two independent functionalities. Though effective, many bugs does not concern such incorrect inferences, and thus may escape Genie's detection. For instance, Genie failed to detect the motivating bug example in Figure 2. On contrast, ODIN successfully reported this previously unknown functional bug. The other three functional bugs missed by Genie require complex event sequences to manifest, which Genie failed to generate.

For the 17 bugs reported by Genie but not ODIN, many (9/17) of them were due to the corresponding GUI model states were not covered. A GUI model state is covered if ODIN finishes manifesting behaviors and mining may beliefs for it. As Table 1 shows, even with 16 emulators and 36 hours, ODIN could only cover a small portion of the GUI model states (18.2%) because executing a test input is extremely time-consuming [40].

As a qualitative and supplementary experiment, we gave ODIN sufficient time (108 hours) to cover all GUI model states on ACTIVITYDIARY (a relatively small app among all experimental subjects), and three of four missed bugs were correctly identified by the belief mining. Therefore, efficient test execution mechanisms [40] could be a potential research direction for enhancing ODIN. There are also 4/17 missed bugs because test input generators failed to manifest them. Therefore, more effective test input generators also benefits ODIN in bug finding.

Finally, we found a fraction of bugs (4/17) missed by ODIN were manifested by a considerable fraction of test inputs reaching the corresponding GUI model states, because calibration procedure did not generate enough additional inputs for these models, which hide deep in the GUI model. These bugs were thus regarded as normal behaviors instead of anomalies. This suggests that the calibration procedure may be further improved in the future.

## 5.3 Evaluation Results: Test Input Calibration

As shown in Table 2, if the calibration procedure is disabled (ODIN-NOCALIB), 6/20 (30%) bugs are missed, and we do not observe significant changes to the GUI model state coverage and true positive rate. All these missed bugs are are due to the skewed distributions of automatically generated test inputs: the erroneous GUI model states occurred too frequently in the traces to be identified as deviant behavior.

## 5.4 Evaluation Results: May-Belief Mining

As shown in Table 2, bug finding capability is significantly reduced if we adopt a trivial clustering algorithm (ODIN-SIMPLE) that adopts a fixed abstraction criteria and uses the mean and standard deviation of all clusters to calculate z-scores, indicating that our hierarchical clustering and anomaly detection algorithm is effective in mining may beliefs and detecting anomalies.

Specifically, 12/20 (60%) bugs are missed because either (1) ODIN-SIMPLE over- or under-abstracted the GUI layouts, and incorrectly clustered normal and anomaly behaviors together or put similar behaviors into different clusters, or (2) the clusters were too few, and the anomaly ones largely affected the z-scores.

## 5.5 Evaluation Results: False Positives and Duplicated Bug Reports

**False Positives.** As shown in Table 1, approximately 2/3 of ODIN's bug reports are false positives. Furthermore, there may also be duplicated bug reports on the same root cause. The root causes of false positives are:

(1) *Imprecise GUI layout model* (47%). We adopted the SwiftHand [8]'s algorithm for GUI state model construction, and semantically dissimilar states may be erroneously grouped

together. This is a fundamental limitation for any GUI layout-based approach [5, 14], and this limitation may be alleviated by developer-provided models.

(2) *Rare but normal behaviors* (32%). Some functionalities of an app are hidden deep and require a complex input to exercise, making our test input calibration procedure insufficient. Increasing the runtime of test calibration will yield more balanced test inputs for these states.

(3) *Unstable replay* (21%). How to stably replay an execution trace is another open challenge [24, 34]. Operations such as network accessing can have non-deterministic latent effect on the app's execution, leading to rare but correct behaviors. This can be tackled with more advanced trace replay techniques.

**Duplicated Reports**. We manually analyzed the GUI execution traces and related code of true reports to determine whether they are duplicated. As shown in Table 1, Odin can produce duplicated reports (∼ 100 bug reports revealing ∼ 1.6 distinct bugs per app).

**Discussions**. Despite the relative high false positive rates and the existence of duplicated reports, we believe that they do not significantly hinder the practical benefits of Odin for finding otherwise hard-to-detect bugs.

