Property-based Fuzzing for Finding Data Manipulation Errors in Android Apps

Jingling Sun
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
jingling.sun910@gmail.com

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

Jiayi Jiang
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
jyjiangsunny@gmail.com

Jue Wang
State Key Lab for Novel Software Tech. and Dept. of Computer Sci. and Tech., Nanjing University
China
juewang591@gmail.com

Geguang Pu*
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
ggpu@sei.ecnu.edu.cn

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

ABSTRACT

Like many software applications, *data manipulation functionalities* (DMFs) are prevalent in Android apps, which perform the common CRUD operations (*create*, *read*, *update*, *delete*) to handle app-specific data. Thus, ensuring the correctness of these DMFs is fundamentally important for many core app functionalities. However, the bugs related to DMFs (named as *data manipulation errors*, DMEs), especially those *non-crashing logic ones*, are prevalent but difficult to find. To this end, inspired by property-based testing, we introduce a *property-based fuzzing* approach to effectively finding DMEs in Android apps. Our *key* idea is that, given some type of app data of interest, we randomly interleave its relevant DMFs and other possible events to explore *diverse* app states for thorough validation. Specifically, our approach characterizes DMFs in (data) model-based properties and leverage the consistency between the data model and the UI layouts as the handler to do property checking. The properties of DMFs are specified by human according to specific app features. To support the application of our approach, we implemented an automated GUI testing tool, PBFDroid. We evaluated PBFDroid on 20 real-world Android apps, and successfully found 30 unique and previously unknown bugs in 18 apps. Out of the 30 bugs, 29 of which are DMEs (22 are non-crashing logic bugs, and 7 are crash ones). To date, 19 have been confirmed and 9 have already been fixed. Many of these bugs are non-trivial and lead to different types of app failures. Our further evaluation confirms that *none* of the 22 non-crashing DMEs can be found by the state-of-the-art techniques. In addition, a user study shows that the manual cost of specifying the DMF properties with the assistance of our tool is acceptable. Overall, given accurate DMF properties, our approach can automatically find DMEs without any false positives. We have made all the artifacts publicly available at: *https://github.com/property-based-fuzzing/home*.

CCS CONCEPTS

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

KEYWORDS

Property-based testing, Model-based testing, Android app testing, Non-crashing functional bugs

ACM Reference Format:
Jingling Sun, Ting Su, Jiayi Jiang, Jue Wang, Geguang Pu, and Zhendong Su. 2023. Property-based Fuzzing for Finding Data Manipulation Errors in Android Apps. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616286

1 INTRODUCTION

Android apps are ubiquitous and serve many different aspects of our daily life [10]. Specifically, like many software applications, *data manipulation functionalities* (DMFs for short) are prevalent in Android apps [17, 43]. These DMFs perform the common data manipulation operations (CRUD) [74] (*i.e.*, *create*, *read*, *update*, *delete*) to handle app-specific data (*e.g.*, creating *files*, reading *emails*, deleting *posts*). Thus, ensuring the correctness of these DMFs is important because they serve as the fundamental of many core app functionalities. However, unlike the crash bugs targeted by many existing automated GUI testing techniques [14, 61, 72], the non-crashing logic bugs related to these DMFs are seldom tackled and may lead to frustrating consequences in real-life [8, 29, 57].

A real example. In this paper, we name the bugs which fail the normal operations of DMFs (*e.g.*, cannot create a file) as *data manipulation errors* (DMEs for short). Figure 1 shows such a DME.

*Ting Su and Geguang Pu are the corresponding authors.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616286

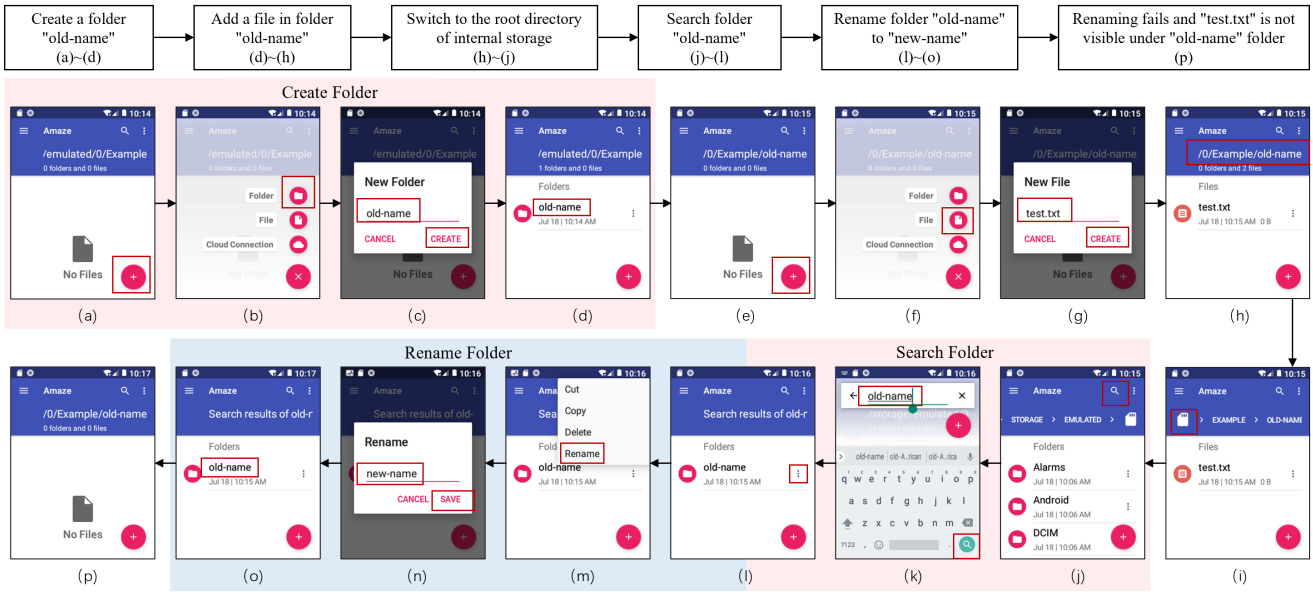

Figure 1: A DME in Amaze (v3.7). The top graph gives an overview of this bug. The small red box on each page denotes a UI event.

Amaze [65], a file management app, allows users to manipulate files or folders on Android devices. However, its DMF "Rename Folder" has a serious bug (shown in Figure 1). Specifically, a user can create a new folder named "old-name" under the directory "`/storage/emulated/0/Example/`" (see Figure 1(a)~(d)), add a file "test.txt" in this folder "old-name" (see Figure 1(e)~(h)), switch to the up-level directory "`/storage/emulated/0/`" (see Figure 1(h)~(j)), and search the created folder "old-name" under "`/storage/emulated/0/`" (see Figure 1(j)~(l)). As expected, the folder "old-name" can be successfully found (see Figure 1(l)). However, in this case, if the user renames the folder "old-name" to "new-name" (see Figure 1(l)~(o)), the app fails as the folder name is not correctly updated to "new-name" (see Figure 1(o)). Even worse, if the user opens the folder "old-name", the original file "test.txt" in this folder cannot be found (see Figure 1(p)). In this example, the failure of renaming the folder "old-name" is a DME, which involves three DMFs "Create folder", "Search folder", and "Rename folder". These DMFs manipulate the app data *folder*. Note that the three DMFs work fine *independently* and the steps ("Create folder", switch to the root directory, "Search folder" and "Rename folder") are the *necessary* operations to manifest the DME.

Prevalence of DMEs. To investigate the prevalence of non-crashing DMEs (the main focus of this paper), we studied the 4 popular, well-maintained open-source Android apps with different categories on GitHub, *i.e.*, Amaze [65] (a file manager, 4K stars), AnkiDroid [3] (a card learning tool, 5.4K stars), K9Mail [32] (an email client, 7.1K stars) and WordPress [76] (a personal blogger, 2.7K stars), and their 245 non-crashing bugs which were reported from August 2018 to July 2021 (spanning three years). By examining the bug reports and reproducing the bugs, we find that a large portion of the non-crashing bugs (29%≈71/245) are DMEs. It indicates that DMEs are indeed prevalent and effective bug finding techniques are in need.

Limitations of existing testing techniques. Modern automated GUI testing techniques have been widely used to help find functional bugs in Android apps [35, 67]. However, existing techniques have two major limitations in finding DMEs. First, the majority of existing testing tools [19, 26, 38, 41, 42, 44, 49, 60, 71] are limited to crash bugs due to the lack of strong test oracles [6]. Thus, they are difficult to find those DMEs which lead to non-crashing failures [77] (like the bug in Figure 1). Second, although some testing tools [62, 70] can find non-crashing bugs by generating automated oracles based on heuristics [20] or metamorphic relations [13]. The oracles are too *generic* to find DMEs which are usually app-specific.

Our approach and its novelty. To this end, inspired by the classic idea of *property-based testing* (PBT) [15], we aim to introduce a novel property-based fuzzing approach to effectively finding DMEs in Android apps. Our insight is that, given a generic app functionality, we can specify its property as ϕ=$\langle Pre, E, Post\rangle$, where E denotes the UI event trace performing the functionality, and *Pre* and *Post* denote the pre- and post-conditions before and after executing E, respectively. For example, for the DMF "Rename Folder" in Figure 1, E denotes the event trace of "Rename Folder" (*i.e.*, open the menu of folder "old-name", select "Rename", input "new-name" and click "Save" in Figure 1(l)~(o)), *Pre* denotes the precondition (*i.e.*, the menu icon of folder "old-name" on Figure 1(l) should exist), and *Post* denotes the postcondition (*i.e.*, the folder name should be "new-name" on Figure 1(o)). In this way, we can apply the classic idea of PBT to generate a number of random inputs (*i.e.*, random UI event traces in our context) on the app under test to validate the correctness of ϕ. If there exists one random input (which leads to an erroneous program state) satisfying *Pre* but violating *Post* after executing E, a bug is found. For example, the property of the DMF "Rename Folder" is falsified by the event trace in Figure 1(l)~(o) because the folder name is not correctly updated to "new-name".

