ROUTE: Roads Not Taken in UI Testing

JUN-WEI LIN, MGM Resorts International, USA
NAVID SALEHNAMADI and SAM MALEK, University of California, Irvine, USA

Core features (functionalities) of an app can often be accessed and invoked in several ways, i.e., through alternative sequences of user-interface (UI) interactions. Given the manual effort of writing tests, developers often only consider the typical way of invoking features when creating the tests (i.e., the "sunny day scenario"). However, the alternative ways of invoking a feature are as likely to be faulty. These faults would go undetected without proper tests. To reduce the manual effort of creating UI tests and help developers more thoroughly examine the features of apps, we present ROUTE, an automated tool for feature-based UI test augmentation for Android apps. ROUTE first takes a UI test and the app under test as input. It then applies novel heuristics to find additional high-quality UI tests, consisting of both inputs and assertions, that verify the same feature as the original test in alternative ways. Application of ROUTE on several dozen tests for popular apps on Google Play shows that for 96% of the existing tests, ROUTE was able to generate at least one alternative test. Moreover, the fault detection effectiveness of augmented test suites in our experiments showed substantial improvements of up to 39% over the original test suites.

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

Additional Key Words and Phrases: GUI test augmentation, test reuse, test amplification, mobile testing

ACM Reference format:
Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2023. ROUTE: Roads Not Taken in UI Testing. *ACM Trans. Softw. Eng. Methodol.* 32, 3, Article 71 (April 2023), 25 pages.
https://doi.org/10.1145/3571851

1 INTRODUCTION

By and large, existing automated test generation techniques [5, 7, 8, 14, 15, 22, 32–34, 36, 38, 45, 48] cannot generate high-quality tests, consisting of both inputs and proper assertions, as they lack developers' knowledge; instead, they generate inputs for exploring applications (apps) without providing proper assertions to verify the resulting behavior. Due to this limitation, the state-of-the-practice in mobile app testing is largely driven by feature-based **user-interface (UI)** tests that are manually developed. In contrast to the majority of automated test generation techniques that focus on improving the code coverage of an ***app under test (AUT)***, feature-based UI testing aims to improve the coverage of the *features* (functionalities) of an AUT. A prior study has showed that this type of testing is preferred by mobile app developers [31]. While it is straightforward to

This work was supported in part by award numbers 2211790, 1823262, and 2106306 from the National Science Foundation.
Authors' addresses: J.-W. Lin, 3600 S Las Vegas Blvd, Las Vegas, NV, 89109, USA; email: jlin@mgmresorts.com; N. Salehnamadi and S. Malek, 5019 Donald Bren Hall, Irvine, CA, 92697, USA; emails: {nsalehna, malek}@uci.edu.

1049-331X/2023/04-ART71 $15.00
https://doi.org/10.1145/3571851

conduct feature-based UI testing manually, developers may choose to write scripted or automated UI test cases to make such testing repeatable in the context of continuous integration [16].

Automated feature-based UI testing has several advantages: reliability, execution speed, and in particular, the manifestation of developers' knowledge regarding software functionality through oracles, i.e., assertions in test scripts. However, a recent study about test automation in open-source Android apps shows that within the real-world projects adopting automated UI testing, half of them contain fewer than eight UI test cases [29].

```
find_element_by_id("school/timetable").click()
find_element_by_id("more_options").click()
find_element_by_id("school/item_1").click()
e = find_element_by_id("school/title")
assertEquals(e.getText(), "Manage timetables"); // Oracle
```

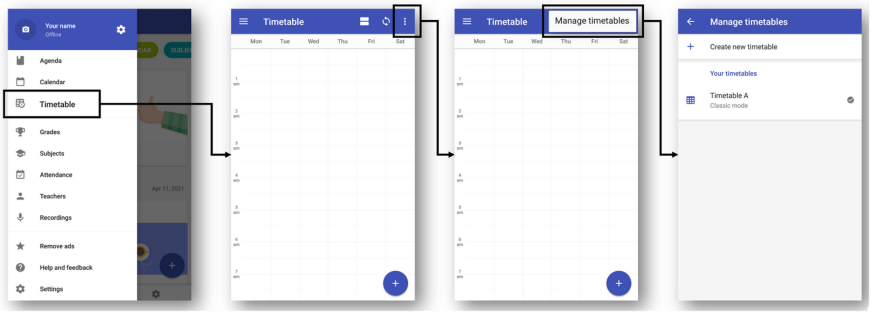
Listing 1. Test script for timetable management in School Planner.

