Empirical Software Engineering (2025) 30:110
https://doi.org/10.1007/s10664-025-10662-w

Detecting data manipulation errors in android applications using scene-guided exploration

Shuqi Liu[1] · Yu Zhou[1] · Wenhua Yang[1] · Taolue Chen[2] · Harald Gall[3]

Accepted: 21 April 2025

Abstract
Data manipulation functionalities (DMFs) refer to operations such as create, read, update and delete which are crucial for maintaining the integrity of Android application (app) data. Existing testing techniques often fail to explicitly verify DMFs in combination, having limited capability to uncover non-crashing data manipulation errors (DMEs). In this paper, we propose a novel approach SceneData, which utilizes a scene-guided exploration strategy to effectively detect DMEs in Android apps. Particularly, we model the UI pages as scenes and present a scene transition graph (SceneTG) to capture the relation between scenes. By utilizing SceneTG, we explore app states to thoroughly validate DMFs across different scenes. We evaluate SceneData on 17 popular real-world apps, and SceneData discovers 25 previously unknown bugs in their latest releases, 21 of which are non-crashing DMEs. The experiments also show that a significant fraction (15/21) of these bugs cannot be detected by the state-of-the-art techniques.

Keywords Scene-based exploration · Non-crashing functional bugs · Model-based testing · Android apps

Communicated by: Shaukat Ali

✉ Yu Zhou
zhouyu@nuaa.edu.cn

✉ Taolue Chen
t.chen@bbk.ac.uk

Shuqi Liu
liushuqi@nuaa.edu.cn

Wenhua Yang
ywh@nuaa.edu.cn

Harald Gall
gall@ifi.uzh.ch

[1] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China

[2] School of Computing and Mathematical Sciences, Birkbeck, University of London, London, UK

[3] Department of Informatics, University of Zurich, Zurich, Switzerland

Published online: 10 May 2025

1 Introduction

Android applications (apps) are integral to numerous daily activities, with millions available in app stores (Google 2024b; Fdroid 2024). The growing user demand has led to increasingly complex functionalities and user interface (UI) designs (Xiong et al. 2023). Android apps commonly perform data operations such as create, read, update and delete (CRUD) to manage app-specific data (e.g., adding new files, browsing news, or deleting files). These data operations are directly related to data integrity and availability, making them essential for maintaining an app's core functionality. Since data manipulation functionalities (DMFs) involve data storage, processing and presentation, failures in these operations may cause data inconsistency or unintended modifications, severely compromising the user experience. To ensure these DMFs work as expected, developers must conduct thorough testing before releasing the app (Jabbarvand et al. 2019; Su et al. 2021; Yang et al. 2022).

Despite significant progress in improving and automating GUI testing for Android apps,detecting non-crashing data manipulation errors (DMEs) remains a challenge. Unlike crash bugs that cause system failures, non-crashing DMEs lead to inconsistencies in data operations. These problems often persist silently without triggering obvious failures, making them difficult to be detected by the current testing techniques. Most existing GUI testing tools, e.g., Monkey (Monkey 2024), Sapienz (Mao et al. 2016) and Stoat (Su et al. 2017), are only capable of detecting crash bugs due to the lack of effective test oracles. Some recent testing tools employ metamorphic relations to mitigate the oracle problem, but most efforts have been made towards addressing specific issues, such as system settings (Sun et al. 2021, 2023b) and data loss (Guo et al. 2022; Riganelli et al. 2020). While Genie (Su et al. 2021) and Odin (Wang et al. 2022) can find generic non-crashing bugs, they struggle to identify logic bugs in data operations such as incorrect file renaming or failed updates.

Figure 1 demonstrates a real-world DME bug in the file management app *Markor*.[1] Renaming the recently viewed documents has a serious bug that subsequently makes the document inaccessible. Specifically, a user creates a file on the home page (Fig. 1(a)-(c)). After a series of operations, the user navigates to the recently-viewed documents page (Fig. 1(c)-(e)). If the user attempts to rename a document titled "To-do.md" to "Done.md", the app fails as the filename is not correctly updated to "Done.md" (see Fig. 1(e)-(h)). Even worse, when the user clicks on this document, a strikethrough appears on the filename, and the device yields no response (see Fig. 1(h)-(j)). In this example, a failure to rename a file in the recently-viewed documents is a typical example of non-crashing DME.

The recent tool PBFDroid (Sun et al. 2023a) leverages the consistency between the data model and the UI layout as an oracle for detecting DMEs. For instance, successfully displaying the "To-do.md" file as text in the GUI indicates the normal operation of the DMF "Create File", as described in Fig. 1(a)-(c). PBFDroid employs an exploration strategy that randomly interleaves related DMFs and other events. Despite its promising performance, two key drawbacks hinder its ability to uncover complex DMEs resulting from interactions between various data manipulations.

First, the same DMF can be executed on different UI pages. For example, the DMF "Rename file" can be performed on both the home page (Fig. 1(c)) and the recently viewed documents page (Fig. 1(e)) of *Markor*. Randomly executing operations may significantly reduce the chance of interacting with the widget highlighted in the red box in Fig. 1(c), which is necessary for navigating to the recently viewed documents page (Fig. 1(e)), where the error occurs.

[1] available at https://github.com/gsantner/markor/

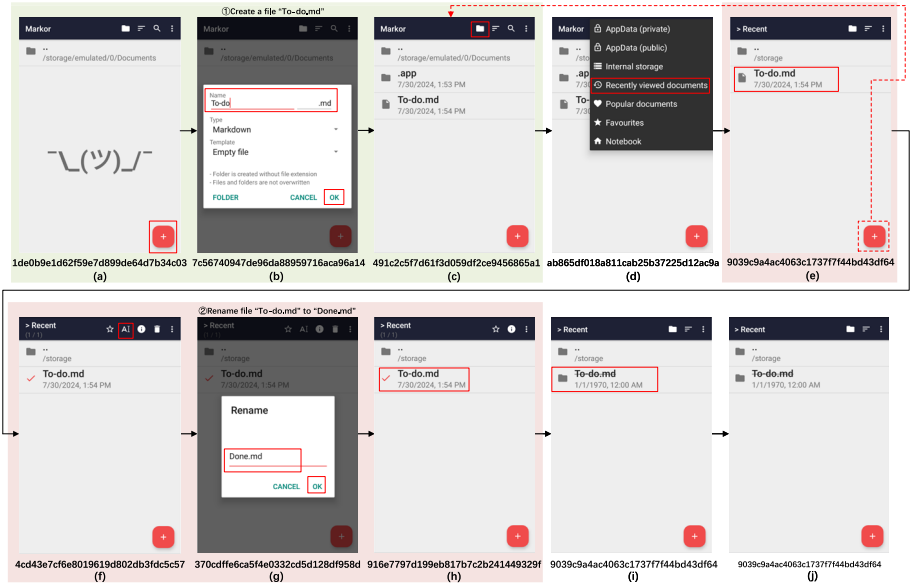

Fig. 1 A DME in app *Markor* (v2.12.0). The small red box on each page denotes a UI event and the string below each page screenshot represents the corresponding scene identifier

Second, multiple DMFs can be executed on the same UI page. For example, the recently viewed documents page in *Markor* (Fig. 1(e)) offers various functionalities that allow users to perform operations such as renaming, deleting and creating files. If the DMF “Create File" is randomly selected for testing (by clicking the add button on the current page), it redirects to the home page (Fig. 1(c)), which may interfere with the testing of other DMFs (e.g., “Rename File") on the recently viewed documents page, making it difficult to trigger the bug illustrated in Fig. 1.

These problems impose two technical challenges in testing app properties. (1) Determining the executable DMFs for the current UI page. UI pages consist of numerous UI views (widgets) that represent the user’s functional intentions. It is essential to determine which widgets on the current page satisfy the conditions for DMF execution. Due to the richness of UI pages in Android apps, relying solely on the DMF preconditions provided by humans to determine their executability may result in omissions. This hinders the comprehensiveness of DMF validation on a single UI page. (2) Enhancing mutual interaction between DMFs across different pages. The complexity of mobile apps often arises from the fact that they contain multiple distinct DMFs. They work collaboratively to complete data management and processing tasks. However, certain bugs only manifest under specific combinations of DMFs, involving interactions and execution sequences among different DMFs. In addition, the implementation of certain features often relies on specific events on the UI pages to trigger. In other words, due to the possible combinations of DMFs, dynamic exploration struggles to generate comprehensive tests to validate whether various app properties operate as expected.

Contributions In this paper, we propose SceneData, a novel approach that utilizes a scene-guided exploration strategy for effectively detecting DMEs in Android apps. Scene-guided exploration systematically models an app's UI pages as scenes and dynamically explores their transitions. Leveraging the scene representation, SceneData comprehensively examines DMF interactions within and across different states. In particular, we adopt a model based method, and leverage two techniques to overcome the aforementioned challenges.

We construct a GUI model to represent the app's behavior which is utilized to determine the executable DMFs on each app page. Specifically, we first model the UI pages as scenes by the dynamically captured widget hierarchy. Subsequently, a *scene transition graph* (SceneTG) is extracted by interacting with the widgets in these scenes to capture the detailed transition relation between scenes. Based on the constructed SceneTG, the local exploration of DMF-related widgets is performed in each explored scene to determine their suitability for DMF execution, enhancing the identification of executable DMF candidates.

For the DMF candidates identified for these unique scenes, a depth-first search (DFS) strategy is employed to generate meaningful test cases that cover diverse DMF combinations. This strategy ensures thorough testing of each scene and systematically explores DMF interactions across different scenes. During the iterative validation process of DMF operations, our strategy prioritizes other possible events that may trigger scene transitions to effectively interlink disparate GUI scenes.

We implement a prototype of SceneData and conduct comprehensive experiments to demonstrate the effectiveness of SceneData. Evaluation on 17 real-world Android apps shows that SceneData successfully identifies 25 previously unknown bugs, 21 of which are non-crashing functional bugs, and the remaining 4 are crashes. We reported these bugs to the developers, and so far, 14 have been confirmed. A detailed comparative analysis revealed that a significant fraction (15 out of 21) of these bugs could not be detected by the state-of-the-art technique PBFDroid. Furthermore, none of the 21 non-crashing defects are identified by Monkey, Genie, and Odin.

Structure The remainder of the paper is organized as follows. Section 2 introduces the background of our work. Section 3 elaborates on the details of SceneData. Section 4 describes the experimental setup, and Section 5 analyzes the experimental results. An extended discussion of our work is presented in Section 6. Related work is reviewed in Section 7, and Section 8 concludes this paper.

2 Background

2.1 Android UI Layout and Event

In Android app development, an Activity provides the main interface for user interaction. It represents an independent screen where users perform specific tasks such as browsing content, inputting data or making selections. User operations on these screens can trigger transitions between different Activities.

A Fragment is a versatile component that enables developers to break down complex Activities into smaller, more flexible units. This modularity not only improves code maintainability but also allows the app to adapt to devices (e.g., with various screen sizes) more naturally. By embedding multiple Fragments within a single Activity, each Fragment can independently

handle a portion of the interface. Furthermore, Fragments can be reused across multiple Activities, reducing code redundancy and ensuring a consistent user experience throughout the app.

An Android app is largely a GUI-based event-driven program. Each UI page is defined by its corresponding user interface (UI) layout file. The UI layout L is essentially of a tree structure composed of numerous UI views (or widgets). Each UI view $w \in L$ has attributes such as `class` (view type), `resource-id` (view id) and `text` (view text). UI views include user-visible leaf nodes (such as buttons or text fields), with their type being `Button` or `EditText`. They also include non-leaf nodes, which can be elements of type `ListView` or `ViewGroup`. These non-leaf nodes organize or display other views in a specific order.

Figure 2 illustrates a screenshot of UI page from an app *markor*, along with its corresponding UI layout. The leaf nodes in the UI layout are annotated with screenshots of the respective widgets on the screen. The file name "To-do.md" displayed on the screen in Fig. 2 can be uniquely identified by the following attributes and their values: {⟨`package`, "*net.gsantner.markor*"⟩, ⟨`class`, "*android.widget.TextView*"⟩, ⟨`resource-id`, "*net.gsantner.markor:id/opoc_filesystem_item__title*"⟩, ⟨`text`, "*To-do. md*"⟩, ⟨`clickable`, "*false*"⟩, ⟨`bounds`, "[158, 498][369, 565]"⟩}.

A UI event $e = (t, w, o)$ is a tuple where $e.t$ denotes the event type (e.g., click, edit, or a system event), $e.w$ denotes a specific UI element (e.g., a button or text field) that is directly interacted with by e, and $e.o$ denotes the optional data associated with e (e.g., the text inputted for editing). Systematically modeling UI layouts and events enables the analysis of how user interactions trigger transitions and update within UI pages, offering an abstract representation of app behavior.

Multiple ordered events constitute an event trace $E = [e_1, \ldots, e_i, \ldots, e_n]$, which implements a certain functionality of the app. Accordingly, the state of the app changes as events are executed, denoted as $s_0 \xrightarrow{e_1} s_1 \cdots s_{i-1} \xrightarrow{e_i} s_i \cdots s_{n-1} \xrightarrow{e_n} s_n$, where the state s_i is reached by executing event e_i on state s_{i-1}. We use $s_n = E(s_0)$ to denote the transition of the initial state s_0 to the final state s_n by executing the events sequence E. In DME detection, analyzing the initial state helps determine the appropriate DMF to execute on the current page, while examining the final state helps validate whether the DMF operation has been performed correctly and achieved its intended functionality.