First, in our evaluation, Odin has already significantly narrowed down the scope for manual examination by filtering out ∼98.7% of the traces (∼28,000 traces were generated for each app, and there were ∼360 reports). We manually examined all the remaining traces (several hours for each app), resulting in ∼1.6 bugs per app. Considering that we found previously unknown bugs in well-tested apps, such an effort is worthwhile and reasonably moderate. For example, the file managing app Amaze has 50+ manually written regressions test cases which evolved over time. Nonetheless, Odin detected a previously unknown bug (shown in Fig 2) in its frequently used functionality, which can severely affect user experience. The developers quickly fixed the bug in the first revision after confirming our report, and explicitly documented it in the Changelog for v3.6.2.

Furthermore, there are opportunities to further reduce human labor:

(1) For practical usage, one can first fix a bug detected by Odin and then eliminate its duplicates by checking the remaining traces against the patched app. As bug duplication for non-crashing functional bugs is still a challenging open issue, this is the typical process currently adopted in practice [36].

(2) Better visualization of test cases (Odin used a simple visualization) and test case triaging can reduce the time for checking a trace. Checking a trace often takes less than one minute for a developer familiar with the app.

Such limitations may also be alleviated by future research on reducing false positives and duplication.

## 5.6 Evaluation Results: Bug Types

The 28 bugs found by Odin are categorized as follows:

(1) *Incorrect inferences between event handlers* (14/28, 50%). An app can be exercised in different scenarios, in some of which multiple (dependent or independent) event handlers that may incorrectly affect each other's execution can be invoked simultaneously or sequentially. For example, Transistor provides search suggestions if more than three characters are entered in the search bar. However, a race condition incorrectly results in app denial-of-service without a crash if a user quickly deletes the characters before search suggestions are returned.

(2) *Improperly handled data format* (8/28, 29%). An app can have functionally similar reactions to different data formats, and some case handler code may be buggy. For example, AntennaPod cannot properly process a subscribed podcast when its metadata is in a less popular CSV format, and incorrectly recognizes audios in the podcast as videos.

(3) *General coding mistakes* (4/28, 14%) Some non-crashing functional bugs are the results of general coding mistakes, e.g., third-party library misuse or incorrect program logic.

(4) *Incompletely implemented functionalities* (2/28, 7%). Due to tight development schedule, some rarely used functionalities of an app may leave unimplemented, e.g., untranslated texts on rarely used languages.

## 5.7 Discussions

**Finding Non-Crashing Functional Bugs**. Finding hidden non-crashing functional bugs in an app is far from trivial. Many bugs (including the five previously unknown ones) are from well-maintained apps, some are even from apps with extensive manual test cases (e.g., AnkiDroid contains over 200 manually written UI/unit test cases with assertions for correctness checking).

Considering the challenges even for experienced developers to find such non-crashing functional bugs, the runtime overhead (for generating and calibrating massive traces) and false positives could be acceptable for developers.

**Existing Non-Crashing Functional Bug Oracles**. Before Odin, non-crashing functional bugs can be automatically detected by differential-based metamorphic relations. Thor [1] and SetDroid [37] perturb a trace by injecting neutral event sequences. Genie [36] extends this idea by injecting a "likely independent" in-app operations. Metamorphic testing is a fundamentally different scope compared with Odin, and thus we compared only with the state-of-the-art technique Genie [36].

**Threats to Validity**. The representativeness of selected test subjects can affect the fidelity of our conclusions. To mitigate this threat, we selected additional evaluation subjects from popular benchmarks evaluated in existing work. These subjects are (1) large in size (around 76 KLoC on average), (2) well-maintained (containing thousands of revisions and hundreds of issues on average), (3) popular (all have 10K+ downloads), and (4) diverse in categories. Moreover, we selected the exact same versions of subjects used to evaluate Genie to provide a direct comparison.

The evaluated techniques (including Odin) involve randomness, and subjects may be non-deterministic. To mitigate this threat, the bootstrapping test input generation tools for both Genie and Odin were given sufficient time to cover almost all states they can explore. Moreover, we manually analyzed the reported bugs and identified that (1) most (8/11=73%) Odin-unique bugs are due to fundamental limitations of Genie, thus unlikely to be found by Genie on independent runs, and (2) many Genie-unique bugs

(9/17=53%) might be found by Odin because the specific GUI states were not explored (no clustering performed and thus no bug reports) before Odin's timeout. Therefore, we believe that randomness is not a primary threat to validity, and Odin and Genie are indeed complementary to each other.