We believe that the two observations in prior work—(1) the developers preferring feature-based test coverage but mostly relying on ad hoc manual testing [31] and (2) half of the open-source projects containing fewer than eight UI test cases [29]—are due to the same fact: The development of feature-based test scripts involves substantial manual effort. As a result, when creating the test scripts, developers often only consider the typical way of invoking a feature (i.e., the "sunny day scenario" or "happy path"), neglecting the alternative ways of invoking it. In fact, the functionalities examined by such manually developed tests are usually the core capabilities of an app that can be accessed in multiple ways, i.e., through alternative sequences of UI interactions.

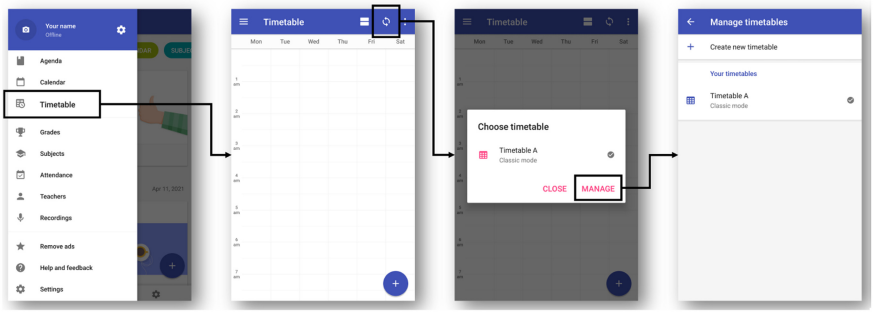
For illustration, Listing 1 shows a GUI test for a feature dealing with timetable management in School Planner, a popular planner app for students [4]. The execution of this test case is depicted in Figure 1(a). This test resembles the actions a user would take to manage timetables through a dedicated pop-up menu. The assertion in the last line checks if the app responds correctly by verifying the title of the page is "Manage timetables" (the last screen of Figure 1(a)). Nevertheless, Figures 1(b) and 1(c) demonstrate that this feature can be performed in two alternative ways, i.e., via the timetable selection dialog or Settings. When the developer writes the test for this feature, she may perceive the scenario of Figure 1(a) as the default or primary way of invoking this functionality. Failure to create tests for the other two ways of invoking the feature, however, leaves the potential faults that can only be revealed by those tests undetected. For instance, the first test shown in Figure 1(a) cannot reveal latent faults in the event listener method of the "MANAGE" button in Figure 1(b).

To reduce the manual effort of creating feature-based tests and help developers more thoroughly verify the features of their apps, we present Route, short for *ROads not taken in Ui TEsting*, which is an automated solution for feature-based UI test augmentation for Android apps. Route first takes a feature-based UI test, including both its inputs and assertions, and the AUT as input. It then applies novel heuristics to explore the AUT and generate additional UI tests that verify *the same feature* as the original test. Route leverages virtualization techniques to increase the accuracy and performance of app exploration. In other words, it saves and restores the snapshots of the memory of the device in certain states. In fact, Figures 1(b) and 1(c) are the alternative scenarios discovered by Route from the original test shown in Figure 1(a).

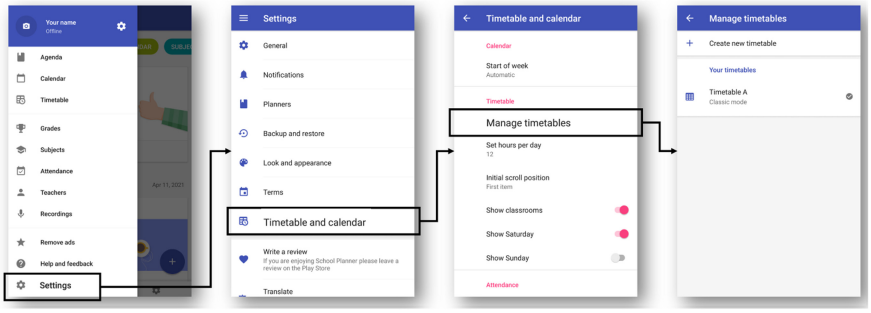
There are several differences between Route and the prior work in test augmentation [6, 11, 13, 18, 23, 27, 35, 41, 44, 46, 49, 50]. First, the proposed augmentation is feature-based. In other words, we aim to generate tests that verify the same functionalities as the original tests. To achieve this goal, we have developed several properties that the generated tests should hold and designed Route based on these properties. Second, the assertions in the original tests are reused in the generated tests, making the augmented test suites capable of detecting feature-related faults and