However, applying the preceding high-level idea to fuzz DMFs is not straightforward. We face two *key* technical challenges: (1) how to explore *diverse* app states to adequately validate the property of a DMF? (2) how to define and check the property of a DMF which involves app data changes? To address these two challenges, we have two *key* observations (detailed in Section 2.2): (1) exploring the mutual interactions between DMFs *w.r.t.* the shared app data

could lead to diverse app states (*e.g.*, in Figure 1, the three DMFs "Create folder", "Search folder", and "Rename folder" *all* manipulate the shared app data "old-name" and thus manifest the DME), and (2) the property of a DMF can be characterized by some data update effect (*e.g.*, for the DMF "Rename Folder", the data update effect is changing "old-name" to "new-name") *and* checked by the data update effect displayed on the UI pages (*e.g.*, in Figure 1(o), when folder "old-name" is renamed to "new-name", "new-name" should be displayed on the UI page).

Inspired by the preceding two observations, given some type of app data of interest, our idea is to (1) randomly interleave the relevant DMFs (and other possible events) to explore diverse app states, thus improving the fault detection ability; and (2) characterize the property of a DMF by a (data) model-based property ϕ=$\langle Pre, E, R, Post\rangle$, where R records the data update effect of E in the abstract data model, thus facilitating property checking. Specifically, the data update effect of these DMFs are recorded in the model in parallel when different DMFs are interleaved during fuzzing. And we leverage the consistency between recorded app data and UI pages as the handler to check the correctness of DMFs. As a result, we can do property testing whenever the execution of a DMF is finished. In practice, we focus on testing the DMFs in terms of five common data manipulation operations[1], *i.e.*, *create*, *read*, *update*, *delete* and *search*. The properties of these DMFs (*i.e.*, DMF specifications) are provided by human according to app features. Overall, the *novelty* of our technique is combining the idea of *property-based testing* with *model-based properties* [31] to facilitate validating DMFs in the context of Android apps.

Evaluation and results. We implemented a GUI testing tool, PBFDroid, to support the application of our property-based fuzzing approach. We applied PBFDroid to find DMEs in 20 popular Android apps (17 open-source and 3 industrial apps), and successfully found 30 unique and previously unknown bugs in 18 apps. Specifically, among the 30 bugs, 29 bugs are DMEs. Out of the 29 DMEs, 22 are non-crashing logic bugs, and 7 are crash bugs. To date, 19 DMEs have been confirmed and 9 have already been fixed by the app vendors. The remaining bugs are still under active discussions between developers. Our further evaluation confirms that *none* of the 22 non-crashing DMEs can be found by the state-of-the-art testing techniques. Moreover, we conducted a user study to investigate the feasibility of manually specifying DMF specifications. It shows that the involved manual cost is reasonable. On average, it only takes 3 minutes to specify one single DMF and 13 minutes for all the DMFs *w.r.t.* the app data of interest per app. Overall, given accurate DMF specifications, our approach can automatically find DMEs *without* false positives.

To sum up, this paper has made the following contributions:

- We introduce a property-based fuzzing approach to finding DMEs, especially the non-crashing ones. Our approach combines the idea of property-based testing with model-based properties to achieve effective fuzzing for DMFs.
- We propose a novel idea, *i.e.*, randomly interleaving different DMFs and other possible events to generate diverse app states, for improving the fuzzing effectiveness.
- We implement a GUI testing tool, PBFDroid, to support the application of our approach. PBFDroid successfully found 30 unique and previously unknown bugs, 29 of which are DMEs (22 are non-crashing logic bugs). PBFDroid significantly complements existing GUI testing techniques.

[1]In our work, we differ *read* and *search*: *read* denotes viewing existing data entries, while *search* denotes retrieving data entries with some criterion.

2 APPROACH

This section presents our approach, a property-based fuzzing approach to finding DMEs in Android apps.

2.1 High-level Idea

An Android app A is a UI-based, event-driven program. A UI page is represented by a *UI layout* L, which contains a number of UI views (or widgets). A *UI view* w ($w \in L$) has some attributes, *e.g.*, `className` (*i.e.*, the view type), `resourceId` (*i.e.*, the view id), `text` (*i.e.*, the view text) and *etc.*

Definition 1. ***UI event***. *A UI view w can be executed by a* UI event e. *We denote e as $e = (t, w, o)$, where $e.t$ denotes the event type (*e.g., `click`, `edit`*), $e.w$ denotes the UI view w on which e is executed, and $e.o$ denotes the optional data of e (*e.g., *the text inputted by* `edit`*).*

Examples. In Figure 1, page (c) corresponds to a UI layout L. On L, an edit field view w exists (its `className` is `EditText`, and `text` is "old-name"). The event e of inputting "old-name" in w can be represented as e = (`edit`, w,"old-name").

Definition 2. ***Test Inputs for Apps***. *An app A accepts as a test input in the form of a sequence of UI events (*i.e., *event trace). An event trace E can be denoted as E=$[e_1, \ldots, e_i, \ldots, e_n]$, where e_i is an event. When E is executed on A, we can obtain a sequence of the app states S,* i.e., *S=$[s_0, \ldots, s_{i-1}, s_i, \ldots, s_n]$, or denoted by $s_0 \xrightarrow{e_1} s_1 \ldots s_{i-1} \xrightarrow{e_i} s_i \ldots s_{n-1} \xrightarrow{e_n} s_n$, where s_i is the app state due to the execution of e_i on s_{i-1} ($1 \le i \le n$). We use s_n=$E(s_0)$ to denote the effect of executing E on the initial app state s_0 and reaching the ending state s_n.*

Definition 3. ***App Property***. *Given some app functionality F, its app property ϕ is distilled from the specification of an app A. Specifically, ϕ is represented in the form of pre- and post-conditions,* i.e., *ϕ=$\langle Pre, E, Post\rangle$, where E denotes F in the form of an event trace, Pre is the precondition imposing a necessary condition that must hold before execution of E, and $Post$ is the postcondition defining the effect on the app state after executing E. Specifically, an app property ϕ can be interpreted as, if $s \models Pre$ and s'=$E(s)$, $s' \models Post$ should hold (s and s' are the states before and after the execution of E, respectively).*

Property-based Testing. Property-based testing (PBT) [15] validates the correctness of a piece of program under test against some specified properties (*i.e.*, the test oracles [6]). Specifically, it generates a large number of test inputs to check whether the properties could be violated. For example, for a function `sort` which takes as input an array `arr` and returns the sorted `arr` with its elements in the ascending order, we can specify one of its property as `arr[i]` $\le$ `arr[j]` (where $0 \le i \le j \le N-1$, N is the length of `arr`). The idea of PBT is to generate a number of arrays with different sizes and elements (*e.g.*, integers) to validate whether the property holds.

Our high-level idea. *Our high-level idea* is to leverage the idea of property-based testing to validate app properties. Given an app A

Table 1: Pre- and post-conditions, semantics, and data update rule of DMFs in the five common data manipulation operations.

op	Pre^{op}	Semantics of E^{op}	R^{op}	$Post^{op}$
Create	$e_1.w \in L_0$	Create a new data object $d, d \notin D_0$	$D_n := D_0 \cup \{d\}$	$\exists w.d \mapsto w \wedge w \in L_n$
Read	$e_1.w \in L_0$	Read the details of any data object d	-	$\exists w.d \mapsto w \wedge w \in L_n$
Update	$e_1.w \in L_0$	Update any data object d to $d', d' \notin D_0$	$D_n := D_0 \setminus \{d\} \cup \{d'\}$	$(\neg\exists w.d \mapsto w \wedge w \in L_n) \wedge (\exists w'.d' \mapsto w' \wedge w' \in L_n)$
Delete	$e_1.w \in L_0$	Delete any data object d	$D_n := D_0 \setminus \{d\}$	$\neg\exists w.d \mapsto w \wedge w \in L_n$
Search	$e_1.w \in L_0 \wedge D_0 \neq \emptyset$	Search any data object $d \in D_0$	-	$\exists w.d \mapsto w \wedge w \in L_n$

and one of its app properties ϕ=$\langle Pre, E, Post\rangle$, we aim to validate the correctness of ϕ. Specifically, we aim to generate different app states $s \in \mathcal{S}$, *s.t.* $s \models Pre$, and check whether $s' \models Post$ always holds after the execution of E. Here, s'=$E(s)$ and $\mathcal{S}$ denote the app states of A. If $s' \models Post$ does not hold for some s, we are guaranteed to find a violation of ϕ (when ϕ is correctly defined).

2.2 Challenges and Observations

However, instantiating the high-level idea for validating the properties of DMFs (*i.e.*, the ϕ in our setting) is non-trivial. We face two technical challenges: (1) how to explore *diverse* app states to adequately validate the property of the DMF? and (2) how to define and check the property of the DMF which involves app data changes? To address these two challenges, We define some concepts in this section to illustrate our two key observations. Section 2.3 will present our approach based on the observations.

App Data, Data Type and Data Object. Following the common Model-View-Controller (MVC) architecture pattern [24], an Android app A accepts UI events from users to manipulate app data and visualizes the data on the UI pages. We observe that an app A may contain different types of app data from the perspective of app functionalities. Each type of app data $\mathcal{T}$ is usually associated with some DMFs. These DMFs manipulate the concrete data objects of $\mathcal{T}$ to achieve some functionality. Specifically, a *data object* d (*i.e.*, an instance of $\mathcal{T}$) is usually associated with some UI view w on some UI layout L for visualization. Here, we use $d \mapsto w$ to denote the mapping from a data object d to its corresponding view w.