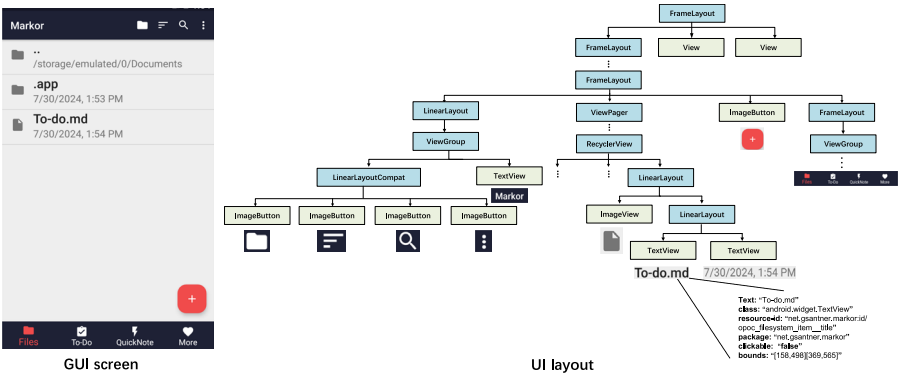

Fig. 2 The GUI screen of UI page captured from app *markor*. The UI layout hierarchy elements highlighted on the right correspond to the screenshot shown on the left

2.2 Model-based Properties

When a user performs DMF-related operations, it is crucial to ensure that the data displayed on the screen corresponds accurately to user's actions. To this end, we need to know (1) the UI layout of the app page and (2) the app data (e.g., file name) involved in the DMF operations. They are specified as an *app state* $s = \langle L, \mathcal{D} \rangle$, where L represents the UI layout and $\mathcal{D}$ represents the data.

Model-based property, first introduced by PBFDroid (Sun et al. 2023a), is employed to verify a DMF, ensuring it correctly implements the desired functionality. For a data manipulation operation *op* (i.e., *op* is *create*, *read*, *update*, *delete*, or *search*), its model-based property is defined as $\phi^{op} = \langle Pre^{op}, E^{op}, R^{op}, Post^{op} \rangle$. Here, Pre^{op} denotes the necessary precondition that must be satisfied before executing the event trace E^{op}, which represents the functionality F as an event trace involved in performing the DMF. R^{op} captures the data updates when each event e_i in E^{op} is executed. An abstract data model D can be introduced to keep track of the app data manipulated by DMF. Under R^{op}, D is updated to reflect the effect of the operation on data. The postcondition $Post^{op}$ describes the expected change in the app state after E^{op} is executed.

We use the example in Fig. 1 to explain how to perform property checking on DMFs. For the DMF "Create Folder" in app *Markor*, let the event path E^{create} be $[e_1, e_2, e_3]$. When E^{create} is executed starting from the initial state of the app (shown in Fig. 1(a)), the state changes can be tracked as $s_a \xrightarrow{e_1} s_b \ldots \xrightarrow{e_3} s_c$, where $s_i = \langle L_i, \mathcal{D}_i \rangle$ for $i \in a, b, c$. The abstract data model D is updated throughout the execution of E^{create}. Note that the precondition Pre^{create}, i.e., the UI view w_1 of the first event e_1 is in initial state s_a, determines the execution of DMF "Create Folder". After completing the execution of the three events, *Markor* switches to the state s_c (depicted in Fig. 1(c)). R^{create} records the data updates, adding "To-do.md" to D_c. Finally, the postcondition $Post^{create}$ checks that D_c appears in the layout L_c, confirming the successful execution of the DMF "Create Folder". Similarly, when multiple DMFs are executed, the same property checking method is followed. With the successful execution of DMF "Update Folder", the abstract data model D should be updated from 'To-do.md" to "Done.md".

Figure 3 illustrates how D is updated through the execution of the corresponding DMFs in Fig. 1. However, the final state s_h of the app records the data D_h that fails to include "Done.md" as expected. This discrepancy indicates that the DMF "update file" property is violated. Model-based properties provide an effective strategy for verifying DMF executions, overcoming the limitations of DME detection. By tracking the app's UI and data status throughout the execution process, it ensures consistency with expected behaviors. This concept lays the foundation of our work. Unlike the random exploration strategy adopted in PBFDroid, SceneData introduces a scene-guided exploration strategy that systematically navigates UI states and DMF interactions, allowing more thorough validation of these properties.

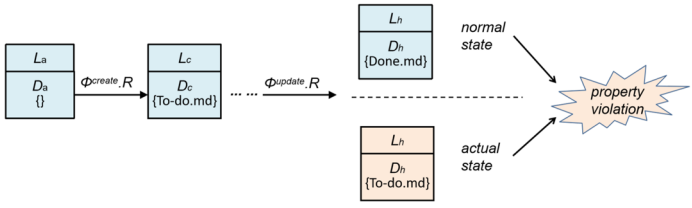

Fig. 3 Abstract app states shown in Fig. 1(a)-(h)

3 Approach

Figure 4 presents an overview of the proposed SceneData approach. In a nutshell, SceneData takes the app under test as input and attempts to reveal any DMEs with bug-reproducing test cases as output. It consists of three modules: (a) DMF Instantiation, (b) SceneTG Construction, and (c) Scene-Driven Exploration.

DMF Instantiation (Section 3.1) defines the DMFs of the app. These DMFs can be specified during the execution of the functionalities of interest. SceneTG Construction (Section 3.2) focuses on the collection of UI transition graph (UITG) for the app and its dynamic exploration. Specifically, we build UITG as the basis for SceneTG. We then collect Inter-Component Communication (ICC) messages to directly launch activities within the app. By identifying scenes within these launched activities and extracting the transition relation between different scenes, the initial UITG is enriched. Scene-driven Exploration (Section 3.3) encompasses filtering executable DMFs for each new scene and generating GUI tests for the app. For the former, all executable DMFs are filtered to construct a candidate set. If no candidate DMF is found, a widget on the current page is randomly selected to trigger a new scene. For the latter, the candidate DMFs are tested across different scenes by adopting a DFS strategy to generate test cases covering different DMF combinations. If new scenes are discovered during the dynamic exploration process, the former step is iterated to thoroughly validate the app state.

3.1 DMF Instantiation

DMF instantiation assists testers in defining DMFs for apps. Since app widgets may change in a newer version, we follow the steps introduced in PBFDroid (Sun et al. 2023a), in which tester's interactions are recorded and translated into DMF specifications using a domain-specific language in the JSON format.

Definition 1 (DMF specification) A DMF specification is given by $\langle EventTrace, Data, DataChange, View, Precondition, Postcondition \rangle$, defining the functional operations performed by the app.

- $EventTrace$ contains a sequence of events that execute the DMF, with each event having a type, a view and optional data.
- $Data$ defines the data objects involved in the DMF operation, specifically the data objects that are added and removed.

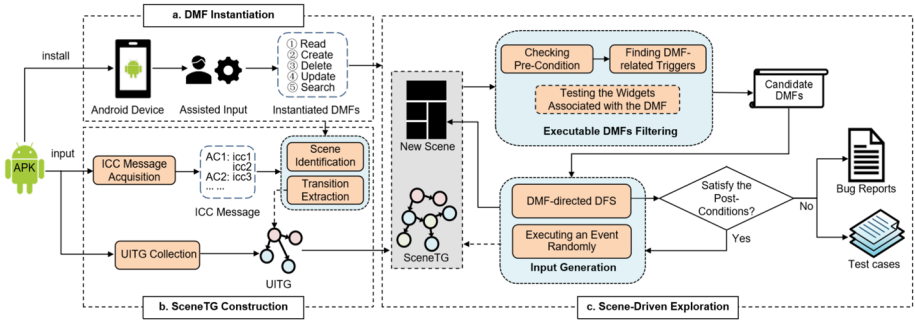

Fig. 4 The workflow of SceneData

– *DataChange* specifies the effects of data updates.
– *View* describes the UI views associated with the DMF. Each view can be identified by attributes such as `className`, `resourceId` and `text`, which are automatically collected based on the recorded events.
– *Precondition* defines the necessary conditions for executing the DMF. The event trace for the DMF can only be executed when the preconditions are met.
– *Postcondition* defines the properties that are valid after executing the DMF.

To generate a DMF specification, one needs to (1) **Specify the target DMF type.** The tester first needs to specify the DMF type to SceneData, whether it is create, delete, update, search or read. (2) **Capture interaction sequences.** As the tester interacts with the running app to achieve the specified functionality, an event trace is automatically generated to capture the sequence of interactions. Concurrently, DMF instantiation extracts the views of each UI widget participating in the interaction directly from the corresponding XML layout of the app page, ensuring an accurate representation of the UI widgets involved. (3) **Define data objects.** Based on the DMF type, the tester specifies the data objects to be manipulated using text input. For example, in the DMF "Update File", where the DMF type is "update", the tester manually specifies the old filename as the deleted data object and the new filename as the added data object.

It is important to note that when creating the DMF, the event trace for interactions with the app should ensure that the preconditions for the observed properties are met on the starting page. Additionally, the postconditions should be adhered to on the ending page. Following these steps, a DMF specification in the JSON format is automatically generated.

Example 1 Figure 5 illustrates the basic steps for generating the DMF specification of the "Update File" in the latest version (v2.12.0) of *Markor*, where the interaction sequence follows the definition provided by PBFDroid (Sun et al. 2023a).

In this case, the specified DMF type is "update". The user inputs are illustrated through operations across the app pages from L_0 to L_3 (as shown in Fig. 5), with each UI widget involved in the interaction marked with a red box. The process involves removing the

Fig. 5 Defining the DMF specification of "Update File" with the assistance of SceneData

editable text object “To-do.md" from page L_2, and adding the editable text object “Done.md" on page L_3. Consequently, the DMF specification for “Update File" is automatically generated, as depicted in Fig. 5. ①The event trace for the DMF “Update File" includes four events: e_1, e_2, e_3, and e_4. For instance, e_1 is a long-click event, with its target view annotated as w_1 on L_0. ②The DMF “Update File" involves one deleted data object and one added data object, specified by the text in view w_3 on L_2 and L_3, respectively. ③The data update effect for “Update File" is to delete the `removedDataObject` and add the `addedDataObject` in the app data. ④The execution of DMF “Update File" involves five views: w_1, w_2, w_3, w_4, and w_5. For example, w_1 can be identified by the resource ID ‘resoure-id=net.gsantner.markor:id/opoc_filesystem_item__title’. ⑤The Precondition for “Update File" includes the view w_1 on the starting page L_0. The Postconditions for “Update File" includes the view w_5 on the ending page L_4, with the text of view w_5 displaying `addedDataObject` instead of `deletedDataObject`.

3.2 SceneTG

To understand the specific UI states and their complex transition relation in the app, we construct a scene transition graph (SceneTG). This graph enhances scene-driven dynamic exploration by providing a reachable path from the current scene to another target scene, allowing more comprehensive and flexible validation of DMF-related behaviors. In a nutshell, a SceneTG connects different scenes through UI events, providing a detailed depiction of app behavior. The SceneTG of an app is formally defined as follows.

Definition 2 (Scene Transition Graph) A scene transition graph (SceneTG) is a tuple (S, E, δ) where S is a set of scenes representing the UI components of the app (e.g., activities, fragments, drawers, menus, and other UI pages); E is the set of events that app handles; the transition function $\delta : S \times E \rightarrow S$ describes the transitions between scenes. Each transition $\langle s_{i-1}, e_i, s_i \rangle \in \delta$ signifies a change from state $s_{i-1} \in S$ to state $s_i \in S$, triggered by the event $e_i \in E$.

To construct SceneTG, we follow three steps. First, we collect the UI Transition Graph (UITG) as the basis for building the SceneTG. Next, we collect the Inter-Component Communication (ICC) messages for activity launching. Finally, we dynamically explore the app to identify scenes within activities and their transition relationships.

UITG Collection The UITG of apps is commonly utilized to depict the interactions between various UIs triggered by typical operations.

Definition 3 (UI Transition Graph) A UI transition graph (UITG) is a directed graph $G = (V, E)$, where V is the set of nodes representing the app activities or fragments, and E is the set of edges representing the possible transitions between these nodes.

Many methods have been proposed for GUI modeling (Chen et al. 2019; Azim and Neamtiu 2013; Yan et al. 2020, 2022). In our approach, we use ICCBot (Yan et al. 2022) to construct an initial UITG for each app. UITG will be enriched by dynamic exploration and serves as the basis to construct the SceneTG.

ICC Messages Acquisition Android supports launching the intended activity using ICC messages through the console interface. Generating ICC messages typically involves creating an

Intent object that carries the necessary information needed to launch these components. This information includes basic attributes and extra parameters. Basic attributes, such as `action` and `category`, define the target of an intent and are usually specified in the intent-filter of the `AndroidManifest.xml` file. Figure 6 shows the exposed activity MainActivity and its attribute information as declared in the *Markor*'s manifest file. The intent-filter specifies that the implicit ICC must carry the values `android.intent.action.MAIN` for action and `android.intent.category.LAUNCHER` for category. Extra parameters ensure that activities receive the specific data needed to launch correctly, especially for those that depend on certain inputs. Since ICCBOT (Yan et al. 2022) infers ICC specifications by extracting component declarations and analyzing ICC messages related to sending and receiving, we utilize its `extras` settings to assign values to parameters.