The bug reports of Odin and Genie were manually analyzed to determine whether they are true positives. Moreover, we manually identified distinct detected bugs and their root causes. This may incur imprecision. To mitigate this threat, three authors of this paper conduct independent examination on all the reports, and cross-check to ensure correctness.

## 6 RELATED WORK

**Detecting Non-Crashing Functional Bugs in Android Apps**. A few pieces of work proposed for fully automatically detecting non-crashing functional bugs in Android apps without given oracles. Inspired by metamorphic testing [7], Both Genie and SetDroid design heuristic metamorphic relations between app execution results and cross-check for relation violations to detect non-crashing functional bugs. For instance, SetDroid utilizes the metamorphic relation that if one changes the system settings and immediately changes them back, the follow-up execution of the app should not be affected. Though effective, these metamorphic relations all have a strong emphasize on the independence of two event fragments, and many functional bugs fall out of these oracles' scopes. As our evaluation results show, Genie missed many bugs because they do not lead to relation violations. DiffDroid [12] on the Other hand, is inspired by differential testing [25] and cross-checks for execution inconsistencies on different devices. Similarly, bugs with consistent consequences across devices would escape DiffDroid's detection.

Our deep-state differential analysis does not rely on these symptomatic features of non-crashing functional bugs in Android apps. Inspired by the "bugs as deviant behaviors" [11] idea, it captures the *statistical features* of the bugs. Therefore, it can detect bugs with various symptoms and root causes. As our evaluation results demonstrate, our approach well complements existing techniques.

Most existing techniques still require manually written oracles, and mainly focus on enhancing these oracles' detecting abilities. Thor [1] and executes test suites in adverse conditions and check if the manually written assertions still hold. QUANTUM[44] utilizes manually provided GUI models as oracles and accordingly examines app behavior under specific user interactions. AppFlow [15] and ACAT [32] utilize machine learning techniques to combine test inputs and oracles from manually written ones for testing complex app functionalities. FARLEAD-Android [16] accepts GUI-level formal specifications as manually written Linear-time Temporal Logic formulas as oracles. Finally, AppTestMigrator [6] and CraftDroid [20] migrate test inputs and oracles from other apps to examine the functional correctness of the app under test. These techniques require tremendous manual efforts to provide required oracles or specifications, while Odin requires no manual guidance.

In conclusion, most existing techniques heavily rely on manually provided oracles and specifications to detect non-crashing functional bugs, while a few fully automated ones detect specific types of non-crashing bugs. This motivates the design of Odin.

**Detecting Non-crashing Functional Bugs in Traditional GUI-Based Softwares**. For traditional GUI-based programs such as web applications and desktop programs, manually provided oracles also play an essential role for detecting non-crash functional bugs. For instance, Memon et al. proposed a series of work [26–28, 42] that derive oracles for desktop programs from manually provided GUI models or specifications. These techniques cannot be directly applied to Android apps, and they still require manual guidance.

**The Must/May Belief**. The classic must/may belief was proposed by Engler et al. [11] for detecting functional bugs in operating systems as deviant behaviors. Be a generic methodology, the must/-may belief has been utilized in various research topics, including specification mining [3, 19, 33, 43], race detection [10, 22], fault localization [21, 29, 31], etc.

A few techniques infer must beliefs from manually written oracles for detecting non-crashing functional bugs in Android apps. For instance, Thor [1] infers must beliefs that if a test case passes the manually written assertions, inserting a neutral action (e.g., rotating the screen and back) into the test case should not change the outcome. To the best of our knowledge, we are the first to fully automatically infer may beliefs for detecting non-crashing functional bugs in Android apps without given oracles.

## 7 CONCLUSION

Leveraging the insight that a large number of traces obtained by executing automatically generated test inputs can reach a similar GUI layout, and only a small portion of them reach erroneous app states, this paper presents a generic, novel, and automatic oracle named deep-state differential analysis for detecting non-crashing functional bugs in Android apps by manifesting both normal and deviant app behaviors via extending calibrated test inputs, and clustering them to mine may beliefs and detect anomalies. We implemented our technique into a exploratory prototype Odin, and the evaluation results demonstrate that Odin can effectively detect non-crashing functional bugs in real-world Android apps, a considerable portion of them cannot be detected by existing techniques.