(a) Original scenario: manage timetables through a dedicated pop-up menu



(b) Alternative scenario: manage timetables through the timetable selection dialog



(c) Alternative scenario: manage timetables through Settings

Fig. 1. Different use-case scenarios for timetable management in the School Planner app.

providing the developers with actionable information to debug their programs. Finally, the test generation algorithm prioritizes the candidate tests according to how likely they are to exercise a feature in a different way than the existing tests.

We have applied ROUTE on several dozens of feature-based UI tests for verification of popular apps on Google Play. The experimental results show that for 96% of the existing tests, ROUTE was able to generate at least one alternative test. Moreover, the fault detection effectiveness of the augmented test suites was improved by up to 39% over the original test suites.

Overall, this article makes the following contributions:

- We propose a feature-based UI test augmentation technique capable of creating high-quality tests, consisting of both inputs and assertions, to verify features of an app in alternative ways.

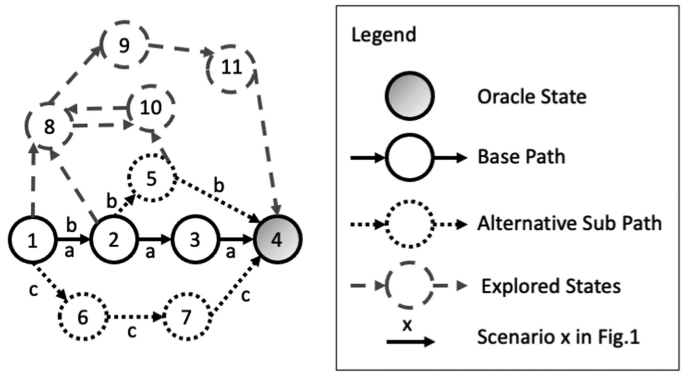


Fig. 2. Example of states and paths in the School Planner app shown in Figure 1.

- We present novel heuristics for the proposed test augmentation technique and implement them as an automated tool, called Route, which is publicly available [3].
- We empirically evaluate Route on real-world tests and demonstrate its utility to generate additional tests capable of detecting new faults.

The rest of this article is organized as follows: Section 2 elaborates on the concept of feature-based test augmentation. Section 3 provides an overview of Route as well as the implementation details of its components. Section 4 presents our test generation algorithm. Section 5 discusses the evaluation results. The article concludes with an overview of the related research and future work.

2 BACKGROUND

An existing UI test can be augmented by modifying its execution path. In this article, we model the dynamic behavior of the AUT as a graph $G = (V, E)$, where:

- V is a set of GUI states (screens). Each $v \in V$ represents a unique runtime GUI state in the AUT.
- E is a set of edges between the GUI states. Each $e = (v_i, v_j, (w, a)) \in E$ represents a transition from v_i to v_j by firing a GUI event that performs an action a on a widget w.

Furthermore, the execution of a test can be perceived as a traversal through the graph. Specifically, for a UI test t, we represent its execution path $p = (v_s, v_f, V_p, E_p, V_o)$ as follows:

- $V_p \subseteq V$ are GUI states visited by t with edges $E_p \subseteq E$.
- $v_s \in V_p$ denotes the start state, i.e., the state before t executes the first event.
- $v_f \in V_p$ denotes the end state, i.e., the state after t executes the last event. v_s and v_f are also referred to as *terminal states*.
- $V_o \subseteq V_p$ denotes the *oracle states*, i.e., the states on which t has assertions. We also store the assertions associated with each $v_o \in V_o$.

Finally, for a usage-based test to be augmented, we call its execution path *base path* and the visited GUI states *base states*. For example, in Figure 2, the solid edges illustrate the base path of the test executing the scenario of Figure 1(a). The base path can then be modified to generate new tests from the original test.