Specifically, for an activity, we extract the `recvIntent` field from `extras` to retrieve the parameter name and its corresponding data type. Based on the required data type, we generate basic data structures such as strings, characters, or Boolean values. For example, if the parameter type is Boolean, we populate a value using the format "`-ez name False`", where `-ez` indicates that the parameter is of Boolean type, `name` represents the extracted parameter name, and `False` is the corresponding Boolean value. ICC messages ensure that the activities are launched with the necessary context and information for proper functionality.

Dynamic Exploration The extracted UITG focuses on the transitions between activities or fragments, resulting in a coarse-grained and potentially incomplete representation. In Android apps, a single fragment can contain multiple specific functionalities. Consequently, the UITG may not fully capture the interactions related to these functionalities and the page transitions they trigger. Namely, some pages with specific functionalities may not be explored. For example, a bug in *Markor*, as shown in Fig. 1, involves operations conducted entirely within MoreInfoFragment of MainActivity. However, UITG captures only high-level transitions between activities or fragments and fails to distinguish interactions occurring within the same fragment. As a result, transitions between different UI states within MoreInfoFragment, such as navigating between the home page (Fig. 1(a)) and the rename dialog (Fig. 1(b)), may not be adequately represented in the UITG, leading to an incomplete understanding of the app's behavior.