As a first exploratory work, the deep-state differential analysis technique provides an new direction for detecting non-crashing functional bugs. Based on the proof-of-concept prototype, a diverse range of technologies can be applied in the future enhancement of this technique. Promising research directions include utilizing information in the traces beyond GUI layouts, such as app logs and method invocation sequences, to calibrate automatically generated test inputs and mine may beliefs, and human-in-the-loop approaches to filter out false positive results.

## ACKNOWLEDGMENTS

This work was supported by the Natural Science Foundation of China under Grant Nos. 62025202 and 61932021, and the Leading-edge Technology Program of Jiangsu Natural Science Foundation under Grant No. BK20202001. The authors would like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Ting Su was partially supported by NSFC Project No. 62072178, and Ting Su and Zhendong Su were partially supported by a Google Faculty Research Award.

# REFERENCES

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic Execution of Android Test Suites in Adverse Conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. Association for Computing Machinery, New York, NY, USA, 83–93. https://doi.org/10.1145/2771783.2771786

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59. https://doi.org/10.1109/MS.2014.55

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon) *(POPL 2002)*. Association for Computing Machinery, New York, NY, USA, 4–16. https://doi.org/10.1145/503272.503275

[4] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-First Exploration for Systematic Testing of Android Apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA 2013)*. Association for Computing Machinery, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[5] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-Based Android GUI Testing Using Multi-Level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 238–249. https://doi.org/10.1145/2970276.2970313

[6] Farnaz Behrang and Alessandro Orso. 2019. Test Migration between Mobile Apps with Similar Functionality. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE 2019)*. IEEE Press, 54–65. https://doi.org/10.1109/ASE.2019.00016

[7] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2018. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 51, 1, Article 4 (jan 2018), 27 pages. https://doi.org/10.1145/3143561

[8] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA 2013)*. Association for Computing Machinery, New York, NY, USA, 623–640. https://doi.org/10.1145/2509136.2509552

[9] Vincent Cohen-addad, Varun Kanade, Frederik Mallmann-trenn, and Claire Mathieu. 2019. Hierarchical Clustering: Objective Functions and Algorithms. *J. ACM* 66, 4, Article 26 (jun 2019), 42 pages. https://doi.org/10.1145/3321386

[10] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles* (Bolton Landing, NY, USA) *(SOSP 2003)*. Association for Computing Machinery, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[11] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (Banff, Alberta, Canada) *(SOSP 2001)*. Association for Computing Machinery, New York, NY, USA, 57–72. https://doi.org/10.1145/502034.502041

[12] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*. 308–318. https://doi.org/10.1109/ASE.2017.8115644

[13] David A Freedman. 2009. *Statistical Models: Theory and Practice*. Cambridge University Press.

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

[15] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: Using Machine Learning to Synthesize Robust, Reusable UI Tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/3236024.3236055

[16] Yavuz Koroglu and Alper Sen. 2019. Reinforcement learning-driven test generation for android gui applications using formal specifications. *arXiv preprint arXiv:1911.05403* (2019). https://doi.org/10.48550/arXiv.1911.05403

[17] Kevin J. Lang, Barak A. Pearlmutter, and Rodney A. Price. 1998. Results of the Abbadingo one DFA learning competition and a new evidence-driven state merging algorithm. In *Grammatical Inference*, Vasant Honavar and Giora Slutzki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12. https://doi.org/10.1007/BFb0054059

[18] Xiujiang Li, Yanyan Jiang, Yepang Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2014. User Guided Automation for Testing Mobile Apps. In *Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014)*. 27–34. https://doi.org/10.1109/APSEC.2014.13

[19] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE 2005)*. Association for Computing Machinery, New York, NY, USA, 306–315. https://doi.org/10.1145/1081706.1081755

[20] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test Transfer across Mobile Apps through Semantic Mapping. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE 2019)*. IEEE Press, 42–53. https://doi.org/10.1109/ASE.2019.00015

[21] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical Model-Based Bug Localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE 2005)*. Association for Computing Machinery, New York, NY, USA, 286–295. https://doi.org/10.1145/1081706.1081753

[22] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. 2007. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP 2007)*. Association for Computing Machinery, New York, NY, USA, 103–116. https://doi.org/10.1145/1294261.1294272

[23] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[24] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-Objective Automated Testing for Android Applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2931037.2931054

[25] William M. McKeeman. 1998. Differential Testing for Software. *DIGITAL TECHNICAL JOURNAL* 10, 1 (1998), 100–107.