When we construct a modified execution path p' from a base path p to generate a new test, we would like p' to still test the same functionality as p, albeit using an alternative path. To that end, we propose several properties that we would like the modified execution path p' to hold. First, the start and end states of p' should be the same as the base path p, because the terminal states provide

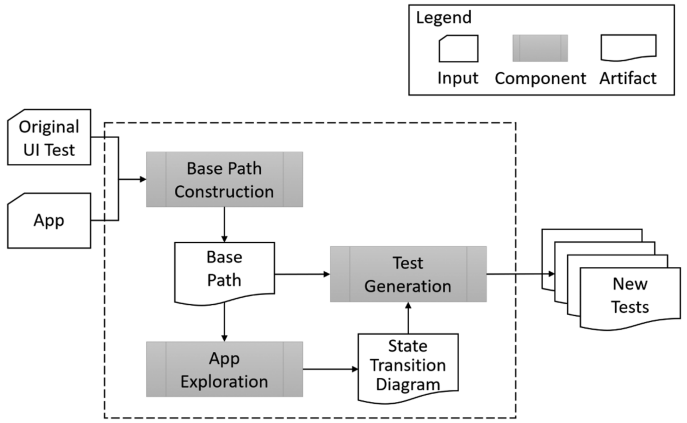


Fig. 3. Overview of ROUTE.

important information about the boundary of the tested feature. Similarly, the oracle states V_o in p, as well as the assertions examined at each state $v_o \in V_o$, should also be included in p', because they are critical for semantically verifying the functionality. Finally, p' should not be drastically different from p. The reason is that overly modifying p may significantly change the behavior of the AUT and invalidate the original assertions. As a result, we need to limit the changes from p to p' to specific types of operations, such as replacing a sub-path in p with an alternative simple path. The paths labeled "b" and "c" in Figure 2 (corresponding to the scenarios of Figures 1(b) and 1(c)) exemplify the modified execution paths satisfying the proposed properties. In the following sections, we describe how we designed ROUTE to integrate these properties into the generated tests.

3 APPROACH

Figure 3 provides an overview of ROUTE. It takes an original test and the AUT as input and generates new tests that examine the same feature as the original test in alternative ways. There are three main components in ROUTE: *Base Path Construction*, *App Exploration*, and *Test Generation*. First, Base Path Construction component executes the original test and retrieves its execution path as the *base path*. Next, based on the visited states in the base path, App Exploration component systematically explores the AUT and outputs the *state transition* diagram that represents the AUT's runtime behavior. Finally, Test Generation component applies novel heuristics to identify and prioritize the executable tests that hold the desired properties discussed in Section 2. We describe the implementation of each component in the following subsections.

3.1 Base Path Construction

Algorithm 1 describes how Base Path Construction component executes the original test t to obtain its execution path p as the base path. As defined in Section 2, p is a 5-tuple of v_s (start state), v_f (end state), V_p (base states), E_p (edges), and V_o (oracle states). The algorithm first initializes E and V_o as an empty set and then launches the AUT. Next, it dumps the current screen of the AUT to obtain the initial state before test execution as the start state and uses it to initialize base states and the variable for previous state (lines 3–6). A dumped screen of an Android app is a widget hierarchy tree in XML format, in which non-leaf nodes are layout widgets and leaf nodes are actionable or visible widgets, such as buttons and text views. We uniquely identify a GUI state by computing a hash value over the widget hierarchy tree.

ALGORITHM 1: Base Path Construction

Input:
t: original test
Output:
$p = (v_s, v_f, V_p, E_p, V_o)$: the base path of t

1: $E_p = \emptyset;\ V_o = \emptyset$ ▷ initialize edges and oracle states
2: $launchApp()$
3: $screen = dumpCurScreen()$
4: $v_s = getHash(screen)$ ▷ start state
5: $V_p = \{v_s\}$ ▷ initialize base states
6: $prevState = v_s$
7: $takeSnapshot(v_s)$
8: **for each** $event \in t$ **do**
9: **if** $event$ is not an assertion **then**
10: $execute(event)$
11: $screen = dumpCurScreen()$
12: $curState = getHash(screen)$
13: $V_p = V_p \cup curState$
14: $E_p = E_p \cup (prevState, curState, event)$
15: $prevState = curState$
16: $takeSnapshot(curState)$
17: **else** ▷ $event$ is an assertion
18: $V_o = V_o \cup prevState$
19: **end if**
20: **end for**
21: $v_f = prevState$ ▷ end state
22: $p = (v_s, v_f, V_p, E_p, V_o)$
23: **return** p