To address this issue, we consider the hierarchical structure of components on the UI page and build a more fine-grained GUI model, SceneTG, based on scenes. For each launched activity, the app state is identified as a scene, and the interactive widgets in each scene are explored exhaustively, which ensures that the transitions between different scenes are fully captured.

```
<activity android:exported="true"
  android:name="net.gsantner.markor.activity.MainActivity"
  android:launchMode="singleTop"
  android:taskAffinity=".activity.MainActivity">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

Fig. 6 Example of the manifest file with component declaration

Scene Identification When dynamically exploring an app, it is crucial to define the appropriate granularity of UI updates. We conduct an in-depth analysis of the existing start-of-the-art UI layout abstraction methods (Wang et al. 2022; Zhang et al. 2023; Lin et al. 2023) and adopt the following rules to model a UI page at the scene granularity. Specifically, scene identification assigns a unique scene identifier by abstracting the dynamically captured UI layout of the app page. (1) We focus only on nodes that match the target app package. System-level UI elements, such as the status bar and input method interface, may partially obscure the screen and hinder the recognition and processing of the target app's UI elements. (2) By combining the values of three key attributes of each widget, `resource-id`, `class`, and `package`, a unique identifier is generated using the MD5 hash algorithm (Rivest 1992). This strategy preserves the core features of the page layout by ignoring minor changes that do not affect the layout (such as text and color). (3) For adapter views such as `ListView` or `RecyclerView`, we only consider the structure of the first child view retrieved from `ListApdapter` (Google 2024a). Since the adapter views often contain repeated views, this strategy ensures that the unique identifier for the UI page remains accurate.

Example 2 As illustrated in Fig. 1, each screenshot of an app page is annotated with its corresponding identifier below. Two UI pages with the same identifier are considered to be the same scene. For instance, the pages shown in Fig. 1 (e) and (i) are considered as the same scene because they share the same identifier, differing only in the text rendering of the file name.

Transition Extraction Algorithm 1 details the process of extracting scene transition relation within the app to construct the SceneTG. In general, the transition extraction algorithm adopts a hierarchical strategy by dynamically exploring the app: at the activity level, it uses breadth-first search (BFS) to ensure coverage of all possible activities that can be launched through ICC messages (Lines 3-12); when delving down into a specific activity, it switches to depth-first search (DFS) to exhaustively explore all interactive widgets in the activity (Lines 14-27).

First, based on the ICC messages ICC_{all} obtained, each activity of the app is directly launched and traversed (Line 3). For successfully launched activities, the UI layout information of the initial page is abstracted to obtain the identifier s_t for scene identification (Lines 5-6). When new scenes associated with the current activity are encountered, their pages are explored (Lines 7-10).

SceneData utilizes the Android Debug Bridge (ADB) tool[2] to dump the current screen, which means it retrieves the current UI hierarchy as an XML file. It extracts widgets with the attribute 'clickable=true' from this XML file to identify all interactive widgets (Line 2, 16-17).

Relying solely on a DFS strategy to explore as many scenes as possible within a specific activity has two major limitations in dynamically exploring data manipulation related behaviors. First, each widget in the scene is triggered in sequence until no new scene is discovered, which will result in a large number of redundant operations. For instance, page (d) of Fig. 7 might be visited in the order of '$(a) \rightarrow (b) \rightarrow (c) \rightarrow (d)$', requiring multiple restarts and navigation back to page (d). In fact, it is possible to reach page (d) directly by triggering the last button of the bottom navigation bar on page (a). To minimize the time cost associated with redundant operations, we adopt a mechanism for storing and managing widgets to be visited in activities. This strategy ensures that each widget across different scenes is visited

[2] https://developer.android.com/tools/adb/

Algorithm 1 Transition Extraction Algorithm.

Input: Target app app, The corresponding ICC messages with all activities in the target app ICC_{all}, UI transition graph $UITG$, Instantiated DMFs of the target app t_{dmfs}
Output: Scene transition graph $SceneTG$

$S \leftarrow \emptyset$, $SceneTG \leftarrow UITG$
$w_{dmfs} \leftarrow getDMFSwidgts(t_{dmfs})$
for act,$icc \in ICC_{all}$ **do**
  $w_v \leftarrow \emptyset$
  if $Success(act, icc)$ **then**
    $s_t \leftarrow abstractStructureInfo()$
    if $s_t \notin S$ **then**
      $actions_t \leftarrow \emptyset$
      ExplorePage(s_t, S, w_v, w_{dmfs}, $SceneTG$, $actions_t$)
    end if
  end if
end for

function EXPLOREPAGE(s_t, S, w_v, w_{dmfs}, $SceneTG$, $actions_t$)
  $S \leftarrow S \bigcup s_t$
  $widgets_t \leftarrow getWidgets(s_t, w_v, w_{dmfs})$
  $w_v \leftarrow getVistetWidgets(widgets_t, w_v)$
  for $widget_t \in widgets_t$ **do**
    $s_t \leftarrow executeAction(app, actions_t)$
    $a_t \leftarrow generateAction(app, widget_t)$
    $s_{t+1} \leftarrow executeAction(s_t, w_t, a_t)$
    $SceneTG \leftarrow SceneTG \bigcup (s_t, (w_t, a_t), s_{t+1})$
    if $s_{t+1} \notin S$ and $len(actions_t) <= threshold$ **then**
      $actions_t \leftarrow recordNavigation(s_t, a_t)$
      ExplorePage(s_{t+1}, S, w_v, w_{dmfs}, $SceneTG$, $actions_t$)
    end if
  end for
end function

Fig. 7 Example of navigating to page (d). Page (d) is visited after navigating through pages (a), (b) and (c)

only once, significantly improving the efficiency of the exploration process. Second, sequential iteration of widgets on a page may prematurely trigger the removal of app data or fail to explore key data manipulations. To ensure comprehensive coverage of data manipulation-related behaviors, SceneData prioritizes the activation of widgets that are integral to the DMF in newly encountered scenes. By prioritizing these widgets, we aim to explore a wider range of DMF-related states early in the exploration process, thereby improving the effectiveness of exploration process.

SceneData traverses the widgets in the new scene for interaction (Lines 18-27). For each widget w_t that appears, the corresponding action a_t is assigned based on the characteristics defined in the UI layout (Line 20). The resulting triggered transitions are recorded as $s_t \xrightarrow{w_t, a_t} s_{t+1}$, where w_t, a_t represent the specific action a_t taken on the widget w_t (Lines 21-22).

The exploration terminates when a depth threshold is reached or no new scenes appear (Lines 23-25). In particular, whenever an action that triggers a new scene is discovered, it is recorded to facilitate initial navigation from the activity launch to the current page. If a widget initiates a transition that neither advances to an unexplored scene nor revisits an explored scene, SceneData reverts to the previous scene, ready to interact with other widgets (Lines 19,24).

3.3 Scene-driven Exploration

The input of the scene-driven exploration is the SceneTG, the app under test and the instantiated DMFs. It filters out executable DMFs for each newly identified scene and achieves a DMF-directed DFS strategy for these candidates, interacting with DMFs to discover DMEs.

Executable DMFs Filtering To test the DMFs-related behavior of an app, we need to determine which DMFs-related operations can be performed in the current state of the app. The model-based properties of DMFs define the prerequisites that must be met to execute the events defined in the instantiated DMFs. In some cases, the preconditions defined for an instantiated DMF might only be set on the app's initial homepage and are designed to align with the user's most common actions. This setup cannot validate the DMF behavior on other pages of the app, potentially limiting the ability to flexibly and comprehensively validate DMF-related behaviors.

For example, in the app *ActivityDiary*[3] that records daily activities, a user might see a visual add activity button on the page after searching for an activity and decide to add an activity. However, the instantiated DMF "Add Activity" might not recognize that the current page allows the addition of activities.

To address this issue, we design a filtering strategy for executable DMFs by leveraging the SceneTG, as described in Algorithm 2. This strategy allows to go beyond the execution paths defined by instantiated DMFs to identify all executable DMFs for each new scene, providing a more comprehensive validation of app properties (Line 5). The strategy consists of three key aspects.

1. **Checking pre-condition.** Similar to Sun et al. (2023a), SceneData checks whether the preconditions of an DMF are met based on the current app page layout and the simulated data.
2. **Finding DMF-related triggers.** SceneData first determines whether there is a widget defined in the instantiated DMF on the current page. If so, SceneData then examines the SceneTG to determine if there are any state transitions in the current scene triggered by these widgets.
3. **Testing widgets associated with DMF.** The SceneTG explored by SceneData may be incomplete and may not cover all new scenes. Since SceneTG is constructed via dynamic

[3] https://github.com/ramack/ActivityDiary

exploration of interactive widgets within launched activities, it prioritizes triggering widgets associated with DMF operations. However, executing a complete DMF often requires multiple widget interactions. For example, creating a file typically involves clicking the "New" button, entering a file name and confirming the creation. While depth-first exploration ensures that individual DMFs are executed, the vast number of possible widget interactions may hinder the exploration of additional new scenes, resulting in the potential incompleteness of SceneTG. If the current scene includes crucial widgets defined in the DMF, an exhaustive exploration strategy will be performed on these widgets to achieve comprehensive validation. When clicking a target widget triggers a scene transition, the newly discovered transition will be updated into the SceneTG. Since a new transition is triggered, the next step is to attempt to return to the previous scene using the system back button or by checking the SceneTG for a direct operation to go back. If both ways fail, try executing the shortest path algorithm on the SceneTG to return to the previous scene. Then, continue testing other target widgets on the current scene, transitioning to the next scene one by one until all widgets are tested.

By employing these strategies, if the executable DMF candidates $canDMFs$ are identified in the current scene, they are incorporated into the pool of candidate DMFs to be visited $toVisitDmfs$ (Lines 6-7). The DMFs in $toVisitDmfs$ will then be executed to explore the new scenes. This process helps to drill down into situations where different states of the app can execute the instantiated DMFs. During the filtration process of executable DMFs, if reverting to the preceding scene is not feasible, the current exploration step is aborted. SceneData then initiates widgets in the current state with the intention of navigating the app back to the prior scene (Lines 10-17). This mechanism involves the generation of inputs, which will be elaborated in the following subsection (Line 13). During this process, once the app state is in a new scene state, SceneData will filter the executable DMFs for the current scene (Lines 4,12).

Algorithm 2 Executable DMFs filtering.

1: **function** FILTERSCENEDMFS($eventTrace$, $Scene_{visited}$, w_{dmfs}, t_{dmfs}, $SceneTG$, $scene_{ep}$)
2: $canDMFs \leftarrow \emptyset$
3: $scene_{now} \leftarrow abstractStructureInfo(app)$
4: **while** *not* $canDMFs$ and $scene_{now} == scene_{ep}$ **do**
5: $canDMFs \leftarrow filterDMFs(t_{dmfs}, canDMFs, SceneTG, scene_{ep})$
6: **if** $canDMFs$ **then**
7: $toVisitDmfs[scene_{ep}] = canDMFs$
8: **break**
9: **else**
10: $Scene_{visited} \leftarrow Scene_{visited} \cup scene_{ep}$
11: $scene_{now} \leftarrow abstractStructureInfo(app)$
12: **while** $scene_{now} \in toVisitDmfs$ or $scene_{now} \in Scene_{visited}$ **do**
13: $eventTrace, e \leftarrow random(app, w_{dmfs})$
14: **if** $scene_{ep} == abstractStructureInfo(app)$ **then**
15: **break**
16: **end if**
17: **end while**
18: **end if**
19: **end while**
20: **end function**

Example 3 As illustrated in Fig. 1, *Markor* supports the following DMFs: Create, Delete, Update, Search, and View File. In an app state, the presence of an existing file is the precondition for performing the delete, update, search and view operations. Notably, the scenes shown in Fig. 1(c) and (e) support all DMFs, whereas the scene in Fig. 1(a) only allows the execution of the DMF "Create File". Moreover, the states illustrated in Fig. 1(i) and (j) share the same UI layout as Fig. 1(e). As a result, they are assigned the same scene identifier and thus allow the same set of executable DMFs.

Input Generation The input generation of SceneData works together with executable DMFs filtering to enable exploration of new scenes for the app (described in Algorithm 3) and is responsible for coordinating the testing process of DMFs-related behaviors. It generates and executes GUI tests that are designed to traverse various app scenes by combining different DMFs and other UI events through DFS strategies. The input generation operates in a compositional manner, similar to data generators in traditional property-based testing (Claessen and Hughes 2000). It generates random type events, selects random target widgets and assigns random strings to edit type events according to the designed rules. These ordered events constitute a GUI test. During the scene-driven dynamic exploration of the app, SceneData starts the abstract data model $\mathcal{D}$ and the executed events eventTrace (Line 6). It starts the app by clearing the data and then enters a loop to generate GUI tests with a maximum sequence length $threshold_1$ (Lines 4-10). This loop continues until a predetermined test time is exceeded (Line 2). In this loop, SceneData recursively passes each new scene in the app through the filtering and execution of the DMFs to validate app properties (Line 8).

For each new scene of the app, SceneData first evaluates which DMFs are executable based on the current state of the app (Line 14). The function *FilterSceneDmfs* verifies whether the conditions for executing the DMF are satisfied. When a DMF is randomly selected from the set $toVisitDmfs$ of DMFs to be visited in the scene $scene_{ep}$, SceneData executes its event trace and records the results (Lines 17,18). If the execution is successful, the abstract data model $\mathcal{D}$ is updated accordingly and the postconditions are checked against the UI layout L and the data model $\mathcal{D}$ (Lines 20-23). If a property violation occurs, the event sequence that records all the executed events is saved as a test case to reproduce the error (Line 24). If the DMF cannot be executed, the data model remains unchanged and the test continues. After executing the selected DMF, SceneData updates the scene to be visited next and the corresponding DMFs (Line 27).

The current state of the app may not align with the intended exploration scene (Lines 28,29). SceneData accounts for two distinct situations: Firstly, if the app is detected to be in a completely new scene, SceneData will prioritize invoking function *ExploreScene* to explore this novel context (Lines 30,31). This process will continue until no executable DMFs remain in the new scene. Secondly, for scenes that have already been explored, SceneData adopts a different strategy by selecting an executable event from the current UI layout to trigger based on the SceneTG (Line 33). SceneData works in this way until the current scene matches the scene to be explored, making it easier to explore various states of the app.