Examples. In Figure 1, page (d) displays the created folder "old-name", which can be viewed as a data object of data type "Folder Name" (hereafter we use "Folder" for short). Specifically, the data object "old-name" is visualized by a text view w (its `text` is "old-name", and `resourceId` is "`amaze:id/firstline`") on the page (d). This relation can be denoted by "old-name" $\mapsto w$. On page (d), the current app data of type "Folder" can be recorded as {"old-name"}.

Based on the preceding concepts, we have two observations.

***Observation 1*: Exploring the mutual interactions between DMFs *w.r.t.* the shared app data could lead to diverse app states**. We observe that, given some type of app data, its relevant DMFs can affect each other by manipulating the shared data objects, which may lead to different app states. This observation is similar to the insight of interleaving class method calls on some shared class objects to improve testing object-oriented programs [46].

Examples. In Amaze, the three DMFs, *i.e.*, "Create Folder", "Search Folder" and "Rename Folder", work fine independently. However, when these three DMFs interact with each other on the shared data object "old-name" (see the event trace in Figure 1), an erroneous app state is manifested.

***Observation 2*: The property of a DMF can be characterized by some data update effect and checked by the data update effect displayed on the UI pages.** We observe that, for a functionally correct app, its each DMF leads to specific impacts on the app data. Therefore, we can record these data update effects to define the properties of these DMFs. Additionally, based on the common MVC architecture pattern adopted by Android [24], the data update effect will be always displayed on the app's UI pages. Therefore, we can check the property of a DMF by checking the consistency between the recorded app data and the data displayed on the UI pages.

Example. In Amaze, the execution of DMF "Create Folder" (see Figure 1) leads to the addition of a new folder to the app data, *i.e.*, adding a data object "old-name" to the app data of type "Folder". Therefore, the data update effect of executing DMF "Create Folder" can be represented as from $\emptyset$ to {"old-name"}. After the execution of DMF "Create Folder", the data object "old-name" is visualized on page (d), which denotes the DMF "Create Folder" achieve its functionality correctly.

2.3 Instantiation of the High-level Idea

Inspired by the preceding two observations, our instantiated idea is that, given one type of app data $\mathcal{D}$ of interest, we randomly interleave the relevant DMFs (with other possible events) on the shared data objects to explore diverse app states for thorough validation. Meanwhile, we record the data update effect of each DMF when different DMFs are interleaved and leverage the consistency between the app data and UI layouts as the handler to check the correctness of DMFs.

Definition 4. ***App State***. *An app state is composed of two key components: (1) the UI layout L for front-end visualization,* and *(2) the app data $\mathcal{D}$ stored in the background (e.g., in files or database). In this way, we can represent an app state as s=$\langle L, \mathcal{D}\rangle$, where L and $\mathcal{D}$ are the UI layout and the app data at the app state s, respectively.*

Model-based properties of DMFs. We specify the model-based property of a DMF as ϕ^{op}=$\langle Pre^{op}, E^{op}, R^{op}, Post^{op}\rangle$, where op denotes one data manipulation operation, (*i.e.*, $op \in$ {Create, Read, Update, Delete, Search}). Here, E^{op} is the event trace performing the DMF. Pre^{op} and $Post^{op}$ are the pre- and post-conditions, respectively. Specifically, we introduce D, an abstract data model, to record the app data of type $\mathcal{T}$ manipulated by the DMF. D is updated by R^{op}, which runs the semantics of E^{op}. Table 1 gives the model-based properties of the five types of DMFs.

How to do property checking on one DMF? We take Create in Table 1 as an example to illustrate how to do property checking on one DMF. Let the event trace E^{op} of Create be $[e_1, \ldots, e_i, \ldots, e_n]$. When E^{op} is executed on some app state s_0, we can obtain a sequence of app states S=$[s_0$=$\langle L_0, \mathcal{D}_0\rangle, \ldots, s_i$=$\langle L_i, \mathcal{D}_i\rangle, \ldots, s_n$=$\langle L_n, \mathcal{D}_n\rangle]$ (*cf.* Definition 2 and 4). Note that the precondition Pre^{op} of Create, $e_1.w \in L_0$ (e_1 is the first event of E^{op}, and $e_1.w$ is e_1's target UI view), decides whether Create is executable on

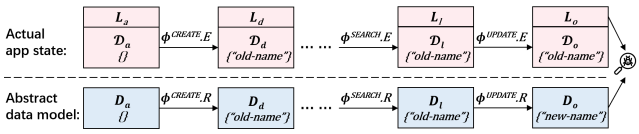

Figure 2: Illustration of our approach on the bug in Figure 1.

state s_0 (recall that s_0=$\langle L_0, \mathcal{D}_0 \rangle$). During the execution of CREATE, the abstract data model D is independently updated according to the semantics of E^{op}, *i.e.*, we can obtain a sequence of mirrored app data, *i.e.*, $[D_0, \ldots, D_i, \ldots, D_n]$. When CREATE is successfully executed (*i.e.*, all the events of E^{op} are executed), R^{op} records data update effect by adding d into D_n ($d \notin D_0$), and the post-condition $Post^{op}$ checks the consistency between $\mathcal{D}_n$ and D_n via the UI layout L_n. Specifically, $Post^{op}$ checks whether there exists one UI view w (w is mapped from the data object d) on the ending page L_n, which displays the data object d. If w does not exist, the property is violated and a DME is found. In this way, we can leverage D to record the app data changes and achieve property checking whenever one DMF is executed. Note that for UPDATE and DELETE in Table 1, if $d \notin D$, $D \setminus \{d\}$ does not delete any element in D.

How to do property checking on multiple DMFs? We use Figure 2 to demonstrate the process of property checking on multiple DMFs, using the example depicted in Figure 1. Figure 2 shows that how our approach records the app update effect when interleaving different DMFs, and does property checking whenever the execution of a DMF is completed. Starting from the app state s_a (corresponding to page (a) in Figure 1), the execution of the three DMFs "Create Folder", "Search Folder" and "Rename Folder" updates the app state changes from $s_a = \langle L_a, \mathcal{D}_a \rangle$ to $s_d = \langle L_d, \mathcal{D}_d \rangle$, $s_l = \langle L_l, \mathcal{D}_l \rangle$ and finally $s_o = \langle L_o, \mathcal{D}_o \rangle$, where the L_a, L_d, L_l and L_o correspond to the UI layout of page (a), (d), (l) and (o) in Figure 1, and $\mathcal{D}_a$, $\mathcal{D}_d$, $\mathcal{D}_l$ and $\mathcal{D}_o$ represent the actual app data, respectively. The abstract data model, D, is updated from $\emptyset$ to {"old-name"}, {"old-name"} and finally {"new-name"} according to the executions of these three DMFs. During this process, we can utilize the consistency between the recorded data in D and the UI layouts to perform property checking after each DMF is executed. In this way, we can find that the property of DMF "Rename Folder" is violated because a UI view displaying the data object "new-name" does exist on L_o.

3 IMPLEMENTATION

We implemented an automated GUI testing tool, PBFDROID, to support the application of our property-based fuzzing approach. Figure 3 shows PBFDROID's workflow. PBFDROID takes as input the app under test, assists users to conveniently specify the DMFs of interest, and outputs any found DMEs (with bug-reproducing tests). PBFDROID contains four modules: (1) *DMF instantiator*, (2) *input generator*, (3) *data recorder*, and (4) *property checker*. In the following, we first present the main module *input generator* (Section 3.1), *data recorder* (Section 3.2), and *property checker* (Section 3.3), which implement our core approach. We discuss *DMF instantiator* (Section 3.4) at last, which is a user interaction module.

3.1 Input Generator

Input generator drives the main testing loop (described in Algorithm 1), and coordinates with *data recorder* and *property checker*

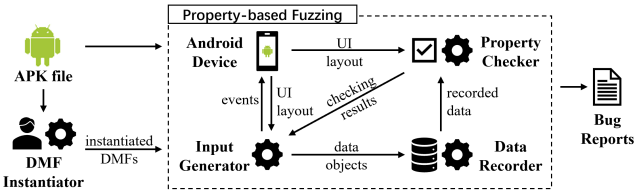

Figure 3: Workflow of PBFDROID

Algorithm 1: Property-based Fuzzing for finding DMEs

Inputs : DMFList: app A's relevant DMFs *w.r.t.* some app data type
Output : DMEs: the found DMEs (a list storing the bug-reproducing tests)

```
1  Procedure Main:
2      while not timeout do
3          D ← ∅; eventTrace ← []
4          clearAndRestartApp(A);
5          while eventTrace.len() < L_max do
6              L ← dumpGUILayout(A);
7              candidateDMFs ← [];
8              foreach dmf ∈ DMFList do
9                  if isPreconditionHold(dmf, L, D) then
10                     candidateDMFs.append(dmf)
11             if rand(0,1) > 0.5 ∧ candidateDMFs ≠ ∅ then
12                 dmf ← randomSelect(candidateDMFs);
13                 succ, events ← execute(A, dmf.E);
14                 eventTrace ← eventTrace :: events;
15                 if ¬succ then
16                     continue;
17                 D ← updateAppData(dmf, D);
18                 L ← dumpGUILayout(A);
19                 if ¬isPostconditionHold(dmf, L, D) then
20                     DMEs.append(eventTrace);
21             else
22                 e ← randomEvent(L, candidateDMFs);
23                 succ, events ← execute(A, [e]);
24                 if succ then
25                     eventTrace ← eventTrace :: events;
26     return DMEs;
```

during fuzzing. In detail, *input generator* is responsible for generating and executing random GUI tests. It randomly interleaves the relevant DMFs *w.r.t.* some data type and other possible events to explore diverse app states. Note that the input generator is *compositional* like the data generators in classic property-based testing. Because it calls sub-generators to generate random string (for the text inputs of `edit` events), random types of events (*e.g.*, `click` or `long-click`), and composes a sequence of events (*i.e.*, a GUI test).