Route models GUI states by considering all the widgets and node attributes. In other words, we do not abstract away node attributes or values of text boxes, and any change in tree structure or node attribute value leads to a different state. This granularity is required, because sometimes the difference between an oracle state and its previous state is subtle, such as a change of the *checked* or *text* attribute of a node. To decrease the likelihood of dynamic content such as timestamps or ads that introduces inaccuracy in Route, we leverage virtualization when we need to revisit base states, i.e., the GUI states in the base path.

Before executing the test, a snapshot of the start state is taken (line 7). We leverage virtualization to save the visited states and later restore and explore them in App Exploration (detailed in the next subsection). In other words, the AUT is installed and executed on a **virtual machine (VM)** such as Android Virtual Device [20] or VirtualBox VM [39], such that the runtime program state of the AUT, including the underlying OS and emulated hardware, can be stored in a snapshot and fully resumed later. This helps improve the accuracy and performance of GUI state exploration. In prior work [5, 22, 33, 34, 45], a GUI state is resumed by restarting the AUT and replaying the recorded event sequence. However, a shortcoming of this restart-and-replay approach is that the background services and dynamic contents such as timestamps or ads may change after restarting the AUT and the GUI state cannot be reached again. Virtualization addresses this issue. Moreover,

the exploration of the AUT can be accelerated, because the snapshots can be restored and processed in parallel with multiple VMs. The practice of using virtualization and snapshots is also adopted by prior work in Android testing [15].

Algorithm 1 executes the original test t after the initialization steps (line 8). For each $event \in t$, if the event is not an assertion, then it first executes the event and then obtains $curState$, the current GUI state after execution (lines 9–12). We subsequently update the base path in terms of the base states V_p and edges E_p and take a snapshot of this new state (lines 13–16). However, if the event is an assertion,[1] then it means some checks are performed on the previous state, and we hence update the oracle states V_o with $prevState$ (lines 17 and 18). Finally, after the execution of t, the end state is updated and the base path p is returned as input to the next phase of App Exploration (lines 21–23).

3.2 App Exploration

With the base path p from the original test, App Exploration component performs *k-step lookahead* on each base node to obtain G, an explored state transition graph of the AUT, as described in Algorithm 2. In particular, the algorithm explores if there are paths with length not greater than k from a base state v to another base state. This base-path-directed exploration restricts the possible paths that can be constructed from the graph later, such that the generated tests will not deviate too much from the base path.

Algorithm 2 first initializes G, the graph to be returned, with the states and edges in the base path (line 1). Next, for each base state $v \in V_p$, it performs initialization steps in lines 3–9. It creates a first-in-first-out *queue* to store the event sequences that need to be executed for exploration (line 3). To initialize the queue, it retrieves all actionable widgets from v (line 4), creates an action event, such as *click*, for each of them, and enqueues the generated event sequences in *queue* (lines 5–9). For example, the solid nodes and edges in Figure 2 depict the initial G for the test in Listing 1. Moreover, for v equal to state 2 of Figure 2, *queue* is initialized with three single events, which when executed result in exploration of states 3, 5, and 8.

For each event sequence $events \in queue$, Algorithm 2 first restores the GUI state v and executes the events sequentially (lines 11–16). The executed events, as well as the encountered GUI states, are also recorded as $subPath$, a sub-path starting from v (lines 17–21). Next, if $curState$, then the GUI state reached after the execution is a base state, and it updates G with the traversed $subPath$ (lines 24 and 25). Otherwise, if the length of $events$ is smaller than the threshold k, then it means we can continue exploring forward from v. To that end, we retrieve all actionable widgets for $curState$ and create $newEvents$, a new event sequence by appending an appropriate action event such as *click* to each of the retrieved widgets (lines 26–29). $newEvents$ is then enqueued in *queue* for further exploration (line 30). For example, for $k = 2$ and v equal to state 2 in Figure 2, subpaths $(2 \rightarrow 3)$ and $(2 \rightarrow 5 \rightarrow 4)$ will be added to G, since states 3 and 4 belong to V_p. Note that some paths (explored states in Figure 2) are not added to G, because they do not visit any base states or they exceed the lookahead threshold, e.g., given $k = 3$, the path $(1 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow 4)$ is not explored. After all event sequences in *queue* are consumed for all base states v, the algorithm stops and returns G (line 35). We describe Test Generation in detail in the next section.