SceneData focuses on interactive widgets within the UI, as the act of clicking these widgets is often crucial for triggering state transitions in the app. Specifically, to thoroughly explore a scene, SceneData first refers to the SceneTG for events recorded in the current scene, seeking out widgets that have not yet been explored, particularly those that do not exist in the DMF operations. If no such a widget is found, SceneData randomly selects an event from all operable widgets to trigger a state change. Based on the types of widgets defined in the UI layout, SceneData randomly selects the type of target event and determines the target widget that can be triggered accordingly. By employing a DFS exploration strategy, SceneData can

Algorithm 3 Scene-driven exploration.

Input: Target app app, instantiated DMFs of the target app t_{dmfs}, scene transition graph $SceneTG$, dmfs-related widgets w_{dmfs}

Output: The found DMEs $DMEs$

$Scene_{visited} \leftarrow \emptyset, scene_{ep} \leftarrow NULL$
while *not timeout* **do**
  $eventTraces \leftarrow \emptyset$
  ClearAndRestartApp(app)
  while $len(eventTrace) < threshold_1$ **do**
    $D \leftarrow \emptyset, eventTrace \leftarrow \emptyset$
    $s_{init} \leftarrow abstractStructureInfo(app)$
    $eventTrace, DMEs \leftarrow ExploreScene(D, eventTrace, Scene_{visited}, w_{dmfs}, t_{dmfs}, SceneTG, s_{init})$
  end while
  $eventTraces \leftarrow eventTraces \cup eventTrace$
end while

function ExploreScene($D, eventTrace, Scene_{visited}, w_{dmfs}, t_{dmfs}, SceneTG, scene_{ep}$)
  $toVisitDmfs \leftarrow FilterSceneDmfs(eventTrace, Scene_{visited}, w_{dmfs}, t_{dmfs}, SceneTG, scene_{ep})$
  if $scene_{ep} \in toVisitDmfs$ **then**
    while $len(toVisitDmfs[scene_{ep}]) > 0$ **do**
      $dmf, toVisitDmfs[scene_{ep}] \leftarrow randomSelect(toVisitDmfs[scene_{ep}])$
      $succ, events \leftarrow execute(app, dmf.E)$
      $eventTrace \leftarrow eventTrace \cup events$
      if $succ$ **then**
        $D \leftarrow updateAppData(dmf, D)$
        $L \leftarrow dumpUILayout(app)$
        if $\neg isPostconditionHold(dmf, L, D)$ **then**
          $DMEs \leftarrow DMEs \cup eventTrace$
        end if
      end if
      $toVisitDmfs, scene_{ep} \leftarrow update(toVisitDmfs)$
      $scene_{now} \leftarrow abstractStructureInfo(app)$
      while $scene_{now} \neq scene_{ep}$ and not $toVisitDmfs$ **do**
        if $newscene(scene_{now})$ **then**
          ExploreScene($D, eventTrace, Scene_{visited}, w_{dmfs}, t_{dmfs}, SceneTG, scene_{ep}$)
        else
          $eventTrace, e \leftarrow random(app, w_{dmfs})$
        end if
        $scene_{now} \leftarrow abstractStructureInfo(app)$
      end while
    end while
  else
    ExploreScene($D, eventTrace, Scene_{visited}, w_{dmfs}, t_{dmfs}, SceneTG, scene_{ep}$)
  end if
end function

selectively validate the app properties, effectively avoiding unnecessary repeated exploration of DMFs within the same scene.

Example 4 As shown in Fig. 1, as the dynamic exploration progresses, the app *Markor* transits to the UI state depicted in Fig. 1(c), which is identified as a new scene. In this scene, five DMFs are available for execution, making it as the next scene to be explored, denoted as $scene_{ep}$. If SceneData randomly selects the DMF “Rename File", it follows the execution

process from Fig. 1(e) to (g), during which SceneData records the newly added data object "Done.md". After execution, SceneData checks whether the newly added data object appears in Fig. 1(h). Due to data inconsistency which deviates from the DMF defined in Fig. 5, this indicates that the execution of the DMF "Rename File" has failed.

At this point, the current state (Fig. 1(h)) is identified as a new scene. However, since no executable DMFs remain in this scene, SceneData applies a random strategy to select a widget from the current scene. For example, it may randomly click a button highlighted with a red box in Fig. 1(h). The resulting scene remains identical to that of Fig. 1(e), meaning the scene has not changed from $scene_{ep}$, and another executable DMF is randomly selected from the remaining options in the current app state for execution. If the scene remains unchanged after executing all DMFs in $scene_{ep}$, SceneData will once again apply a random strategy to select and execute a widget from the current scene.

4 Experiment Design

This section presents the experiment design following the guidelines (Wohlin et al. 2012). SceneData is implemented as an automated GUI testing tool for finding DMEs, builds upon and extends several existing tools. The Apktool[4] tool is utilized to extract the original `AndroidManifest.xml`, resource files, and source code from the APK files. The static analyzer component is built on top of the data-flow framework Soot (Vallée-Rai et al. Vallée-Rai et al. 2010) and ICCBot (Yan et al. 2022) to construct UITG. UIAUTOMATOR2[5] is used to extract GUI hierarchy files from UI pages. The Android Debug Bridge (ADB)[6] facilitates the launching of app activities and the sending of UI events, while LOGCAT[7] is used to log runtime exceptions.

4.1 Research Questions and Hypotheses

To evaluate the effectiveness of the SceneData in detecting DMEs, we consider the following three research questions (RQs).

- **RQ1:** How effectively does SceneData detect DMEs in real-world Android apps?
- **RQ2:** To what extent does SceneData complement the state-of-the-art techniques in detecting DMEs?
- **RQ3:** How effective are ScenceTG and Scene-Driven Exploration in improving the testing capability of finding DMEs? How does test sequence length impact the DME detection capability of SceneData?

To guide our investigation, we have the following hypotheses.

- **H1:** SceneData will effectively detect DMEs in real-world Android apps, and will provide insights into their types and characteristics.
- **H2:** SceneData will complement current state-of-the-art techniques by detecting additional DMEs.

[4] https://apktool.org/

[5] https://developer.android.com/training/testing/other-components/ui-automator/

[6] https://developer.android.com/tools/adb/

[7] https://developer.android.com/tools/logcat/

– **H3:** The integration of ScenceTG and Scene-Driven Exploration will significantly enhance the effectiveness of DME detection. Additionally, different test sequences will impact SceneData's effectiveness and efficiency in detecting DMEs.

In **RQ1**, we aim to demonstrate the performance of SceneData in finding DMEs, and analyze the types and characteristics of these DMEs. We hypothesize **H1**, which asserts that SceneData will successfully detect a significant number of previously undetected DMEs and offer detailed insights into their types and characteristics. **RQ2** compares SceneData with state-of-the-art techniques to assess the extent to which these existing approaches can detect the DMEs uncovered by SceneData. **H2** posits that SceneData will detect additional DMEs that are not identified by current state-of-the-art techniques, complementing existing approaches. **RQ3** is designed to employ two variants of SceneData to investigate the effect of incorporating SceneTG and Scene-Driven Exploration on the effectiveness of DME detection, as well as to evaluate the impact of the test sequence length by configuring four different event limits per test. **H3** predicts that the use of these strategies will significantly improve SceneData's ability to detect DMEs, while different test sequences affect the effectiveness and efficiency of DME detection.

4.2 Datasets

App Subjects PBFDroid (Sun et al. 2023a) is currently the only automated testing tool specifically designed for detecting non-crashing DMEs, and is closely related to this work. Therefore, to ensure comparability, we select the 17 real-world open-source Android apps used in that study as our experimental subjects. We collect these 17 open-source apps from GitHub[8] the detail of which is presented in Table 1, including the latest app version available at the time of our study, the number of stars on GitHub (#Stars), the number of installations on Google Play[9] (#Installations) and the main app features (#DMFs). These apps are selected to ensure a diverse range of functionalities, covering various categories, and mostly have a substantial number of installations. They have been widely applied in state-of-the-art research (Su et al. 2021; Wang et al. 2022) on non-crashing functional bug detection, providing a fair evaluation environment.

DMF Specifications To establish the DMF specifications for each app, we recruited three graduate students in software engineering, each possessing foundational knowledge of Android apps. Each participant was assigned all 16 open-source apps, ensuring that the DMF specifications for each app were obtained from three participants. Rather than introducing new DMF specifications, we strictly adhered to those defined by PBFDroid, focusing on replicating and validating them through the same process. Specifically, we selected app *Markor* as a reference app and recorded a tutorial video demonstrating the complete process of generating DMF specifications, providing clear guidance on the necessary steps. To ensure accuracy, participants utilized PBFDroid's instantiation helper to generate and validate each DMF specification. The time spent on each DMF replication was recorded. After discussion, the final DMF specifications for DME detection were determined. This process ensured consistency with PBFDroid while minimizing manual variations in DMF definitions.

[8] https://github.com/

[9] https://play.google.com/store/apps

Table 1 Open-source app subjects evaluated in our study

App Name	Version	#Stars	#Installations	App Feature	Target Data (#DMFs)
Markor	2.12.0	3.5K	100K-500K	Text Editor	File(5)
Aard2	0.56	419	10K-50K	DictionaryReader	Word(4)
SimpleTask	10.9.3	475	10K-50K	Task Manager	Task(4)
SkyTube	2.988	2.2K	100K-500K	Video Player	Channel(5)
AnyMemo	10.11.7	152	100K-500K	Learning Software	Card(7)
Amaze	3.10	5.1K	1M-5M	File Manager	Folder(7)
AnkiDroid	2.18.0	8.1K	10M-50M	Learning Software	Card(7)
Wikipedia	2.7.50489	2.2K	50M-100M	Wikipedia Reader	Favorite(5)
Tasks	13.8.1	3.3K	100K-500K	Task Manager	Task(5)
RadioDroid	0.86	689	100K-500K	Radio Manager	Radio(5)
ActivityDiary	1.4.2	73	1K-5K	ActivityRecorder	Activity(5)
MyExpenses	3.3.7	442	1M-5M	Expense Tracker	Account(5)
Antennapod	2.7.1	4.6K	500K-1M	Podcast Manager	Podcast(5)
Materialistic	3.3	2.2K	100K-500K	News Browser	Story(4)
Notepad	3.0.3	321	500K-1M	Note Manager	Note(5)
Transistor	4.1.1	420	10K-50K	Station Browser	Station(4)
OmniNotes	6.3.1	2.7K	100K-500K	Note Manager	Note(4)

Defining DMFs for an app involves determining both the data type of DMF and the corresponding data manipulation operations. For each target app, a primary data type is identified to represent its core data management functionality, ensuring frequent user interaction. For example, in *AnyMemo*,[10] the primary data type is “card", which corresponds to flashcards used for memorization, while in *Markor*, it is “file", referring to text documents for note-taking and editing. For the selected data type, we define DMFs for data manipulation operations *create*, *read*, *search*, *update*, and *delete*. The first three operations are fundamental for interacting with the data type, while *update* and *delete* are included as they modify or remove data previously created.

In Table 1, the last column lists the selected data types and their corresponding number of DMFs. The number of DMFs for an app is determined by its specific features. For instance, the app *Amaze*[11] includes the target data type “folder", with features such as hide and unhide, affecting data changes within the app. It is defined to contain seven instantiated DMFs, including “Create Folder", “View Folder", “Rename Folder", “Delete Folder", “Hide Folder", “Unhide Folder", and “Search Folder". Some apps (e.g., *OmniNote*[12]) might have fewer than five DMFs due to their less extensive functionality. Selecting a single data type per app for the study is sufficient to demonstrate the effectiveness of SceneData (see the results of RQ1). SceneData is highly scalable and can be extended to accommodate more data types and DMFs, enabling the identification of more DMEs.

[10] https://github.com/helloworld1/AnyMemo

[11] https://github.com/TeamAmaze/AmazeFileManager

[12] https://github.com/federicoiosue/Omni-Notes

4.3 Experiment Setup

Experiment Setup of RQ1 SceneData constructs SceneTG to assist in the dynamic exploration of apps. In the SceneTG construction module, following the setup in SceneDroid (Zhang et al. 2023), a state-of-the-art GUI modeling tool, we set a timeout of 15 minutes for the UITG collection and ICC messages acquisition phase, and a timeout of 30 minutes for the dynamic exploration phase for each app. For the scene-driven exploration module, we allocated 6 machine hours of testing time per app and set a maximum number of 200 events for generating a GUI test. These two parameters are determined based on the findings of RQ3, where we analyzed the effect of different sequence lengths on DME detection effectiveness. For each app bug report generated, we conducted a detailed manual review, combining captured screenshots and recorded event triggers for each step, to identify and categorize bugs based on different trigger conditions and manifestations. For each unique bug, we provided app developers with a detailed report that includes concise yet essential reproduction steps. These steps capture the key interactions leading to the errors, along with screen recordings of the error occurrence, facilitating rapid diagnosis and resolution. We actively incorporated developer feedback to ensure the authenticity of the reported issues.

Experimental Setup of RQ2 We select four state-of-the-art GUI testing tools as baselines, i.e., Monkey (Monkey 2024), Genie (Su et al. 2021), Odin (Wang et al. 2022) and PBFDroid (Sun et al. 2023a).

- Monkey (Monkey 2024). Monkey is a state-of-the-art automated random testing tool for testing the stability of Android apps and their ability to respond to random user interactions. It sends pseudo-random events to the app under test.
- Genie (Su et al. 2021) and Odin (Wang et al. 2022). These two tools focus on detecting non-crashing bugs through two different test oracles. Genie first generates seed tests that validate certain properties of the app, and then creates mutation tests that preserve the app properties from the seed tests based on independent view properties. It uses mutant tests to detect non-crashing logic bugs by violating properties. Odin builds GUI model by analyzing UI actions and then automatically expands those actions until the expected app state is achieved. It clusters UI behavior and quickly identifies abnormal behavior patterns from normal behavior.
- PBFDroid (Sun et al. 2023a). PBFDroid explicitly considers DMF during testing to find non-crashing bugs, which is the closest to SceneData. It randomly interleaves different DMFs and other possible events to explore the state of the app, and utilizes user-specified model-based attributes as the test oracle to find DMEs.

To thoroughly test each app, we allocate 48 hours of runtime for each tool within the same experimental environment. This execution time setup is aligned with PBFDroid (Sun et al. 2023a), the cloest approach to ours, ensuring a fair and consistent comparison. Based on the previous study (Wang et al. 2020), we set an event interval of 200 milliseconds for Monkey. Genie, Odin and PBFDroid are configured to the default values in the original papers (Su et al. 2021; Wang et al. 2022; Sun et al. 2023a). We allocate the same time for SceneData to test the 17 open-source apps. Specifically, in the SceneTG construction module, we allocate 15 minutes for the UITG collection and ICC message acquisition, and 30 minutes for the dynamic exploration phase. The remaining time is dedicated to the scene-driven exploration module.

Experiment Setup of RQ3 With the guidance of the SceneTG, SceneData explores different app scenes by traversing different DMFs and intersecting other possible events through the DFS strategy. Therefore, we study how the SceneTG and the scene-driven exploration strategy affect the SceneData's ability to find bugs. Specifically, we set up two variants for comparison:

- SceneData-NoGraph. SceneData-NoGraph refrains from constructing a SceneTG, nor does it participate in SceneTG related operations during the scene-driven exploration phase. Specifically, when filtering executable DMFs SceneData-NoGraph directly tests widgets associated with DMF after checking the preconditions. If a new page is encountered during the exploration process, SceneData-NoGraph only attempts to navigate back to the actively explored scene through the "home" button. During the input generation phase, only widgets defined by the UI layout that are unrelated to DMF are considered as candidates for randomly selecting an event to trigger in the current scene.