Specifically, D is the abstract data model and *eventTrace* records the executed events during testing. In the main testing loop (Lines 2-26), PBFDROID first initializes D and *eventTrace*, and prepares the app under test by clearing its app data (Lines 3-4), and then iteratively generates GUI tests to fuzz the app (controlled by L_{max}, the maximal length of a GUI test). PBFDROID checks which DMFs (stored in *candidateDMFs*) are qualified to be executed (Lines 7-10). Here, `isPreconditionHold` checks whether one DMF's precondition holds according to the current app state. To randomly interleave the DMFs and other possible events, PBFDROID randomly selects one DMF when *candidateDMFs* is not empty (Lines 12-20) *or* a random UI event (Line 11) by coin flipping. If one DMF is selected, all events in the event trace of this DMF ($dmf.E$ here) will be executed by `execute` (Line 13). Note that `execute` returns two variables: (1) *succ*

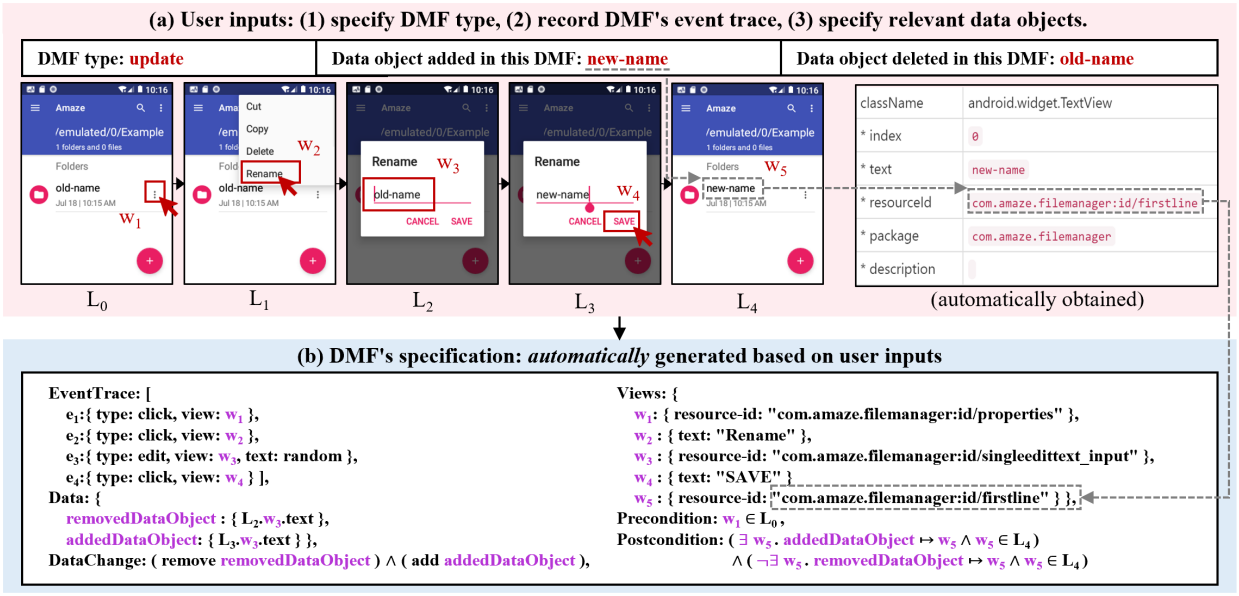

Figure 4: Defining the DMF specification of "Rename Folder" with the assistance of PBFDroid: (a) the given user inputs, (b) the *automatically* generated DMF specification.

which indicates whether the DMF is successfully executed, and (2) *events* which records the successfully executed events.

If the event trace of this DMF is successfully executed, the function `updateAppData` updates the abstract data model D (Line 17). Then, function `isPostconditionHold` checks whether the DMF's postcondition holds according to the UI layout L and the abstract app data model D (Line 19). If the property is violated, the event trace *eventTrace* (which records all the executed events) will be stored as a bug-reproducing test (Line 20). However, if the DMF fails to execute (Lines 15-16), D will not be updated, and the GUI testing will continue until reaching the limit L_{max} (Lines 5-25). And if a random event e is selected and successfully executed, this event will be recorded in *eventTrace* (Lines 22-25). Note that function `randomEvent` randomly selects one executable event e on the current layout L, and e should not be the first event of any DMF in *candidateDMFs*. In practice, PBFDroid uses `UIAutomator` [69] to execute events and obtain UI layouts.

3.2 Data Recorder

Data recorder maintains the abstract data model D according to the executed DMFs (*cf.* R^{op} in Table 1) by `updateAppData`. Specifically, it *only* updates D when a DMF is successfully executed. For example, only when the DMF "Rename Folder" is completely executed, the data object "old-name" will be replaced by "new-name" in D. *Data recorder* records the data update effect of each DMF to facilitate property checking whenever a DMF is executed.

3.3 Property Checker

Property checker checks whether the precondition or postcondition of a DMF is satisfied based on the UI layout and the simulated app data (*cf.* Pre^{op} and $Post^{op}$ in Table 1) by `isPreconditionHold` and `isPostconditionHold`. For example, when the DMF "Rename Folder" is successfully executed, this module will check whether there exists a UI view on the layout displaying the data object ("new-name"). If the UI view does not exist, the postcondition is violated.

3.4 DMF Instantiator

DMF instantiator assists users to conveniently define DMFs. Figure 4 uses the DMF "Rename Folder" to illustrate the basic procedure. As shown in Figure 4(a), when a user selects "Rename Folder" as the target DMF, the user needs to tell PBFDroid about (1) DMF type (*e.g.*, Create, Read, Update, Delete, Search), (2) DMF's event trace, and (3) the manipulated data objects. In this case, the DMF type of "Rename Folder" is Update, and the event trace of "Rename Folder" will be *automatically* recorded when the user interactively executes the events on the device screens (the views of events are annotated in the red boxes on L_0, L_1, L_2, L_3). The manipulated data objects are specified by the user in text according to the DMF type. In this case, the removed data object is "old-name" (on L_2), and the added data object is "new-name" (on L_3). Based on the preceding user inputs, this module *automatically* generates the DMF specification (defined in a `JSON`-style domain specific language). Figure 4(b) shows the generated DMF specification of "Rename Folder". Specifically, it contains six main fields: *EventTrace*, *Data*, *DataChange*, *Views*, *Precondition*, and *Postcondition*. We explain these fields as follows.

- ***EventTrace***. This field records the event trace of a DMF (corresponding to E^{op} in Table 1). Each event has its type, target view, and optional data. In this case, the event trace of DMF "Rename Folder" contains four events (e_1, e_2, e_3, e_4). Specifically, e_3 is an edit event, its target view is w_3 (annotated on L_2 and specified in field *Views*), and its data is *random* (which informs *input generator* to generate a random string with letters, numbers or symbols). When creating a DMF, the event trace should ensure that the pre- and the post-condition of the property could be observed on the starting page (*e.g.*, L_0) and the ending page (*e.g.*, L_4), respectively.

- ***Data***. This field defines the manipulated data objects (corresponding to d and d' in Table 1). In this case, the DMF "Rename Folder" has one removed and one added data objects, which are specified by view w_3's `texts` on L_2 and L_3, respectively.
- ***DataChange*** This field defines the data update effect *w.r.t.* the DMF type (corresponding to R^{op} in Table 1). In this case of "Rename Folder", the data update effect is removing *removedDataObject* and adding *addedDataObject* in the app data D (*removedDataObject* and *addedDataObject* are specified in field *Data*).
- ***Views***. This field defines the UI views related to the DMF. Each view can be identified by some of its attributes (*e.g.*, `className`, `resourceId`, `text`). These views and their attributes are automatically collected according to the recorded events or user inputs.
- ***Precondition*** and ***Postcondition***. These two fields define the pre- and post-conditions of a DMF (corresponding to Pre^{op} and $Post^{op}$ in Table 1). The event trace of DMF can be executed only when the precondition is satisfied, while the postcondition checks whether the property of DMF is valid after the event trace of DMF is successfully executed.

Note that the mapping from data objects to the relevant views are automatically identified by this module according to the user inputs. For example, when a user specifies "new-name" as the added data object, this module will analyze UI layouts (L_0~L_4) to find the relevant views whose `text` is "new-name". The grey dotted boxes and arrows in Figure 4 show how this module maps the user-specified text ("new-name") to the relevant view (w_5) and records this view in the DMF specification. w_5 is used in the postcondition. PBFDroid uses WEDITOR [73] to automatically record user events.

Discussion. In practice, some DMFs may require additional predicates in its precondition to accurately model their properties. For example, in Amaze, an app user can create a folder under any directory rooted by "`/storage/emulated/0`". We know that, if a user creates a folder under "`/storage/emulated/0/A`" but searches it under "`/storage/emulated/0/B`", the user will not find the folder. It is the expected app behavior. Thus, to accurately define the DMF "Search Folder" of Amaze, we manually added an additional predicate into its precondition (*i.e.*, the search directory should be "`/storage/emulated/0`"). In detail, we added a new view w_0:`{text: "/storage/emulated/0"}` to field $Views$ of the generated DMF specification, and added the predicate $w_0 \in L_0$ in field $Precondition$. We observe that among the 101 DMFs used in our experiment in Section 4, 32 DMFs (31.6%) require adding additional preconditions.

4 EVALUATION

Our evaluation aims to answer these research questions:

- **RQ1** : Can PBFDroid find DMEs in real-world Android apps, including open-source and industrial apps?
- **RQ2** : Can the state-of-the-art techniques find the DMEs uncovered by PBFDroid? Can PBFDroid complement them?
- **RQ3** : How do different testing strategies or configurations affect the bug finding abilities of PBFDroid?
- **RQ4** : How much manual effort is required to define DMFs with the assistance of PBFDroid?