4 TEST GENERATION

Test Generation component takes the base path p and the state transition graph G as input and generates T, a set of executable tests with size n, as described in Algorithm 3. To that end, in

[1]The assertions considered in this article are UI-based assertions, such as checking the existence of widget or text, or verifying the attributes of a widget.

ALGORITHM 2: App Exploration

Input:
$p = (v_s, v_f, V_p, E_p, V_o)$: base path
k: lookahead threshold
Output:
G: state transition diagram of the AUT

```
1:  G = {V_p, E_p}
2:  for each v ∈ V_p do
3:      queue = ∅                                   ▷ A first-in-first-out queue
4:      widgetList = getActionable(v)
5:      for each widget ∈ widgetList do
6:          events = ∅
7:          events = events ∪ (widget, getAction(widget))
8:          queue.enqueue(events)
9:      end for
10:     while queue ≠ ∅ do
11:         restoreSnapshot(v)
12:         events = queue.dequeue()
13:         subPath = ∅
14:         prevState = v; curState = v
15:         for each e ∈ events do
16:             execute(e)
17:             screen = dumpCurScreen()
18:             curState = getHash(screen)
19:             if curState ≠ prevState then
20:                 subPath =
                        subPath ∪ (prevState, curState, e)
21:                 prevState = curState
22:             end if
23:         end for
24:         if curState ∈ V_p then
25:             G.update(subPath)
26:         else if events.length < k then
27:             widgetList = getActionable(curState)
28:             for each widget ∈ widgetList do
29:                 newEvents =
                        events ∪ (widget, getAction(widget))
30:                 queue.enqueue(newEvents)
31:             end for
32:         end if
33:     end while
34: end for
35: return G
```

ALGORITHM 3: Test Generation

Input:
$p = (v_s, v_f, V_p, E_p, V_o)$: base path
G: state transition diagram of the AUT
n: number of tests to be generated
Output:
T: a set of executable tests with size n

```
1:  T = ∅
2:  paths = getSimplePaths(G, v_s, v_f, V_o)
3:  T = generateTestsFromPaths(T, paths, V_o)
4:  if |T| < n then
5:      paths = getCyclicPaths(G, v_s, v_f, V_o)
6:      T = generateTestsFromPaths(T, paths, V_o)
7:  end if
8:  return T

9:  function generateTestsFromPaths(T, paths, V_o)
10:     paths = sort(paths)          ▷ Sort paths by their "distance" to base path p (Algorithm 4)
11:     for each p' ∈ paths do
12:         launchApp()
13:         executable = execute(p', V_o)
14:         if executable is true then
15:             T = T ∪ p'
16:             if |T| = n then
17:                 return T
18:             end if
19:         end if
20:     end for
21:     return T
22: end function
```

line 2, it retrieves all *simple paths* in G that (1) are from start state (v_s) to end state (v_f) and (2) visit all oracle states (V_o). We focus on the paths that satisfy these two constraints, because they are required properties for the generated tests (as discussed in Section 2). Moreover, tests with an execution path containing a cycle may include repetitive operations such as navigating back to a previous screen, and hence considered less useful. As a result, the algorithm first considers simple paths (i.e., paths without repeating states) when generating the tests.

Next, Algorithm 3 verifies the retrieved simple paths and generates executable tests from them (line 3) by calling `generateTestsFromPaths` function (line 9). This function first sorts the paths by their difference from the base path (line 10, detailed in the next subsection) and then launches the AUT and executes each of the prioritized paths to verify if it is executable (lines 11–13). If a candidate path p' is executable, then it is added to T (lines 14 and 15). Note that when verifying p', if an oracle state $v_o \in V_o$ is encountered, then the associated assertions are also performed. That way, the generated tests include the original assertions that have passed. `generateTestsFromPaths` function returns when the size of T is n (line 16) or all candidate paths are verified (line 21).