- SceneData-Simple. SceneData-Simple employs a simple DMF execution strategy for each scene in the input generation phase, which only randomly interleaves the DMF of the scene and other possible events to explore the state of the app. Namely, if there are candidate DMF in the scene, a single DMF or a widget defined by the UI layout that are unrelated to DMF is randomly selected to interact with the app.

We focus on evaluating the ability of the two variants of SceneData to generate an equal number of test cases for detecting DME under different experimental settings. Specifically, for SceneData-NoGraph, we set it to generate 40 GUI tests during functional testing, each of which contains a maximum of 200 events. For SceneData-Simple, we allocate a 45-minute timeout to construct the SceneTG, of which 15 minutes are used for the UITG collection and ICC message acquisition and 30 minutes for dynamic exploration. Afterwards, similar to SceneData-NoGraph, in the scenario-driven exploration phase we set it to generate 40 GUI tests with a maximum of 200 events.

Since the maximum event length $threshold_1$ (cf. Algorithm 3, Line 5) of each GUI test constrains DMF execution and its interleaving with other events, it may impact the effectiveness of SceneData in detecting DMEs. To assess this effect, we set the default event length to 200 and conduct experiment with three different configurations (i.e., 100, 300 and 400 events per test). All the other experimental conditions remain unchanged. We then compare the number of detected DMEs to analyze the impact of different event lengths.

Execution Environment All experiments are conducted on a machine equipped with Intel Core i7-12700 CPU @2.10 GHz, 32 GB of memory, and Windows 10 OS. We use the official Android emulator as the experiment device, configured to simulate a Pixel XL running Android 8.0. Each emulator is set up with 2GB memory, an x86 system image accelerated by KVM, and runs the Oreo version (API level 26).

5 Experimental Results

5.1 RQ1: How Effectively Does SceneData Detect DMEs in Real-World Android Apps?

For RQ1, we evaluate the ability of SceneData to detect DMEs in real-world Android apps. We carefully examine the output of SceneData for each app, focusing on runtime logs that indicated the presence of DMEs. Specifically, we analyze the execution of DMFs to identify

non-crashing DMEs and determine crash-related bugs by inspecting logs recorded through LOGCAT.

Table 2 presents the results of the bugs found by SceneData. Specifically, for each detected bug in an app, it lists the bug ID, the state (fixed, confirmed, reported) of the submitted issue, the related DMFs, the length of the minimal test needed to reproduce the bug, the type and a brief description of the bug. Overall, SceneData finds 25 previously unknown bugs across 12 apps. Among these found bugs, 21 are non-crashing bugs, and the remaining four are crash bugs. We submitted the found bugs as issues to the developers, including information about the devices app ran on, videos of the bugs, and reproducible steps, to facilitate the developers in quickly identifying and fixing the bugs. So far, 15 bugs have been confirmed, of wich 6 have been fixed. Additional bugs are pending resolution, with none being rejected. Moreover, 21 out of the 25 require a combination of two or more DMFs to manifest the bugs, which demonstrates the effectiveness of SceneData in detecting DMEs. Moreover, we received positive feedback from developers regarding the issues we submitted. For example, *Markor*'s developer mentioned "*It's a pain to keep dialog state across rotations*" and *OmniNote*'s developer commented "*You're right, thanks for bringing that up*".

Bug Types and Assorted Samples SceneData detects various non-crashing DMEs. Based on the bug symptom and consequence, we categorize these 21 non-crashing DMEs into four types which are reported in Table 2. To illustrate these types, we provide specific sample for each type.

(1) Data update delay (1/21 bugs, 5%). This bug type prevents user data from being updated immediately. SceneData found such an issue in *Anymemo*, a spaced repetition flashcard learning software. Figure 8(a) shows a card list that is being studied. When a user updates a card in the card list, for example, updating the card 'heads' to 'head', the card list does not change. However, when the user clicks 'heads' in the card list, the detailed information of the updated card 'head' is displayed (Fig. 8(b)). Only after reloading the card list, these updates would be displayed normally (Fig. 8(c)).

(2) Unexpected wrong behavior (11/21 bugs, 52%). This bug type indicates that the UI page does not display the expected results after performing a specific functionality. Most DMEs causes abnormal behavior of the app. SceneData found such an issue in *ActivityDiary*. Figure 8(e) shows the search page that the user accesses by clicking the search button from the home page. On this page, the user first searches for an activity "Sl" and then clicks the add button to add the activity "Sliding". However, after performing these two operations, the current page does not display the added activity. Unexpectedly, *ActivityDiary* displays the page of the added activity normally after cancelling the search (Fig. 8(f)). Such a DME may lead the user to believe that the activity was not successfully added and prompting them to try adding it again.

(3) User data loss (5/21 bugs, 24%). This type of bugs results in user data loss. Users may perform uncommon actions while using an app, which some apps find challenging to handle. Such DMEs occurred when the user rotated its phone screen while entering text for a search, causing the app to terminate the search. For example, in *Skytube*,[13] rotating the screen can result in the entered text disappearing (Fig. 8(g)). Figure 8(h) shows that the page should remain unchanged after two screen rotations.

(4) Function blocked (4/21 bugs, 19%). This type of bug means that a specific functionality cannot proceed or lost effect. UI widgets are designed to perform unique functions, but developers might not have thoroughly tested these widgets to ensure they works flexibly.

[13] https://github.com/SkyTubeTeam/SkyTube

Table 2 Bug finding results of SceneData

App name	Issue ID	Issue State	Related DMFs	#Steps	Type	Description
Markor	#2287	Fixed	Create,Update,Read	8	Function blocked	Renaming the file name in "Recently viewed documents" is invalid, and it cannot be opened when viewed again
	#2289	Fixed	Create	4	Wrong behavior	Incorrect extension will be included as part of the file name
	#2301	Confirmed	Create,Update	9	Data loss	Insert some symbols separately, file save failed after viewing
	#2311	Confirmed	Create,Search	8	Data loss	Rotating during editing will result in search failure
Aard2	#179	Fixed	Create,Search	6	Crash	The app crashes when filtering
	#180	Confirmed	Read	4	Wrong behavior	The layout is inconsistent using the "Zoom Out" setting
	#181	Reported	Create,Read	6	Wrong behavior	When a word is bookmarked, duplicates appear marked but are not actually bookmarked
SkyTube	#1269	Reported	Create,Search	7	Data loss	Rotating during editing will result in search failure
AnyMemo	#536	Reported	Create,Update, Search,Read	8	Update delay	Card list update delay, error in viewing the card that has not been updated
	#535	Reported	Search	3	Crash	Can't look up card in a dictionary
Amaze	#4179	Confirmed	Create,Search	9	Function blocked	Cannot search in Recent files
	#4180	Confirmed	Create,Search,Update	13	Function blocked	Cannot search or rename files within Documents
	#4182	Confirmed	Create,Search	10	Crash	Sorting by relevance when search fails causes a crash

Table 2 continued

App name	Issue ID	Issue State	Related DMFs	#Steps	Type	Description
	#4185	Confirmed	Create,Search	8	Data loss	Rotating during editing will result in search failure
AnkiDroid	#16463	Confirmed	Create,Search	12	Wrong behavior	Search will fail if the text contains "_"
	#16460	Fixed	Create,Read	9	Wrong behavior	Cards of type Basic (type in the answer) do not display answers
Wikipedia	#T366132	Fixed	Create,Read, Search	12	Wrong behavior	Invalid removal after saving articles in history
Tasks	#2887	Reported	Create,Search	8	Wrong behavior	Search results are not sorted by relevance
RadioDroid	#1216	Reported	Create,Read	6	Crash	Clicking on the radio category information in favorites causes a crash
ActivityDiary	#317	Reported	Search,Create	5	Wrong behavior	Main page failed to add a note during search
	#318	Reported	Search,Delete	4	Wrong behavior	Delete a note during the search process on the main page, the note will not disappear
	#319	Reported	Search,Create	9	Wrong behavior	The activity added from the menu search page is not displayed (two ways)
NotePad	#153	Reported	Create,Update	8	Data loss	Cannot discard changes during note editing process
Omni Notes	#979	Confirmed	Create	8	Wrong behavior	Share the note to this app and it will be added by default
	#980	Fixed	Create,Search	9	Function blocked	Cannot cancel the checklist filtering condition to continue searching

App Name	Issue ID	Issue State	Related DMFs	#Steps	Type	Description
Markor	#2287	Fixed	Create,Update, Read	8	Function blocked	Renaming the file name in "Recently viewed documents" is invalid, and it cannot be opened when viewed again
	#2289	Fixed	Create	4	Wrong behavior	Incorrect extension will be included as part of the file name
	#2301	Confirmed	Create,Update	9	Data loss	Insert some symbols separately, file save failed after viewing
	#2311	Confirmed	Create,Search	8	Data loss	Rotating during editing will result in search failure
Aard2	#179	Fixed	Create,Search	6	Crash	The app crashes when filtering
	#180	Confirmed	Read	4	Wrong behavior	The layout is inconsistent using the "Zoom Out" setting
	#181	Reported	Create,Read	6	Wrong behavior	When a word is bookmarked, duplicates appear marked but are not actually bookmarked
SkyTube	#1269	Reported	Create,Search	7	Data loss	Rotating during editing will result in search failure
AnyMemo	#536	Reported	Create,Update, Search,Read	8	Update delay	Card list update delay, error in viewing the card that has not been updated
	#535	Reported	Search	3	Crash	Can't look up card in a dictionary
Amaze	#4179	Confirmed	Create,Search	9	Function blocked	Cannot search in Recent files
	#4180	Confirmed	Create,Search, Update	13	Function blocked	Cannot search or rename files within Documents
	#4182	Confirmed	Create,Search	10	Crash	Sorting by relevance when search fails causes a crash
	#4185	Confirmed	Create,Search	8	Data loss	Rotating during editing will result in search failure
AnkiDroid	#16463	Confirmed	Create,Search	12	Wrong behavior	Search will fail if the text contains "_"
	#16460	Fixed	Create,Read	9	Wrong behavior	Cards of type Basic (type in the answer) do not display answers
Wikipedia	#T366132	Fixed	Create,Read, Search	12	Wrong behavior	Invalid removal after saving articles in history
Tasks	#2887	Reported	Create,Search	8	Wrong behavior	Search results are not sorted by relevance
RadioDroid	#1216	Reported	Create,Read	6	Crash	Clicking on the radio category information in favorites causes a crash
ActivityDiary	#317	Reported	Search,Create	5	Wrong behavior	Main page failed to add a note during search
	#318	Reported	Search,Delete	4	Wrong behavior	Delete a note during the search process on the main page, the note will not disappear
	#319	Reported	Search,Create	9	Wrong behavior	The activity added from the menu search page is not displayed (two ways)
NotePad	#153	Reported	Create,Update	8	Data loss	Cannot discard changes during note editing process
Omni Notes	#979	Confirmed	Create	8	Wrong behavior	Share the note to this app and it will be added by default
	#980	Fixed	Create,Search	9	Function blocked	Cannot cancel the checklist filtering condition to continue searching

Fig. 8 Examples of different types of non-crashing DMEs. In each group, the page at top shows the erroneous behavior, while the page at bottom shows the correct behavior, and the red boxes indicate the clues of each issue

For example, in *OmniNotes*, the user may search for a created note and click the checklist filter, but not find the desired notes (Fig. 8(i)). Additionally, the user cannot clear the filter to resume searching on the page (Fig. 8(j)).

By examining the types of the previously unknown DMEs detected by SceneData, we find that while most of these DMEs generally do not result in app crashes, they have varying impacts on user experience. These findings substantiate our hypothesis **H1**, which ensures

that SceneData is effective in detecting previously unknown and diverse types of DMEs in real-world android apps.

5.2 RQ2: To What Extent Does SceneData Complement the State-of-the-Art Techniques in Detecting DMEs?

Table 3 provides a summary of the total number of previously unknown bugs (non-crashing DMEs) detected by five state-of-the-art testing tools: Monkey, Genie, Odin, PBFDroid and SceneData. For instance, under the 'SceneData' column, the row labeled 'total' with the entry 25(21) indicates that SceneData has discovered a total of 25 bugs, of which 21 are non-crashing DMEs and the remaining 4 are crash-related bugs.

We find that Monkey, Genie and Odin only covered all crash bugs except for the amaze app, but failed to find the 21 non-crashing DMEs detected by SceneData. We analyzed their results and attributed their missing these non-crashing DMEs to two factors. (1) Lack of a suitable oracle for detecting DMEs. Monkey detects crash bugs by monitoring Android system logs during execution. However, it lacks a mechanism to identify non-crashing DMEs since it does not analyze UI behavior and validate data consistency. For Genie, based on the principle that other views should not be affected by interacting with an independent view, the general oracle is difficult to generalize DME. Odin adds the same events to similar states to expand the test input, and clusters the behaviors shown through abstract rules to find anomalies. However, this rule does not consider text content changes, which are critical for determining whether data manipulations are executed normally. For example, it fails to capture file name modifications, such as renaming a file from "To-do.md" to "Done.md," which is essential for validating the correctness of update operations. (2) Inefficient search strategy generates low-quality test. These tools employ random strategy in test generation, which often result in incomplete execution of the entire data manipulation workflow, failing to cover the semantics of data manipulations. The triggering of DME typically requires executing multiple interdependent

Table 3 Previously unknown bugs found by five tools. Numbers indicate the total bugs, with non-crashing DMEs in parentheses if applicable

App	Monkey	Genie	Odin	PBFDroid	SceneData
Markor	0	0	0	0	4(4)
Aard2	1(0)	1(0)	1(0)	2(1)	3(2)
SkyTube	0	0	0	0	1(1)
AnyMemo	1(0)	1(0)	1(0)	1(0)	2(1)
Amaze	0	0	0	1(1)	4(3)
AnkiDroid	0	0	0	0	2(2)
Wikipedia	0	0	0	0	1(1)
Tasks	0	0	0	0	1(1)
RadioDroid	1(0)	1(0)	1(0)	1(0)	1(0)
ActivityDiary	0	0	0	1(1)	3(3)
Notepad	0	0	0	0	1(1)
OmniNotes	0	0	0	0	2(2)
Total	3(0)	3(0)	3(0)	6(3)	25(21)

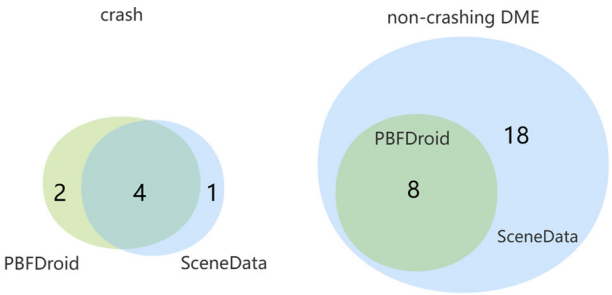

Fig. 9 Comparison of PBFDroid and SceneData in detecting all DMEs. The left (resp. right) diagram represents crash (resp. non-crashing) DMEs; the green (resp. blue) area represents DMEs detected by PBFDroid (resp. SceneData). (The overlapping areas indicate DMEs detected by both)

DMFs in a specific order. As shown in Table 2, each reported DME involves a sequence of steps, highlighting the necessity of exploration strategy to ensure a comprehensive coverage.

Our findings confirm that while a suitable oracle is essential for detecting DMEs, it alone does not guarantee effective detection. While PBFDroid and SceneData both utilize specialized oracles, PBFDroid is only able to find three of the non-crashing DMEs found by the SceneData. This indicates that search strategy is also crucial in ensuring the discovery of non-crashing DMEs. SceneData improves the exploration through a scene-guided strategy, which targets scenes and enhances app property validation. It execute all candidate DMFs in the scene only once. When encountering the scene again, the state of the app is explored by triggering other widgets, which helps find DMEs in the deep state of the app. Taking the scene corresponding to the state in Fig. 1(e) as an example, SceneData may first execute the DMF “Rename File" and, upon revisiting the same scene, select another DMF, such as “View File", until all executable DMFs have been covered. Scenes can help identify changes in the state of the app. SceneData can determine non-crashing exceptions of data operations based on scene inconsistencies, while PBFDroid can only detect related crashes. For example, the state of the app should not change when the screen is rotated and then restored. In addition, the DMF-directed DFS exploration strategy requires the SceneData to prioritize the validation of candidate DMFs. When new scenes appear in the app, the target is shifted to the new scene, which explicitly considers the combination of different DMFs. Dynamic exploration is driven by scenes and can record the DMF history executed in each scene, which facilitates thorough validation of app properties. For example, while executing DMFs in the scene shown in Fig. 1(c), if the exploration transitions to a new scene shown in Fig. 1(e), SceneData prioritizes executing all DMFs in new scene before returning to the previous one. This strategy avoids unnecessary repeated testing in PBFDroid, which ensures SceneData detects DMES more effectively.