Table 2: App subjects evaluated in our study (K=1,000; M=100,000; B=100,000,000)

App ID	App Name	Version	#Stars	#Installations	App Feature	Target Data (#DMFs)
1	Markor	2.8.6	2.2K	100K-500K	Text Editor	File(5)
2	Aard2	0.51	314	10K-50K	Dictionary Reader	Word (4)
3	SimpleTask	10.9.3	475	10K-50K	Task Manager	Task (4)
4	SkyTube	2.980	1.6K	100K-500K	Video Player	Channel (5)
5	AnyMemo	10.11.7	140	100K-500K	Learning Software	Card (7)
6	Amaze	3.6.7	3.9K	1M-5M	File Manager	Folder (7)
7	AnkiDroid	2.16	5.3K	10M-50M	Learning Software	Card (7)
8	Wikipedia	2.7.5	1.7K	50M-100M	Wikipedia Reader	Favorite (5)
9	Tasks	12.5	2.2K	100K-500K	Task Manager	Task (5)
10	RadioDroid	0.84	485	100K-500K	Radio Manager	Radio (5)
11	ActivityDiary	1.4.2	67	1K-5K	Activity Recorder	Activity (5)
12	MyExpenses	3.3.7	442	1M-5M	Expense Tracker	Account (5)
13	Antennapod	2.7.1	4.6K	500K-1M	Podcast Manager	Podcast (5)
14	Materialistic	3.3	2.2K	100K-500K	News Browser	Story (4)
15	Notepad	3.0.3	252	500K-1M	Note Manager	Note (5)
16	Transistor	4.1.1	420	10K-50K	Station Browser	Station (4)
17	Omni Notes	6.1.0	2.5K	100K-500K	Note Manager	Note (4)
18	TikTok	20.5.0	-	1B-5B	Video Platform	User (5)
19	CapCut	7.8.0	-	100M-500M	Video Editor	User (5)
20	Feishu	5.14.6	-	10M-50M	Collaboration Platform	Folder (5)

4.1 Evaluation Setup

App subjects. To our knowledge, Genie [62] and Odin [70] are the *only* two work which can automatically find *generic* non-crashing bugs in Android apps. They are close to our work. Thus, we selected *all* the apps from these two work as our subjects (selecting these apps also allows a fair comparison between PBFDroid, Genie and Odin in RQ2). Specifically, Genie and Odin evaluated 12 and 17 apps, respectively. After removing the duplicated apps, we obtained 18 apps. From the 18 apps, we excluded 3 apps, *i.e.*, UnitConverter and Fosdem (because their functionalities are too simple to contain any DMF) and And-Bible (because it is not usable due to the unavailable Web services). To complement the subjects, we additionally selected 2 popular open-source apps (Notepad and OmniNotes) from [63], and 3 industrial closed-source apps (TikTok [66], CapCut [11], FeiShu [23]) from ByteDance. Thus, we have 20 apps in total, all of which are released on Google Play [25]. In Table 2, "Version", "#Stars", "#Installations" and "App Feature" give the latest app versions at the time of our study, the numbers of stars on GitHub, the installations on Google Play, and the main app feature, respectively. These apps are representative with diverse features and most of them are popular. Note that our approach is not limited to specific app types.

DMF specifications. According to the app feature, we selected one major data type per app and defined the specifications of the relevant DMFs. Specifically, we defined the DMFs in terms of Create, Read, Update, Delete and Search. Given the target data type, we defined one DMF of Create, Read, and Search, respectively, and all the DMFs of Update and Delete. Because all the Update and Delete may affect the app data added by Create. Thus, some apps (*e.g.*, Amaze) may have more than five DMFs as they have multiple DMFs for Update and Delete, while others (*e.g.*, Aard2) may have fewer than five DMFs as they do not have some types of DMFs. For example, one major data type of the app Amaze (in Figure 1) is "Folder", and we defined its seven relevant DMFs (*i.e.*, "Create Folder", "View Folder", "Rename Folder", "Delete Folder", "Hide Folder" , "Unhide Folder" and "Search Folder"). In Table 2, "Data Type (#DMFs)" gives the selected data type for which we defined the DMFs and the number of defined DMFs. Note that we selected only one data type because it is already enough to demonstrate the feasibility and effectiveness of our technique (see the results of

RQ1). It is not a conceptual or technical limitation of our technique. If we consider more data types (and more DMFs), we expect to find more DMEs. The numbers of selected data types or DMFs are orthogonal to our approach, and do not affect PBFDroid's scalability or effectiveness. In practice, the exact number of DMFs we can test for an app is decided by the relevant app features.

Evaluation setup of RQ1. We ran the 17 open-source apps on Android emulators (Pixel XL, Android 8.0) deployed on a 64-bit Ubuntu 18.04 machine (64 cores, AMD 2990WX CPU, and 64GB RAM), and the 3 industrial apps on one real Android device (OPPO A11s, Android 10.0). We allocated 48 machine hours for extensively testing each app, and configured the maximum length of one GUI test generated by PBFDroid as 100 events. For each app, we manually inspected all the reported bugs to find the distinct ones which have different bug-triggering tests and manifestations. Then, we reported each distinct bug to the app vendors. Each bug is provided with a bug-reproducing test and video to ease confirmation and diagnosis. If a bug is not marked as duplicated by the app vendors, we regard it as a unique bug.

Evaluation setup of RQ2. Two major categories of automated GUI testing tools for Android exist based on the oracle problem [6]. One category of tools [22, 27, 50, 62, 63, 70] can automatically find non-crashing bugs. In this category, Genie [62] and Odin [70] are the only two tools which can find *generic* non-crashing bugs, which may find DMEs. Thus, we selected Genie and Odin for comparison. We did not select other tools because they target other specific types of non-crashing bugs rather than DMEs. We will discuss the differences between these work and ours in Section 6. The other category of tools (*e.g.*, Monkey [44]) is limited to crash bugs due to the lack of strong test oracles [6]. Although PBFDroid focuses on finding non-crashing bugs, we still selected Monkey [44] for comparison. Specifically, we allocated the same 48 hours for Genie, Odin and Monkey to test each of the 17 open-source apps in the same experimental environment. Genie and Odin were setup with their default settings in their original papers [62, 70]. We manually inspected all the bugs reported by Genie, Odin and Monkey to see whether they can find the DMEs uncovered by PBFDroid.

Evaluation setup of RQ3. PBFDroid interleaves different DMFs and other possible events to generate diverse program states. Thus, we investigated whether different interleaving strategies could affect the bug finding abilities of PBFDroid. Specifically, we setup two baseline strategies for comparison:

- **Baseline A**. Baseline A only tests *one single DMF without* interleaving with other possible events. Specifically, given one DMF, we manually identify the shortest event trace from the app starting page to the GUI page satisfying the precondition of the DMF (*e.g.*, Figure 1(a)~(d) is a qualified event trace for the DMF "Rename Folder"). During testing, Baseline A always (1) follows this event trace to reach this DMF, (2) executes the events of the DMF, (3) randomly generates necessary text inputs for some events (*e.g.*, `Edit`) of the DMF, and (4) checks the postcondition.
- **Baseline B**. Baseline B interleaves *one single DMF with other possible events*. It corresponds to Algorithm 1 when the input `DMFList` only contains one DMF under test.

We allocate the same testing time (48 hours per DMF in each of the 17 open-source apps) and experimental environment for Baseline A and B, and compare them with PBFDroid. Baselines A and B will not terminate before reaching the time bound because they are set up for running continuously.

Additionally, our experiment sets the default maximum length of each GUI test L_{max} (see Algorithm 1, Line 5) as 100 events. L_{max} constrains the possible interleavings between DMFs and other possible events, and thus may affect the tool's effectiveness. Thus, we evaluated the other two configurations of L_{max}, *i.e.*, 200 and 400 events per test, respectively, with the same testing time (48 hours per app) and experimental environment to test the 17 open-source apps, and compared the numbers of found bugs. Roughly, 40K GUI events can be generated in the 48-hour testing time for each app.

Evaluation setup of RQ4. We recruited 10 graduate students to study the manual cost of defining DMF specifications. The number of participants in our study is similar to those of prior relevant work [12, 80] (which recruited 8 and 12 students, respectively). All these students major in software engineering and have basic knowledge on Android, and none of them was from the environment of PBFDroid developers or this paper's authors. They voluntarily participated in this study and signed the informed consent [68]. Note that it is reported that graduate students can represent professionals (*e.g.*, developers or testers) in the software engineering experiments [54]. In this study, we let the participants define the same set of DMFs used in RQ1~RQ3. Specifically, we selected the app *Markor*, a text editor app, from our subjects to record a video tutorial (about 30 minutes) illustrating how to define DMF specifications. To avoid biases, we excluded Markor from the study. For the remaining 16 open-source apps in Table 2, we randomly assigned each participant eight apps (at the same time we make sure each app is evaluated by five different participants to ensure diversity). To mimic the realistic testing environment in which app developers or testers are familiar with app functionalities, each participant was provided with the necessary information about the DMFs via recorded videos, which explain the UI steps of DMFs and their functionalities (see the discussion paragraph in Section 3.4). During the study, we recorded the whole process of participants' activities on a desktop, and counted their time spent on defining each DMF specification (including the time of generating DMFs by interacting with *DMF instantiator*, adding additional necessary preconditions to the generated specification, and self-checking the defined DMFs). We validated the accuracy of the final DMF specifications.