[26] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. 2003. What Test Oracle Should i Use for Effective GUI Testing?. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (Montreal, Quebec, Canada) *(ASE 2003)*. IEEE Press, 164–173. https://doi.org/10.1109/ASE.2003.1240304

[27] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. 2000. Automated Test Oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: 21st Century Applications* (San Diego, California, USA) *(ESEC/FSE 2000)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/355045.355050

[28] Rodrigo M. L. M. Moreira, Ana Cristina Paiva, Miguel Nabuco, and Atif Memon. 2017. Pattern-based GUI testing: Bridging the gap between design and quality assurance. *Software Testing, Verification and Reliability* 27, 3 (2017), e1629. https://doi.org/10.1002/stvr.1629

[29] Syeda Nessa, Muhammad Abedin, W. Eric Wong, Latifur Khan, and Yu Qi. 2008. Software Fault Localization Using N-Gram Analysis. In *Proceedings of the Third International Conference on Wireless Algorithms, Systems, and Applications* (Dallas, Texas) *(WASA 2008)*. Springer-Verlag, Berlin, Heidelberg, 548–559. https://doi.org/10.1007/978-3-540-88582-5_51

[30] Eric S Raymond. 2003. *The art of Unix programming*. Addison-Wesley Professional.

[31] Manos Renieris and Steven P. Reiss. 2003. Fault Localization with Nearest Neighbor Queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (Montreal, Quebec, Canada) *(ASE 2003)*. IEEE Press, 30–39. https://doi.org/10.1109/ASE.2003.1240292

[32] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2018. Automation of Android Applications Functional Testing Using Machine Learning Activities Classification. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems* (Gothenburg, Sweden) *(MOBILESoft 2018)*. Association for Computing Machinery, New York, NY, USA, 122–132. https://doi.org/10.1145/3197231.3197241

[33] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. 2007. Static Specification Mining Using Automata-Based Abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis* (London, United Kingdom) *(ISSTA 2007)*. Association for Computing Machinery, New York, NY, USA, 174–184. https://doi.org/10.1145/1273463.1273487

[34] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/

3106237.3106298

[35] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking Automated GUI Testing for Android against Real-World Bugs. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 119–130. https://doi.org/10.1145/3468264.3468620

[36] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully Automated Functional Fuzzing of Android Apps for Detecting Non-Crashing Logic Bugs. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 156 (oct 2021), 31 pages. https://doi.org/10.1145/3485533

[37] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and Finding System Setting-Related Defects in Android Apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 204–215. https://doi.org/10.1145/3460319.3464806

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

[39] Jue Wang, Yanyan Jiang, Chang Xu, Chun Cao, Xiaoxing Ma, and Jian Lu. 2020. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE 2020)*. Association for Computing Machinery, New York, NY, USA, 469–480. https://doi.org/10.1145/3377811.3380382

[40] Wenyu Wang, Wing Lam, and Tao Xie. 2021. An Infrastructure Approach to Improving Effectiveness of Android UI Testing Tools. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 165–176. https://doi.org/10.1145/3460319.3464828

[41] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An Empirical Study of Android Test Generation Tools in Industrial Cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 738–748. https://doi.org/10.1145/3238147.3240465

[42] Qing Xie and Atif M. Memon. 2007. Designing and Comparing Automated Test Oracles for GUI-Based Software Applications. *ACM Trans. Softw. Eng. Methodol.* 16, 1 (feb 2007), 4–es. https://doi.org/10.1145/1189748.1189752

[43] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE 2006)*. Association for Computing Machinery, New York, NY, USA, 282–291. https://doi.org/10.1145/1134285.1134325

[44] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *2014 IEEE 7th International Conference on Software Testing, Verification and Validation (ICST 2014)*. 183–192. https://doi.org/10.1109/ICST.2014.31

[45] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (dec 1989), 1245–1262. https://doi.org/10.1137/0218082