Since some of the discovered bugs have been reported before, we further examine how well PBFDroid and SceneData detect both newly discovered and previously identified DMEs. As shown in Fig. 9, PBFDroid and SceneData exhibit complementary capabilities in detecting both crash and non-crashing DMEs. Among all detected DMEs, PBFDroid uniquely identified 2, whereas SceneData uniquely detected 19, and both methods identified 12 DMEs. While PBFDroid detected some unique DMEs, all of them are crash-related errors, whereas SceneData successfully detected a larger number of non-crashing DMEs. These results suggest that PBFDroid’s random exploration strategy is particularly effective at discovering

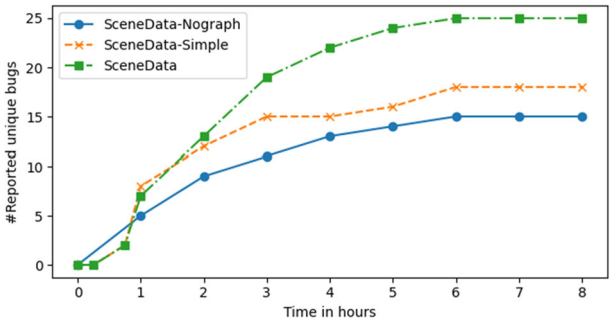

Fig. 10 The number of revealed bugs over execution time

system-level crashes. It identifies the crash in the app *Transistor*[14] caused by pressing the Next key on the keyboard while editing, which is challenging for SceneData to trigger. In addition, SceneData's scene-guided search strategy significantly enhances the detection of non-crashing DMEs by systematically validating app states and transitions.

To evaluate the statistical significance of the performance differences in DME detection between SceneData and the state-of-the-art baseline PBFDroid, we conduct a Wilcoxon signed-rank test (Wilcoxon 1992). In our study, the null hypothesis is set as H_0: There is no significant difference between SceneData and PBFDroid in detecting previously unknown DMEs. The significance level for this test is set at 0.05. The input for this test consists of paired data points representing the number of DMEs detected by PBFDroid and SceneData for each app reported in Table 3. For the Wilcoxon signed-rank test, the p-value is 5.0*e*-4, which is lower than 0.05. This shows that we can reject the null hypothesis H_0, confirming that the SceneData significantly outperforms PBFDroid in detecting previously unknown DMEs. Futhermore, the results presented in Fig. 9 demonstrate that SceneData complements PBFDroid in detecting all DMEs. In addition, Table 3 highlights that PBFDroid can complement other baselines in detecting reported unique DMEs. These results lead us to accept hypothesis **H2**. Therefore, we conclude that SceneData demonstrates superior performance by detecting unique DMEs that are missed by state-of-the-art techniques.

5.3 RQ3: How Effective are ScenceTG and Scene-Driven Exploration in Improving the Testing Capability of Finding DMEs? How does test sequence length impact the DME detection capability of SceneData?

ScenceTG As shown in Fig. 10, the lack of guidance in the SceneTG (SceneData-Nograph) resulted in missing 10 bugs. Overall, the number of unique bugs discovered by SceneData-Nograph was consistently lower than that of SceneData at each time interval (1h) until no new bugs were found.

In particular, during the first hour, SceneData-Nograph and SceneData exhibit significantly different trends in bug detection. Initially, SceneData does not find any bugs due to the analysis of the app used to collect data for constructing the SceneTG. After 15 minutes, when SceneData entered the dynamic exploration phase, its effectiveness becomes more apparent. Until 30 minutes later, SceneData successfully identifies two crash bugs by monitoring the anomalies during app runtime. We observed that the remaining two crash bugs

[14] https://github.com/y20k/transistor/

involved app data manipulation, which were missed during the sceneTG construction phase because no relevant app inputs were generated. Taking the *radiodroid* app as an example, the crash is triggered only when clicking on the radio category after a radio has been saved. Due to the varying order in which dynamic exploration triggers widgets, the state of saving the radio may not be explored.

Furthermore, the dynamic exploration phase captures the transition relation between app scenes. The number of explored activities, transition pairs, and scenes recorded by the constructed SceneTG are depicted in Fig. 11. On average, across the 17 collected apps, SceneData explores 9 activities, extracts 51 transition pairs, and identifies 27 scenes. We observe that in exploration activities with a mean difference of 1, the graph generated by SceneData has 14 more transition pairs than the graph generated by SceneDroid (Zhang et al. 2023), a state-of-the-art GUI modeling tool, and discovers 9 new scenes. For example, SceneDroid visites page (d) of Fig. 7 in the order $(a) \rightarrow (b) \rightarrow (c) \rightarrow (d)$, requiring multiple restarts and navigation back to page (d). In contrast, SceneData leverages saved widgets in page (a) to directly navigate to pages (c) and (d) via the navigation bar. This shows that our SceneData is able to build a more complete SceneTG in a shorter time. This phenomenon is beneficial for thoroughly validating app properties in each scene.

By inspecting the experimental results, we found that all missed errors were related to non-crashing DMEs. The non-crashing DMEs missed by SceneData-Nograph can be attributed to two main factors. First, the current scene is not included in the preconditions defined by the DMF, making it impossible to validate certain properties of the app in the current state. Second, the inability to identify potentially executable DMFs may lead to a reduction in relevant data, limiting the number of DMF combinations that can be validated. For example, in the app *Anymemo*, executing DMF "Search Card" can only be initiated from the "MORE" widget on the home page. This restriction prevents from performing the search operation directly from the card list detail page, thereby failing to fully test what normal behaviors are affected by the delay in updating the card.

Scene-Driven Exploration As shown in Fig. 10, for any explored scene, if we adopt the strategy of only randomly executing a DMF or interleaving possible event (SceneData-Simple) to test the behavior of the app, we observe that it eventually uncover 18 bugs. This phenomenon indicates that our DMF-directed DFS exploration strategy is effective in discovering possible bugs.

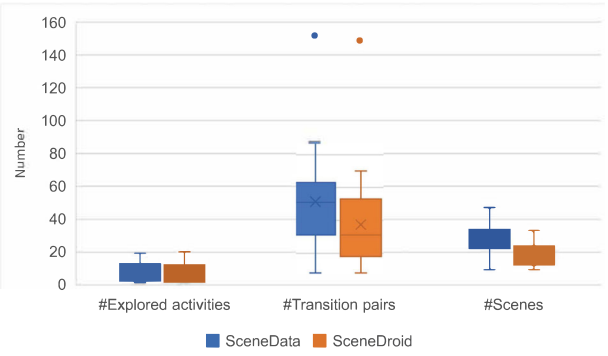

Fig. 11 Comparison of #Explored activities, #Transition pairs, and #Scenes

Specifically, there are two main reasons to explain why SceneData-Simple missed 7 out of 25 (28%) bugs. First, the randomness in interacting with the app introduces considerable uncertainty. This means a particular DMF within a scene may be executed repeatedly, or it might never be triggered at all. For example, irregularly executing the DMFs of the scene in Fig. 1(e) reduces the probability of executing DMF “Rename File". Second, some bugs are located in obscure areas of the app. Random exploration might coincidentally overlook these corners, making it even more challenging to conduct DMF interactions in these less accessible regions. To detect the DME in Fig. 1, the recently viewed documents must first be accessed, which are nested under a widget in the top navigation bar, making navigation challenging.

Test Sequence Length As shown in Fig. 12, setting the maximum event length $threshold_1$ to 100 gives the lowest number of detected DMEs, as the test with only 100 operations cannot cover a sufficient number of DMF interactions. When the test sequence length is set to 200, 300 and 400, no significant difference is observed for SceneData, but a longer event length allows bugs to be detected more quickly. With prolonged execution, SceneData eventually behaves like a random exploration tool, as it attempts to match the most closely explored scene for DMFs execution.

Our results demonstrate that the utilization of ScenceTG and Scene-Driven Exploration has a significant positive impact on the detection of DMEs. In addition, a longer test sequence accelerates the discovery of DMEs but does not significantly increase the total number of DMEs, and a shorter test sequence may reduce the effectiveness of DME detection. Therefore, we accept hypothesis **H3**, confirming that the integration of ScenceTG and Scene-Driven Exploration significantly enhances the effectiveness of DME detection, as well as different test sequence lengths affect the effectiveness and efficiency of DME detection.

6 Discussion

Finding Non-Crashing DMEs Detecting non-crashing bugs in apps, especially those related to data manipulation, present significant challenges. Many of the bugs discovered in this study originate from well-maintained apps, including those extensively tested. For instance,

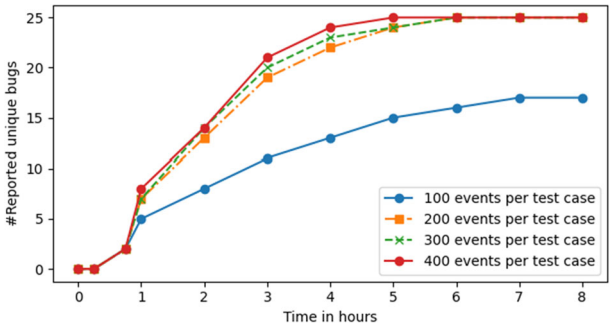

Fig. 12 Number of found unique bugs under different the maximum event length $threshold_1$

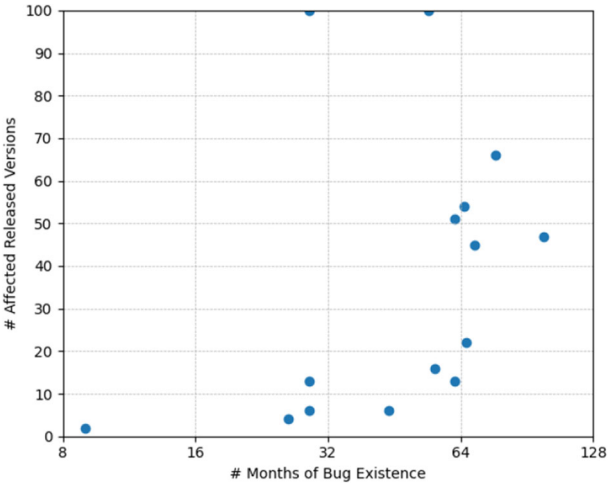

Fig. 13 Statistics of 21 non-crashing DMEs: (1) #months they have resided (x-axis, in logarithmic scale); (2) #affected releases (y-axis), before they were uncovered

AnkiDroid[15] includes more than 200 well-written UI and unit test cases, each equipped with assertions to verify correctness. Despite the rigorous testing, SceneData discovered 21 non-crashing DMEs, demonstrating its ability to help developers effectively explore various states of their app.

Figure 13 illustrates the characteristics of the number of versions affected and the number of months on Google Play before these 21 non-crashing DMEs were uncovered, with each point representing a non-crashing DME (some points overlap). We observe that most of these DMEs have existed over two years, and 16 (76.2%) non-crashing DMEs have affected more than 10 versions. For example, three data loss issues triggered by screen rotation have been present since the initial release of the app. SceneData detects this type of DMEs by identifying UI pages as scenes based on layout information, monitoring changes in UI pages during testing. These results reveal that many of the detected bugs have remained latent for extended periods and were missed by traditional testing methods, providing significant assistance to developers in maintaining the apps.

Why Does SceneData Work? We conduct a comprehensive analysis of SceneData's inner mechanisms to elucidate the underlying key factors in detecting DMEs. First, SceneData enhances DME detection by leveraging prior knowledge, including GUI scenes and scene relationships. RQ3 demonstrates that the absence of model guidance significantly impairs the performance of DME detection. Constructing a model based on scene granularity enables accurate capture of DMF operations. Additionally, the dynamic exploration strategy, which flexibly handles page transition logic, allows for the rapid construction of a more comprehensive app model. Second, testing at the scene granularity reduces redundant validation and focuses on validating DMF operations across different scenes. By introducing scene-guided exploration, SceneData records the history of DMF execution, allowing developers to effectively filter and validate DMF across different scenes without unnecessary repeti-

[15] https://github.com/ankidroid/Anki-Android/

tion. Furthermore, randomly triggering events that promote scene transitions in the explored scenes facilitates exploration of app diverse states.

Manual Effort Required for SceneData DME detection relies on predefined DMF specifications, which require users to provide necessary information, as described in Section 3.1. We evaluated the manual effort involved in defining these specifications. Figure 14 presents the time taken by three participants to define a single DMF for each of the 16 subject apps. The horizontal axis displays the apps, and the vertical axis represents the time (in minutes). The results indicate that the time cost for defining a single DMF ranges from 0.98 to 5.05 minutes, with an average of 2.99 minutes. For all DMFs of a single app, the time ranges from 6.8 to 23.42 minutes, with an average of 14.56 minutes. The time (over 20 minutes) is longer for AnyMemo and AnkiDroid due to their larger number of DMFs. Since these results are obtained using PBFDroid?s DMF instantiator, they are approximately the same as those of PBFDroid.

Potential Applications of SceneData In addition to detecting DMEs, SceneData can also assist developers with other aspects of testing. First, SceneData constructs a SceneTG by exhaustively exploring the interactive widgets in the activities launched by ICC messages, which can be used for regression testing. SceneTG records scene transitions within the app and the widgets that trigger these transitions, providing the corresponding real UI page for each identified scene. Developers can employ the SceneTGs from different app versions to target the functionalities of the modified parts. In the second phase, scene-driven exploration uses property-based testing to validate the behavior of app. Given accurate DMF properties, SceneData can automatically find DMEs without any false positives. Property-based testing, due to its broad applicability, is not limited to testing DMEs and can be extended to identify other types of errors, making it suitable for a wider range of software testing scenarios. An instantiated DMF specifies the preconditions and postconditions required for executing a functionality. These necessary conditions facilitate the validation of whether the app correctly achieves its functionality. While SceneData effectively detects DMEs, it remains unclear whether existing baselines (e.g., Monkey, Genie and Odin) encounter DMEs but fail to report them due to the absence of a suitable oracle. Future work could investigate this

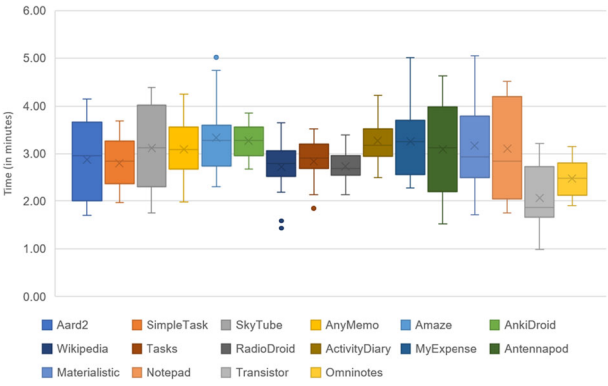

Fig. 14 Time spent by participants in our study on defining one single DMF

by integrating SceneData?s oracle into these tools to enable a fairer comparison of search strategies. Understanding wherther their limitation lies in search strategy or detection capability would provide insights for enhancing the practical applicability of DME detection in real-world scenarios.

Threats to Validity A major threat to external validity is the representativeness of the app subjects. To mitigate this threat, we included multiple open-source apps from various categories, as detailed in Table 1. These apps are selected based on their high visibility and substantial user bases on both GitHub and Google Play. This selection ensures that our findings are generalizable and reflective of real-world apps. Another external threat relates to the selection of DMF types for validating app properties. To mitigate this, we focus on common operation types including create, read, update, delete, and search, which are fundamental for data manipulation in apps and essential for completing various tasks. By concentrating on these common operations, we ensure that that the DMFs effectively cover the core functionality of each app.

The first internal threat is the potential errors in the implementation of our approach. To mitigate this threat, we carefully check the code and utilize mature libraries such as uiautomator2. The second internal threat is the implementation of the baseline approaches. To mitigate this threat, we follow the previous work (Wang et al. 2020) to set the event interval of Monkey. For other baselines, we use the default parameter settings of the baseline approaches to conduct experiments. The third internal threat concerns human factors in the DMF instantiation process, which may introduce inaccuracies or biases. To address this, we take several measures to ensure the fairness of the experiment. First, we follow the DMFs defined in PBFdroid (Sun et al. 2023a) to instantiate the DMFs. In case where the widgets of the app changed in the new version, participants meticulously validate the DMF accordingly. Second, to minimize human errors, we provid detailed tutorials during the DMF instantiation process to measure the time cost. Third, each DMF is independently evaluated by three different participants to reduce possible biases. Additionally, the correctness of DMF execution is evaluated by recording screenshots and events at each step. Furthermore, for the detected bugs in the apps and their root causes, all reports are independently reviewed and cross-checked by the three authors of this paper to ensure correctness.