4.2 Results for RQ1

Effectiveness of PBFDroid. Table 3 shows the bug finding results. It gives the app name, the bug ID, the bug state (*fixed*, *confirmed*, or *pending*), related DMFs (which are *necessary* for bug manifestation), the length of minimal bug-reproducing test (in the number of UI events), consequence and a brief description of the bug. PBFDroid found 30 unique and previously unknown bugs in 18 out of the 20 tested apps. Out of these 30 bugs, 29 bugs are DMEs, of which 22 are non-crashing bugs and 7 are crash bugs. To date, 19 bugs have been confirmed and 9 have already been fixed, while the remaining are still pending (none of them was rejected). The remaining bugs are still under active discussions between developers. Specifically, 24 DMEs require the combination of two or more DMFs for bug

Table 3: Bug finding results of PBFDroid.

App Name	Issue ID	Issue State	Related DMFs	#Steps	Consequence	Description
Markor	#1652	Fixed	Create,Search	11	Wrong behavior	No files can be searched in the root directory
	#1668	Fixed	Create,Update	8	Wrong behavior	Renaming will fail if new name contains "?"
	#1695	Fixed	Create,Update,Search	8	Data loss	Renaming will overwrite the same case sensitive name files
	#1698	Fixed	Search	6	Crash	Rotating the screen after searching causes a crash
	#1699	Fixed	Create,Update	6	Crash	Rotating the screen while editing will cause a crash
Aard2	#140	Fixed	Create,Search	7	Wrong behavior	Symbols in search text are ignored when searching
SimpleTask	#1156	Confirmed	Create,Search	9	Wrong behavior	Searching again after canceling a search will not work
SkyTube	#1045	Pending	Create,Delete,Search	9	Wrong behavior	The function of clearing the blocked channel list is unstable
	#1044	Confirmed	Create,Search	4	Infinite loading	Refreshing video list after blocking any channel causes infinite loading
AnyMemo	#524	Pending	Create,Update	6	Update delay	The card list is not updated in time after editing any card
	#523	Pending	Create,Search	12	Data loss	The "Reset All Preferences" option will delete the added card
Amaze	#3207	Fixed	Create,Search	9	Infinite loading	Searching for hidden folders causes crashes or infinite loading
	#3298	Confirmed	Create,Search,Update	14	Wrong behavior	Renaming will fail on the search results page
	#3357	Confirmed	Create,Search	7	Wrong behavior	Search results are not sorted by relevance
AnkiDroid	#10431	Fixed	Create,Read	14	Wrong behavior	Cards that use the wrong card template will show up empty
	#11626	Confirmed	Create,Update,Read	12	Wrong behavior	Cards cannot be edited when their type is changed to cloze
	#11280	Fixed	Create	10	Crash	Saving an empty video in card causes a crash
Wikipedia	#T305555	Fixed	Create,Update,Search	11	Update delay	Cannot search the favorites by new name after renaming the favorites
Tasks	#1869	Confirmed	Create,Read	7	Wrong behavior	Tasks can be filtered by other criteria but not by date
RadioDroid	#1099	Pending	-	4	Crash	Long-pressing the radio information in the history causes a crash
ActivityDiary	#310	Fixed	Search	6	Crash	Rotating the screen after entering invalid date in the search bar cause a crash
	#311	Pending	Create,Delete	11	Crash	Deleted activity cannot be recovered correctly
Materialistic	#1463	Pending	Read	4	Crash	Rotating the screen before selecting "zoom in or zoom out" causes a crash
NotePad	#134	Pending	Create,Read	13	Wrong behavior	The layout is inconsistent using the "right to left layout" setting
Transistor	#432	Pending	Create	7	Crash	Pressing the keyboard's "next" key while editing causes a crash
Omin Notes	#886	Confirmed	Create,Delete,Search	14	Wrong behavior	Deleted items can still be searched
TikTok	-	Confirmed	Create,Read	11	Wrong behavior	Videos of blocked users can still be seen in the recommendation page
	-	Pending	Create,Read	9	Update delay	The status is not updated in time after unblocking the user
CapCut	-	Confirmed	Create,Read	11	Wrong behavior	Videos of blocked users can still be seen in the recommendation page
FeiShu	-	Pending	Create,Update,Search	14	Update delay	Cannot search the folder by new name after renaming the folder

manifestation. These results show that PBFDroid is effective. Moreover, we received positive feedback from app developers, who commented the found bugs are important and non-trivial. For example, SkyTube [58]'s developer commented "*This bug is really important and annoying*", and AnkiDroid [3]'s developer commented "*It's a lot more complicated than I thought at first glance*".

Diversity of bugs found by PBFDroid. PBFDroid found 22 non-crashing DMEs of different consequences. We classified them into four types and illustrate some assorted bug samples. (1) *Unexpected wrong behaviors* (14/22 bugs): 14 DMEs lead to unexpected wrong app behaviors. For example, SimpleTask [56], a task management app, has a search function. If a user searches again after canceling a previous search, the user will not be able to find any matching results. (2) *Delayed data update* (4/22 bugs): 4 DMEs lead to delayed data update. For example, in Wikipedia [75], if a user changes the name of the favorite, the user will not be able to search the favorites via the new name for a long time. (3) *User data loss* (2/22 bugs): 2 DMEs lead to severe user data loss. For example, AnyMemo [4], a flashcard learning app, has a creation function for users to add their own cards. However, due to an error in this function, if a user selects the option "reset all preferences" in the app setting, all the user-added cards will be unexpectedly deleted from the database. (4) *Infinite loading* (2/22 bugs): 2 DMEs lead to infinite loading. For example, in Skytube [58], if a user adds a channel to the blocked list and then refreshes the video list, the app will enter into infinite loading and do not respond anymore.

4.3 Results for RQ2

We find that Genie, Odin and Monkey missed all the 22 non-crashing DMEs found by PBFDroid, and they only found 1, 1 and 3 out of the 8 crash bugs found by PBFDroid, respectively. We further analyzed the results of Genie and Odin to understand the reasons why they missed all the non-crashing DMEs. We find two major reasons. First, *their automated oracles are limited in finding DMEs*. Genie finds non-crashing bugs based on the *independent view property*, that is interacting with one GUI view should not affect the states of the others and only adds additional GUI effects. However, we find many of the DMEs are not within the scope of this generic oracle. For example, the Amaze's bug in Figure 1 cannot be detected by Genie as no independent view exists for Folder "old-name". Odin uses a heuristic oracle to find non-crashing bugs, that is appending the same events to the test inputs terminating at similar GUI layouts, and clustering the manifested behaviors to identify the minorities as suspicious bugs. However, one abstraction rules used by Odin ignores the texts displayed on the GUI pages when clustering. As a result, it cannot identify those bugs which lead to incorrect displayed texts. For example, in Figure 1(o), the folder "old-name" is not correctly renamed, which thus cannot be captured by Odin. Second, *their randomly generated tests are of low quality*. Many tests generated by Genie and Odin did not cover meaningful app functions. As a result, it is difficult for them to explore diverse app states to find DMEs, which may require long and specific event traces (see #Steps in Table 3). As a side note, Genie and Odin reported many false positives, which took us much time for inspection. These results show that PBFDroid can complement the state-of-the-art techniques in finding DMEs.

4.4 Results for RQ3

Table 4 compares Baseline A, B and PBFDroid in the numbers of found bugs. Baseline A and B only found 1 and 14 bugs, respectively. Moreover, *all* the bugs found by Baseline A and B were found by PBFDroid. Thus, PBFDroid is much more effective than Baseline A and B. *It indicates that interleaving different DMFs and other possible events is* crucial *for improving the bug finding ability.*

Baseline B is limited due to two major reasons. First, most of the DMEs (82.7%≈24/29) requires combining two or more DMFs for bug manifestation. When interleaving one single DMF with other possible events, Baseline B may not be able to cover other DMFs by random exploration. For example, to find the bug of "Rename folder" in Figure 1, Baseline B has to create a folder "old-name", switch

Table 4: Comparison between Baseline A, Baseline B, and PBFDroid, . "C" and "NC" represent crash and non-crashing bugs, respectively.

Tool	Baseline A		Baseline B		PBFDroid	
Failure Type	C	NC	C	NC	C	NC
#Bugs	0	1	8	6	8	22

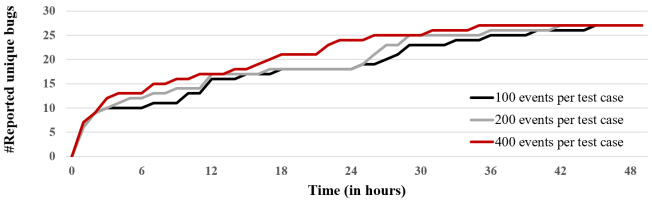

Figure 5: Number of found unique bugs under different L_{max}.

to the up-level directory and search folder "old-name" via random exploration, which is difficult. In our experiment, Baseline B only finds 6 bugs which require two DMFs. Second, because Baseline B only considers one single DMF, it cannot simulate the data changes when other DMFs are covered by chance. For example, if Baseline B tests a Search DMF, the data added by Create cannot be checked because that data will not be recorded. Baseline A is further limited because most DMFs only fail in some corner cases when running independently. However, Baseline A cannot reach such cases due to the lack of some random events. For example, AnkiDroid's card editing function fails only when the default card type is changed (which can only be covered by a random event).

Comparison between different L_{max}s. Figure 5 compares the tool effectiveness under the configurations of 100, 200 and 400 events per test, which are denoted by the black, grey, and red curves, respectively. The horizontal axis denotes the 48-hour testing time, and the vertical axis denotes the number of found unique bugs. *We can see that the three configurations of L_{max} do not have distinct impacts on tool effectiveness.* When L_{max} is set as 400, PBFDroid can find bugs more quickly (because it may interleave more DMFs in one GUI test), but at last the three configurations found the same numbers of unique bugs on the 17 open-source apps.