If the size of T is smaller than n after checking all the simple paths, then Algorithm 3 will further retrieve the execution paths containing only one cycle (lines 4 and 5). Specifically,

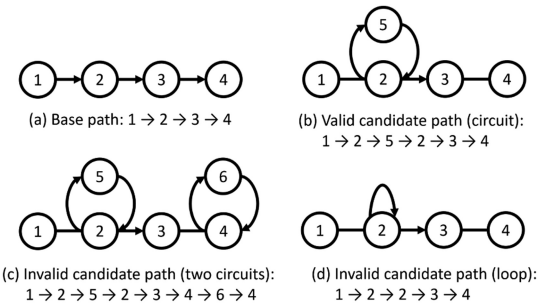


Fig. 4. Examples of valid and invalid paths considered by `getCyclicPaths()` in Algorithm 3.

ALGORITHM 4: Path Difference

```
1:  function pathDiff(p, p')
2:      L = getStateList(p)                          ▷ L = {v_1, v_2, ..., v_|L|}
3:      L' = getStateList(p')                        ▷ L' = {v'_1, v'_2, ..., v'_|L'|}
4:      d_1 = editDist(L, L')
5:      score = 0; count = 0
6:      for i from 2 to |L'| do
7:          if v'_i ∉ L then
8:              score = score + xmlEditDist(v'_i, v'_{i-1})
9:              count = count + 1
10:         end if
11:     end for
12:     d_2 = score/count
13:     return α × d_1 + (1 − α) × d_2
14: end function
```

`getCyclicPaths` function in line 5 only considers the paths with a non-loop circuit (i.e., a cycle with a length larger than one) on a base state, as illustrated in Figure 4. We only allow this special case for execution paths containing cycles, because we would like to modify the base path conservatively. The retrieved paths are then verified in the same way to generate more tests (line 6). The algorithm returns when the required number of tests are generated or after checking all valid candidate paths (line 8).

4.1 Computing Path Difference

With the explored state transition graph of the AUT, the number of candidate paths may grow exponentially and become too many to verify. As a result, Test Generation component prioritizes the candidate execution paths by their difference from the base path (line 10 in Algorithm 3). In other words, we compute a distance score between the base path p and each candidate path p' with the `pathDiff` function in Algorithm 4. The paths with the highest distance scores are executed and verified first.

We would like to select the candidate paths that are most different from the base path, as they are more likely to exercise a feature in a different way than the existing test. To that end, we consider the difference between a candidate path and the base path at both the *path level* and the *state level*. By path level, we mean how different the two paths are in terms of their length and visited states. At the state level, we further look into the new states in the candidate path and determine the extent they are different from their preceding states.

Algorithm 4 first gets the list of states visited by p and p' as L and L', respectively (lines 2 and 3). It then computes a coarse-grained score (d_1) about how different L and L' are in terms of their edit distance (line 4). Here, we treat L and L' as two strings and the contained state identifiers (hash values) as characters and compute their Levenshtein distance [28], i.e., the number of required edits, inserts, or deletions that converts L to L'. Next, it traverses each state $v'_i \in L'$ sequentially. If v'_i is not a base state (i.e., $v'_i \notin L$), then we compute a fine-grained score to determine how different v'_i and v'_{i-1} (its previous state) are in terms of the edit distance between their XML representations, i.e., XML tree edit distance (line 8). This helps us estimate if the change from v'_{i-1} to v'_i is significant or merely incidental (such as a checkbox is selected). Finally, d_1 and the averaged fine-grained score (d_2) are normalized into $[0, 1]$ and a weighted sum of them is returned (line 13, with $\alpha = 0.5$ in our implementation).

5 EVALUATION

We investigated the following research questions in our experimental evaluation of ROUTE:

RQ1. What is the quality of tests generated by ROUTE? Are they for the same feature and useful?
RQ2. What are the main reasons for ROUTE failing to generate useful feature-based tests?
RQ3. Is the fault detection effectiveness of the augmented test suites improved? What types of faults can ROUTE identify?
RQ4. How does the size of the augmented test suites affect their fault detection effectiveness?