The first construct validity pertains to the appropriateness of our evaluation metrics. To mitigate this, we analyze the number and categories of detected bugs, along with their root causes. This evaluation helps demonstrate the effectiveness of the generated test cases and ensure they reflect potential issues within the apps. The second construct validity focuses on the functional correctness of the test cases generated by SceneData. The consistency between the data model and UI layout is used to validate whether the DMF achieves its functionality correctly, thus illustrating the feasibility of functional validation through test cases. Based on this, we manually inspected the screenshots captured during the testing process and the operations performed in each event of the test case to further evaluate the effectiveness of SceneData.

7 Related Work

Detecting Crash Bugs in Android Apps Various automated GUI testing approaches have been introduced to enhance the reliability and quality of Android apps. These approaches employ different strategies such as random (Monkey 2024; Machiry et al. 2013; Kong et al.

2018), model-based (Su et al. 2017; Wang et al. 2020; Gu et al. 2019), search-based (Mao et al. 2016; Dong et al. 2020; Mahmood et al. 2014; Amalfitano et al. 2015), and integrated reinforcement learning (Koroglu and Sen 2019; Pan et al. 2020; Romdhana et al. 2022) to verify app behavior. However, those approaches rely on app runtime exceptions as implicit oracles, which makes it challenging to detect non-crashing bugs.

Detecting Non-Crashing Bugs in Android Apps Several tools have been proposed to detect non-crashing bugs without requiring predefined oracles. These existing works are based on the idea of metamorphic testing, by designing metamorphic relations applied to the app execution results and checking whether the relations are violated to detect non-crashing bugs. However, most of these tools are designed for specific types of bugs, such as data loss (Guo et al. 2022; Riganelli et al. 2020) and system settings (Sun et al. 2021, 2023b), which not consider explicit DMF operations during testing. While Genie (Su et al. 2021) targets generic bugs, its metamorphic relations are limited by the independence between event fragments, making it less effective in detecting DMEs. On the other hand, Odin (Wang et al. 2022) detects non-crashing bugs by identifying abnormal behaviors that deviate from normal app behavior. However, it clusters exhibited behaviors using abstract rules without considering text features, which results in bugs arising from data inconsistencies escaping detection.

Most existing works enhance the ability to validate app behaviors through manually written oracles. Test migration techniques (Liu et al. 2022, 2024; Lin et al. 2019; Behrang and Orso 2019; Talebipour et al. 2021) manually construct tests for one app to generate tests for another app with similar functionalities. These approaches are designed purely for single functionalities of the app and do not consider the interaction between different functionalities. The no one-to-one event matching during test migration motivates the design of SceneData. Unlike traditional approaches, SceneData does not limit the implementation of a functionality to the initial state in the specified path. Instead, it recognizes the GUI scene and explores the state transition details to identify whether a DMF can be executed in the scene. Although the aforementioned works are effective in detecting non-crashing functional bugs, they are not capable of detecting DMEs.

Finding Data Manipulation-Related Bugs Our work focuses on testing data manipulation-related behavior, specifically targeting CRUD related bugs in the software. Existing studies (Rigger and Su 2020a, b have detected CRUD errors in database management systems, but these approaches are not specifically tailored for Android apps. PBGT Costa et al. 2014) utilizes the concept of User Interface Test Patterns (UITPs) to test common recurrent behaviors in apps through their possible different implementations. However, it only models the Find operation as a UITP to check correctness, without considering other related data manipulations. Augusto (Mariani et al. 2018) leverages semantic knowledge related to CRUD operation functionalities to automatically generate effective test cases with functional oracles. It systematically covers critical scenarios and detects failures by encoding expected functionality into models customized for the app under test. However, it primarily generates system test cases for individual CRUD operation independently, without explicitly modeling their interdependencies. CRUD operations often interact in complex ways, where an operation (such as update or delete) may depend on prior create operations. Ignoring these dependencies may lead to inconsistencies or failures that single-operation testing might not reveal.

Complementarily, PBFDroid (Sun et al. 2023a) validates app properties through the consistency of the data recorded in the data model and the data visualized in UI, enabling

validation of the correctness of data manipulations. It randomly interleaves different DMFs and possibly other events to generate various app states. However, its random exploration may struggle to systematically cover all possible interactions between DMFs, potentially overlooking deep DMEs. Building on the defined DMF specifications, SceneData enhances this approach with an improved exploration strategy to further validate app behavior. Different from PBFDroid, SceneData designs a DMF-directed DFS strategy to traverse the executable DMFs for each new scene. This strategy explores multiple combinations of DMFs, effectively and thoroughly validating the states of the app.

8 Conclusion and Future Work

Developers must test whether the app behaves normally under various DMF combinations, which is often time-consuming and cumbersome. In this paper, we have introduced SceneData, an automated approach that adopts scene-guided exploration to detect DMEs in Android apps. SceneData launches app activities through ICC messages and identifies scenes within these activities. It constructs a SceneTG by exhaustively exploring the operational widgets in each scene, capturing the details of app's state transitions. Guided by the SceneTG, SceneData performs DMF-directed DFS and intersects with other possible events to intentionally explore the interactions between DMFs. SceneData employs model-based properties of the DMFs as oracles to validate app properties. SceneData identified 21 non-crashing DMEs in 12 real-world Android apps. We reported these bugs to the app developers, and So far, 15 of the submitted issues have been confirmed, with 6 of them already fixed. Comparative experiments demonstrate that the state-of-the-art testing techniques struggle to find the non-crashing DMEs that SceneData successfully discovers.

In the future, we plan to extend the oracle of our current approach to existing tools such as Monkey and Genie, and conduct a comprehensive empirical study to determine whether their inability to detect non-crashing DMEs arises from search inefficiencies or the absence of a suitable oracle.

Author Contributions **Shuqi Liu**: Data curation, Software, Writing−original draft; **Yu Zhou**: Conceptualization, Methodology, Supervision, Writing−review & editing; **Wenhua Yang:** Conceptualization, Methodology; **Taolue Chen:** Conceptualization, Methodology, Writing−review & editing; **Harald Gall:** Conceptualization, Methodology.

Funding This work was partially supported by the National Natural Science Foundation of China (NSFC, No. 62372232), the Fundamental Research Funds for the Central Universities (No. NG2023005), and the Collaborative Innovation Center of Novel Software Technology and Industrialization. T. Chen is partially supported by overseas grants from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2023A04).

Data Availability The dataset used in this paper is available at https://github.com/liushuqi-2022/DME-detection. To foster the replication of our study, we will publish the implementation of our SceneData at https://github.com/liushuqi-2022/Scene-Data.

Declarations

Ethical Approval The datasets used in this study are publicly available from previous studies, and the sources have been indicated in the manuscript. All data were used in accordance with the terms and conditions specified by the provider.

Informed Consent All participants in this study provided informed consent before participating. In addition, all co-authors were informed about the study.

Conflicts of Interest The authors declare that they have no conflict of interest.

Clinical Trial Number not applicable.

References

Amalfitano D, Amatucci N, Fasolino AR, Tramontana P (2015) Agrippin: a novel search based testing technique for android applications. In: Proceedings of the 3rd international workshop on software development lifecycle for mobile, pp 5–12

Azim T, Neamtiu I (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, pp 641–660

Behrang F, Orso A (2019) Test migration between mobile apps with similar functionality. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 54–65

Chen S, Fan L, Chen C, Su T, Li W, Liu Y, Xu L (2019) Storydroid: automated generation of storyboard for android apps. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 596–607

Claessen K, Hughes J (2000) Quickcheck: a lightweight tool for random testing of haskell programs. SIGPLAN Not 35(9):268–279. https://doi.org/10.1145/357766.351266

Costa P, Paiva AC, Nabuco M (2014) Pattern based gui testing for mobile applications. In: 2014 9th International conference on the quality of information and communications technology, IEEE, pp 66–74

Dong Z, Böhme M, Cojocaru L, Roychoudhury A (2020) Time-travel testing of android apps. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 481–492

Fdroid (2024) F-droid. https://f-droid.org/

Google (2024a) Android widget listview. https://developer.android.com/android/widget/ListView/

Google (2024b) Google play. https://play.google.com/store/apps/

Gu T, Sun C, Ma X, Cao C, Xu C, Yao Y, Zhang Q, Lu J, Su Z (2019) Practical gui testing of android applications via model abstraction and refinement. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 269–280

Guo W, Dong Z, Shen L, Tian W, Su T, Peng X (2022) Detecting and fixing data loss issues in android apps. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, pp 605–616

Jabbarvand R, Lin JW, Malek S (2019) Search-based energy testing of android. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 1119–1130

Kong P, Li L, Gao J, Liu K, Bissyandé TF, Klein J (2018) Automated testing of android apps: a systematic literature review. IEEE Trans Reliab 68(1):45–66

Koroglu Y, Sen A (2019) Reinforcement learning-driven test generation for android gui applications using formal specifications. arXiv:1911.05403

Lin JW, Jabbarvand R, Malek S (2019) Test transfer across mobile apps through semantic mapping. In: 2019 34th IEEE/ACM International conference on Automated Software Engineering (ASE), IEEE, pp 42–53

Lin JW, Salehnamadi N, Malek S (2023) Route: roads not taken in ui testing. ACM Trans Softw Eng Method 32(3):1–25

Liu S, Zhou Y, Han T, Chen T (2022) Test reuse based on adaptive semantic matching across android mobile applications. 2022 IEEE 22nd International Conference on Software Quality. Reliability and Security (QRS), IEEE, pp 703–709

Liu S, Zhou Y, Ji L, Han T, Chen T (2024) Enhancing test reuse with gui events deduplication and adaptive semantic matching. Sci Comput Program 232:103052

Machiry A, Tahiliani R, Naik M (2013) Dynodroid: An input generation system for android apps. In: Proceedings of the 2013 9th Joint meeting on foundations of software engineering, pp 224–234

Mahmood R, Mirzaei N, Malek S (2014) Evodroid: segmented evolutionary testing of android apps. In: Proceedings of the 22nd ACM SIGSOFT International symposium on foundations of software engineering, pp 599–609

Mao K, Harman M, Jia Y (2016) Sapienz: multi-objective automated testing for android applications. In: Proceedings of the 25th international symposium on software testing and analysis, pp 94–105

Mariani L, Pezzè M, Zuddas D (2018) Augusto: exploiting popular functionalities for the generation of semantic gui tests with oracles. In: Proceedings of the 40th international conference on software engineering, pp 280–290

Monkey (2024) Monkey. http://developer.android.com/tools/help/monkey.html/

Pan M, Huang A, Wang G, Zhang T, Li X (2020) Reinforcement learning based curiosity-driven testing of android applications. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 153–164

Riganelli O, Mottadelli SP, Rota C, Micucci D, Mariani L (2020) Data loss detector: automatically revealing data loss bugs in android apps. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, pp 141–152

Rigger M, Su Z (2020) Finding bugs in database systems via query partitioning. Proc ACM Program Languages 4(OOPSLA):1–30

Rigger M, Su Z (2020b) Testing database engines via pivoted query synthesis. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp 667–682

Rivest R (1992) The md5 message-digest algorithm. Tech. rep

Romdhana A, Merlo A, Ceccato M, Tonella P (2022) Deep reinforcement learning for black-box testing of android apps. ACM Transactions on Software Engineering and Methodology (TOSEM) 31(4):1–29

Su T, Meng G, Chen Y, Wu K, Yang W, Yao Y, Pu G, Liu Y, Su Z (2017) Guided, stochastic model-based gui testing of android apps. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 245–256

Su T, Yan Y, Wang J, Sun J, Xiong Y, Pu G, Wang K, Su Z (2021) Fully automated functional fuzzing of android apps for detecting non-crashing logic bugs. Proc ACM Program Languages 5(OOPSLA):1–31

Sun J, Su T, Li J, Dong Z, Pu G, Xie T, Su Z (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, pp 204–215

Sun J, Su T, Jiang J, Wang J, Pu G, Su Z (2023a) Property-based fuzzing for finding data manipulation errors in android apps. In: Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering, pp 1088–1100

Sun J, Su T, Liu K, Peng C, Zhang Z, Pu G, Xie T, Su Z (2023b) Characterizing and finding system setting-related defects in android apps. IEEE Trans Softw Eng

Talebipour S, Zhao Y, Dojcilović L, Li C, Medvidović N (2021) Ui test migration across mobile platforms. In: 2021 36th IEEE/ACM International conference on Automated Software Engineering (ASE), IEEE, pp 756–767

Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (2010) Soot: a java bytecode optimization framework. In: CASCON first decade high impact papers, pp 214–224

Wang J, Jiang Y, Xu C, Cao C, Ma X, Lu J (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, pp 469–480

Wang J, Jiang Y, Su T, Li S, Xu C, Lu J, Su Z (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, pp 434–446

Wilcoxon F (1992) Individual comparisons by ranking methods. In: Breakthroughs in statistics: methodology and distribution, Springer, pp 196–202

Wohlin C, Runeson P, Höst M, Ohlsson MC, Regnell B, Wesslén A et al (2012) Experimentation in software engineering, vol 236. Springer

Xiong Y, Xu M, Su T, Sun J, Wang J, Wen H, Pu G, He J, Su Z (2023) An empirical study of functional bugs in android apps. In: Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis, pp 1319–1331

Yan J, Liu H, Pan L, Yan J, Zhang J, Liang B (2020) Multiple-entry testing of android applications by constructing activity launching contexts. In: Proceedings of the ACM/IEEE 42nd international conference on software engineering, pp 457–468

Yan J, Zhang S, Liu Y, Yan J, Zhang J (2022) Iccbot: fragment-aware and context-sensitive icc resolution for android applications. In: Proceedings of the ACM/IEEE 44th international conference on software engineering: companion proceedings, pp 105–109

Yang S, Zeng Z, Song W (2022) Permdroid: automatically testing permission-related behaviour of android applications. In: Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis, pp 593–604

Zhang X, Fan L, Chen S, Su Y, Li B (2023) Scene-driven exploration and gui modeling for android apps. In: 2023 38th IEEE/ACM International conference on Automated Software Engineering (ASE), IEEE, pp 1251–1262