4.5 Results for RQ4

Figure 6 shows the time cost of defining one single DMF (denoted by the white boxes) and all the DMFs (denoted by the red boxes) for each of the 16 open-source apps evaluated by five different participants, respectively. The horizontal axis gives the app names and the vertical axis gives the time in minutes. From Figure 6, we find that the time cost of defining one single DMF across all the 16 apps ranges from 1~6 minutes with an average of 3 minutes, and the time cost of defining all the DMFs per app ranges from 5~23 minutes with an average of 13 minutes. Because some apps (*e.g.*, AnyMemo) have more DMFs than the other apps. They took more time than the others. Overall, the time cost of defining the DMF specifications is reasonable. *All* the participants commented that PBFDroid is convenient to use. As a side note, we find that most of the DMF specifications defined by participants, *i.e.*, 361 out of the 400 (≈90.3%) DMF specifications, are accurate. We analyzed the 39 inaccurate DMFs and found the inaccuracies were caused by some trivial mistakes of participants, *e.g.*, clicking a wrong UI widget when recording a DMF, confused with the *resourceId* and *description*

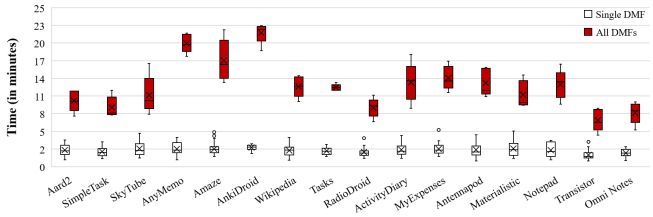

Figure 6: Time cost of participants in our study spent on defining one single DMF and all the DMFs per app.

fields of a UI widget when adding additional preconditions. We believe these issues are orthogonal to our technique and can be mitigated by better engineering (*e.g.*, giving warning messages).

5 DISCUSSION

This section provides an extended discussion of our work in the following three aspects: (1) DMF specifications (2) generability of our approach, and (3) threats to validity.

DMF specifications. In our experiment, we carefully inspected the app features when defining DMFs. Thus, we did not incur any false positives. Defining DMF specifications requires human participation (similar to writing program specification in formal verification), but the involved effort is one-time. Even if an app upgrades, only a slight revision is needed. Moreover, our examination on the 30 found bugs shows that these bugs are nontrivial and escaped from developer testing for a long time. On average, they had been hidden for 28 months and affected 23 release versions. Thus, the benefits outweigh the spent effort.

Generability of our approach. Despite our approach focusing on finding DMEs. This property-based fuzzing (PBF) approach has broader applicability. When the property ϕ=$\langle Pre, E, Post\rangle$ is instantiated on other app functionalities, PBFDroid can help automatically find the other types of bugs in Android apps. For example, if we define the property of some registration functionality in the app, our approach can check whether the registration can always succeed. In the future, we will explore how to extend property-based testing for more generic app properties [77].

Threats to validity. The first threat is the representativeness of app subjects. To this end, our study includes different categories of open-source and industrial apps, many of which are popular on GitHub and Google Play. Thus, they can represent real-world apps. The second threat is the human factors in the user study, which may cause inaccuracies or biases. To this end, we instructed the participants by providing informative tutorials and video-recorded all the activities of participants during the study to carefully measure the time cost. Moreover, each DMF is evaluated by five different participants to mitigate possible biases.

6 RELATED WORK

Finding non-crashing bugs for Android apps. Most automated GUI testing tools [19, 26, 41, 42, 44, 60, 71] use runtime exceptions as the oracle [21, 59]. Therefore, they *cannot* find the non-crashing DMEs which our work focuses on. To tackle the oracle problem, three major categories of testing techniques have been proposed. Table 5 compares the representative work in these categories, which we explain as follows. The first category of prior

Table 5: Differences between existing testing tools and ours in finding non-crashing bugs of Android apps.

Tool	What to provide before testing?	Types of test oracle
Genie	-	Metamorphic relations
Odin	-	Implicit knowledge
AppFlow	Modular tests	Pseudo-oracles
ChimpCheck	Example-based Tests	Assertions
PBFDroid	Specification	Model-based specification

work [1, 28, 50, 62, 63, 78] adopts metamorphic relations to generate automated oracles. However, most of these work targets specific types of non-crashing bugs (*e.g.*, system setting related defects [63, 64], data loss issues [28, 50]). Only Genie [62] targets *generic* non-crashing bugs, but it missed all the non-crashing DMEs due to the limitation of its metamorphic relation (discussed in Section 4.3). The second category uses differential analysis [70] or testing [22, 40]. Odin [70] finds non-crashing bugs by differing the buggy and normal app behaviors based on the heuristic oracle "bugs as deviant behaviors" [20]. But it missed all the non-crashing DMEs due to the limitation of its abstraction rule (discussed in Section 4.3). Some work [22, 40] uses differential testing to find (non-crashing) compatibility bugs based on different devices. They *cannot* find DMEs. The third category [7, 30, 39, 53] synthesizes the tests for one app (manually constructed) to test another app with similar features. AppFlow [30] synthesizes from the modular (abstract) tests. However, these work does not target DMEs. The synthesis accuracy is not high (which may lead to many false positives) [79]. In contrast, PBFDroid can automatically generate GUI tests without false positives.

To our knowledge, ChimpCheck [36] is the *only* work applying property-based testing for Android apps. However, ChimpCheck and PBFDroid have two key differences. First, ChimpCheck's main contribution is its novel UI trace generators, which can fuse example-based tests with random testing (*e.g.*, Monkey [44]) to generate test inputs. It does not focus on finding non-crashing bugs (including DMEs). Second, ChimpCheck uses the *assertions* in the user-provided *example-based tests* as the oracle. PBFDroid uses the user-specified *model-based properties* as the oracle. Due to the limited expressiveness, the assertions are difficult to validate DMFs when different DMFs are interleaved in our setting. Therefore, if adapted in our scenario (by manually providing example-based tests for the DMFs), ChimpCheck can only reach the ability of Baseline B in our evaluation(see Section 4.4).

Property-based testing and model-based testing. Property-based testing (PBT) was popularized by Claessen and Hughes [16]. Specifically, the two key elements of PBT are (1) the data generators *and* (2) the property specifications. In our work, the data generator is designed to generate random UI-based event traces (by interleaving the DMFs and other events), and the properties are specified in the model-based properties [31] (an abstract model capturing data update effect of the DMFs in the apps). Our work is different from the prior PBT work in two aspects. First, the prior work designs the data generators for their own specific domains [34, 37, 45, 47, 48, 55, 55], which cannot be applied for Android apps. Second, these prior work specifies the properties in the *assertions* [34, 37, 47, 48, 55] or temporal logic formula [45, 55], which are not suitable (or expressive) to capture the data update effect in our setting. To our knowledge, few work in the literature combines PBT with the model-based properties [31].

On the other hand, using an abstract model as the specification is the foundation of model-based testing (MBT) [9]. A lot of work [2, 5, 18, 26, 33, 38, 60] exists in applying MBT to test Android apps. However, the models in these work are the finite state machine based UI models (in which each state is an abstract UI layout and each edge is a UI event), which are used to guide test generation rather than deriving the oracles. These models are different from the model in our work. Therefore, all of these MBT work [2, 5, 18, 26, 38, 60] cannot find non-crashing bugs.

Finding the errors related to CRUD. Our work finds software errors related to the CRUD operations. Costa *et al.* [17] model the "Find" operation (*i.e.*, Search) in Android apps as one UI test pattern to check its correctness. Mariani *et al.* [43] specify the CRUD operations of Web apps in Alloy specification language and build a tool Augusto to generate semantic tests with oracles. However, Augusto *only* tests one CRUD operation independently, and does not consider their combinations. Since a direct comparison is infeasible (as Augusto targets Web apps), Baseline A in our evaluation (see Section 4.4) can be viewed as a similar implementation of Augusto. Our approach which interleaves different DMFs of CRUD operations is shown to be much more effective than Baseline A. Some work [51, 52] uses metamorphic testing to find the CRUD errors in database management systems.

7 CONCLUSION

We introduce a property-based fuzzing approach to effectively finding DMEs, which combines the idea of property-based testing and model-based properties. Given some type of app data, we interleave different DMFs and other possible events to generate diverse app states for validation. The realization of our idea, PBFDroid, has successfully discovered 30 previously unknown bugs from 20 popular Android apps. Among these bugs, 22 are non-crashing DMEs, which cannot be found by the state-of-the-arts. So far, 19 have been confirmed and 9 fixed by the developers. The majority (20 out of 25) cause non-crashing failures and are hard to be detected by state-of-the-art automatic testing tools.

DATA AVAILABILITY

We have made all the artifacts (including PBFDroid and its source code, the defined DMF specifications of all app subjects, the data of the user study) publicly available at: *https://github.com/property-based-fuzzing/home*.

ACKNOWLEDGMENTS

We thank the anonymous ESEC/FSE reviewers for their valuable feedback. This work was supported in part by NSFC Grant 62072178, National Key Research and Development Program (Grant 2022YFB3104002), "Digital Silk Road" Shanghai International Joint Lab of Trustworthy Intelligent Software under Grant 22510750100, National Key Research and Development Program (Grant 2020AAA0107800), and the Shanghai Collaborative Innovation Center of Trusted Industry Internet Software.

REFERENCES

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

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

[3] AnkiDroid Team. 2022. *AnkiDroid.* Retrieved 2023-1 from https://github.com/ankidroid/Anki-Android.

[4] AnyMemo Team. 2022. *AnyMemo.* Retrieved 2023-1 from https://github.com/helloworld1/AnyMemo.

[5] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 238–249. https://doi.org/10.1145/2970276.2970313

[6] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering (TSE)* (2014), 507–525. https://doi.org/10.1109/TSE.2014.2372785

[7] Farnaz Behrang and Alessandro Orso. 2019. Test migration between mobile apps with similar functionality. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 54–65.

[8] Benoît Joossen. 2018. *Voice Memos "Epidemic Failure" and How to Avoid it.* Retrieved 2023-1 from https://aeroquartet.com/wordpress/2019/03/05/voice-memos-epidemic-failure-and-how-to-avoid-it/.

[9] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner. 2005. *Model-based testing of reactive systems: advanced lectures.* Springer Science & Business Media.

[10] Business of Apps. 2022. *Android Statistics (2022).* Retrieved 2023-1 from https://www.businessofapps.com/data/android-statistics/.

[11] CapCut Team. 2021. *CapCut.* Retrieved 2023-1 from https://lv.faceueditor.com.

[12] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 665–676. https://doi.org/10.1145/3180155.3180240

[13] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. 2020. *Metamorphic testing: a new approach for generating next test cases.* Technical Report. HKUST-CS98-01, Hong Kong University of Science and Technology. https://doi.org/10.48550/arXiv.2002.12543

[14] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 429–440. https://doi.org/10.1109/ASE.2015.89

[15] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP)*. 268–279. https://doi.org/10.1145/351240.351266

[16] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming (ICFP)*. 268–279. https://doi.org/10.1145/357766.351266

[17] Pedro Costa, Ana CR Paiva, and Miguel Nabuco. 2014. Pattern based GUI testing for mobile applications. In *Proceedings of the 9th International Conference on the Quality of Information and Communications Technology (QUATIC)*. 66–74. https://doi.org/10.1109/QUATIC.2014.16

[18] Guilherme de Cleva Farto and Andre Takeshi Endo. 2015. Evaluating the model-based testing approach in the context of mobile applications. *Electronic notes in Theoretical computer science* (2015), 3–21. https://doi.org/10.1016/j.entcs.2015.05.002

[19] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. 1–12. https://doi.org/10.1145/3377811.3380402

[20] Dawson R. Engler, David Yu Chen, and Andy Chou. 2001. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP 2001)*. 57–72. https://doi.org/10.1145/502034.502041

[21] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). 408–419. https://doi.org/10.1145/3180155.3180222

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

[23] FeiShu Team. 2021. *FeiShu.* Retrieved 2023-1 from https://www.feishu.cn/.

[24] GeeksforGeeks. 2020. *MVC (Model View Controller) Architecture Pattern in Android.* Retrieved 2023-1 from https://www.geeksforgeeks.org/mvc-model-view-controller-architecture-pattern-in-android-with-example/.

[25] Google. 2021. *Google Play.* Retrieved 2023-1 from https://play.google.com/store.

[26] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 269–280. https://doi.org/10.1109/ICSE.2019.00042

[27] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and fixing data loss issues in Android apps. In *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*. 605–616.

[28] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. 2022. Detecting and fixing data loss issues in Android apps. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 605–616. https://doi.org/10.1145/3533767.3534402

[29] Hacker News. 2013. *Tell Facebook: There's a severe bug when changing profile pics on the iOS app.* Retrieved 2023-1 from https://news.ycombinator.com/item?id=6456285.

[30] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 269–282. https://doi.org/10.1145/3236024.3236055

[31] John Hughes. 2020. How to Specify It!. In *Trends in Functional Programming*. Springer International Publishing, 58–83.

[32] k-9 Team. 2022. *k-9.* Retrieved 2023-1 from https://github.com/thundernest/k-9.

[33] Stefan Karlsson, Adnan Causevic, Daniel Sundmark, and Mårten Larsson. 2021. Model-based Automated Testing of Mobile Applications: An Industrial Case Study. In *14th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW 2021*. 130–137.

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

[35] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Trans. Reliability* 68, 1 (2019), 45–66. https://doi.org/10.1109/TR.2018.2865733

[36] Edmund SL Lam, Peilun Zhang, and Bor-Yuh Evan Chang. 2017. ChimpCheck: property-based randomized test generation for interactive apps. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*. 58–77. https://doi.org/10.1145/3133850.3133853

[37] Leonidas Lampropoulos, Michael Hicks, and Benjamin C. Pierce. 2019. Coverage guided, property based testing. *Proc. ACM Program. Lang.* OOPSLA (2019), 181:1–181:29. https://doi.org/10.1145/3360607

[38] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 23–26. https://doi.org/10.1109/ICSE-C.2017.8

[39] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. 2019. Test transfer across mobile apps through semantic mapping. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 42–53. https://doi.org/10.1109/ASE.2019.00015

[40] Ying-Dar Lin, José F. Rojas, Edward T.-H. Chu, and Yuan-Cheng Lai. 2014. On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. *IEEE Trans. Software Eng.* (2014), 957–970. https://doi.org/10.1109/TSE.2014.2331982

[41] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. 2022. Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 135:1–135:5. https://doi.org/10.1145/3551349.3559505

[42] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*. 94–105. https://doi.org/10.1145/2931037.2931054

[43] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. 2018. Augusto: Exploiting Popular Functionalities for the Generation of Semantic GUI Tests with Oracles. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. 280–290. https://doi.org/10.1145/3180155.3180162

[44] Monkey Team. 2022. *Android Monkey*. Retrieved 2023-1 from https://developer.android.com/studio/test/monkey.

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

[46] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *29th International Conference on Software Engineering (ICSE)*. 75–84. https://doi.org/10.1109/ICSE.2007.37

[47] Sumit Padhiyar and KC Sivaramakrishnan. 2021. ConFuzz: Coverage-guided property fuzzing for event-driven programs. In *Practical Aspects of Declarative Languages: 23rd International Symposium, PADL 2021, Copenhagen, Denmark, January 18-19, 2021, Proceedings 23*. 127–144. https://doi.org/10.1007/978-3-030-67438-0_8

[48] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: coverage-guided property-based testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 398–401. https://doi.org/10.1145/3293882.3339002

[49] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 153–164. https://doi.org/10.1145/3395363.3397354

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

[51] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* OOPSLA (2020), 211:1–211:30. https://doi.org/10.5281/zenodo.4032401

[52] Manuel Rigger and Zhendong Su. 2020. Testing Database Engines via Pivoted Query Synthesis. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 667–682. https://doi.org/10.48550/arXiv.2001.04174

[53] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2018. Automation of android applications functional testing using machine learning activities classification. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 122–132. https://doi.org/10.1145/3197231.3197241

[54] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are students representatives of professionals in software engineering experiments?. In *Proceedings of the 2015 IEEE/ACM 37th IEEE international conference on software engineering (ICSE)*. 666–676. https://doi.org/10.5555/2818754.2818836

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

[56] SimpleTask Team. 2022. *SimpleTask*. Retrieved 2023-1 from https://github.com/mpcjanssen/simpletask-android.

[57] Sixth Tone. 2019. *E-Commerce App Loses 'Tens of Millions' From Coupon Glitches*. Retrieved 2023-1 from https://www.sixthtone.com/news/1003483/e-commerce-app-loses-tens-of-millions-from-coupon-glitches.

[58] SkyTube Team. 2022. *SkyTube*. Retrieved 2023-1 from https://github.com/SkyTubeTeam/SkyTube.

[59] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2022. Why My App Crashes? Understanding and Benchmarking Framework-Specific Exceptions of Android Apps. *IEEE Trans. Software Eng.* 48, 4 (2022), 1115–1137. https://doi.org/10.1109/TSE.2020.3013438

[60] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 245–256. https://doi.org/10.1145/3106237.3106298

[61] Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking automated GUI testing for Android against real-world bugs. In *29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 119–130. https://doi.org/10.1145/3468264.3468620

[62] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages (OOPSLA)* (2021), 1–31. https://doi.org/10.1145/3485533

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

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

[65] Amaze Team. 2022. *AmazeFileManager*. Retrieved 2023-1 from https://github.com/TeamAmaze/AmazeFileManager.

[66] Tiktok Team. 2021. *Tiktok*. Retrieved 2023-1 from https://www.tiktok.com.

[67] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal* 27, 1 (2019), 149–201. https://doi.org/10.1007/s11219-018-9418-6

[68] David Travis. 2020. *What user researchers ought to know about informed consent*. Retrieved 2023-1 from https://userfocus.co.uk/articles/what_user_researchers_ought_to_know_about_informed_consent.html.

[69] uiautomator2 Team. 2022. *uiautomator2*. Retrieved 2023-1 from https://github.com/openatx/uiautomator2.

[70] Jue Wang, Yanyan Jiang, Ting Su, Shaohua Li, Chang Xu, Jian Lu, and Zhendong Su. 2022. Detecting non-crashing functional bugs in Android apps via deep-state differential analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 434–446. https://doi.org/10.1145/3540250.3549170

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

[72] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 738–748. https://doi.org/10.1145/3238147.3240465

[73] weditor Team. 2022. *weditor*. Retrieved 2023-1 from https://pypi.org/project/weditor/.

[74] Wikipedia. 2022. *Create, read, update and delete*. Retrieved 2023-1 from https://en.wikipedia.org/wiki/Create,_read,_update_and_delete.

[75] Wikipedia Team. 2022. *Wikipedia*. Retrieved 2023-1 from https://github.com/wikimedia/apps-android-wikipedia.

[76] WordPress Team. 2022. *WordPress*. Retrieved 2023-1 from https://github.com/wordpress-mobile/WordPress-Android.

[77] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 1319–1331. https://doi.org/10.1145/3597926.3598138

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

[79] Yixue Zhao, Justin Chen, Adriana Sejfia, Marcelo Schmitt Laser, Jie Zhang, Federica Sarro, Mark Harman, and Nenad Medvidovic. 2020. FrUITeR: a framework for evaluating UI test reuse. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. 1190–1201. https://doi.org/10.1145/3368089.3409708

[80] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 128–139. https://doi.org/10.1109/ICSE.2019.00030