5.1 Experimental Setup

Subject tests. We collected a set of feature-based UI tests from prior work in mobile app testing [12, 42, 43]. We avoided including multiple apps from the same category (e.g., two or more shopping list apps) to ensure our dataset is diverse.[2] In the dataset from Reference [43], most of the subject apps came with only two or three tests. We hence expanded the original test suites to have at least six test cases for each app. Moreover, we added appropriate assertions to the tests from Reference [42] that had no assertion. The tests with oracles unsupported by our current implementation of ROUTE, such as assertions for Intent (the message object used in Android), were also excluded. Finally, we examined the subject tests to ensure each of them is written for exactly one feature. Table 1 shows the 73 subject tests in our study, including the test names, app names, and the contained GUI events and assertions. With the exception of Writeily Pro and Money Tracker, all of the other subject apps are popular (i.e., 100K+ to 50M+ installs on Google Play).

Implementation. We implemented ROUTE in Python for UI tests written using Appium [2], an open-source and cross-platform testing framework. Existing usage-based tests for the subject apps are written with Appiums' Python client. The generated tests are stored in JSON format and executed by our test runner. When collecting actionable widgets from current screen (line 4 and line 27 in Algorithm 2), we considered all leaf nodes in the XML representation if the *clickable* attribute is true. Moreover, the current implementation only supports action *click* in the App Exploration component, i.e., line 7 and line 29 in Algorithm 2. We do not support system events such as pressing back or home button. Furthermore, ROUTE does not generate text input other than the values used in the original tests.

In terms of the lookahead step in the App Exploration component, i.e., k in Algorithm 2, we conducted a pilot study to check the quality of the generated tests with different k and determined $k = 3$ to achieve a good balance between the execution time (the larger the value of k, the larger the search space) and the ability to find tests that cover the same feature as in the original test but in a different way. Regarding the number of generated tests in the Test Generation component, i.e.,

[2]The package, version, and category of subject apps can be found on the project website [3].

Table 1. Subject Tests and Experimental Results

App	Test	#Evt	#Asst	%Mutant Killed		App	Test	#Evt	#Asst	%Mutant Killed	
				Orig	Aug'd					Orig	Aug'd
Astro	TestAddFavorite	5	1	2	2	School Planner	TestAddGrade	8	1	8	14
	TestAppManager	2	1	5	16		TestAddSubject	8	1	0	14
	TestCreateFolder	5	1	2	2		TestAgenda	3	1	0	6
	TestListView	3	1	7	11		TestCalendar	2	1	0	6
	TestOpenDirectory	3	1	2	2		TestManageTimeTable	5	1	6	11
	TestSearch	6	1	9	9		TestSettings	3	1	0	6
Easy Bills Reminder	TestAddBill	7	1	0	19		TestViewProgression	3	1	0	8
	TestAddCategory	8	1	12	16		TestViewTimeTable	4	1	6	8
	TestDeleteBill	6	1	0	6	Shopping List	TestAddNewItem	4	1	0	0
	TestEditBill	9	1	0	3		TestAddNewList	5	1	0	10
	TestEditCategory	7	1	12	16		TestCheckAll	3	1	0	13
	TestMarkPaid	6	1	6	9		TestDeleteChecked	6	1	3	15
	TestSearch	3	1	0	9		TestDeleteList	3	1	3	13
Fuelio	TestAddCostLog	7	1	7	15		TestRenameList	5	1	3	10
	TestAddFuelLog	8	1	10	37		TestSearch	4	1	0	13
	TestAddReminder	7	1	7	28	To Do List	TestChangeCategory	8	2	16	16
	TestCalcTripCost	9	1	0	0		TestCheckEmptyList	3	1	11	11
	TestGasStations	5	2	7	7		TestCreateList	5	1	3	32
	TestViewTripLog	3	1	0	22		TestCreateTask	4	1	16	18
	TestViewVehicle	3	1	5	5		TestDeleteTask	6	2	16	16
Money Tracker	TestAccount	4	1	3	3		TestFinishTask	5	2	16	16
	TestAddExpense	7	2	12	12		TestSearch	3	1	0	3
	TestAddIncome	7	2	12	12	WordPress	TestAddDraft	10	1	7	25
	TestDeleteAllData	3	1	12	12		TestBlogPost	9	1	4	21
	TestEditExpense	5	1	12	12		TestMenuBlog	2	1	4	21
	TestEditIncome	5	1	12	12		TestMenuMedia	2	1	0	18
	TestExchangeRate	9	1	9	9		TestMenuPage	2	1	0	18
	TestImport	6	1	6	24		TestPageSearch	4	1	0	25
OwnCloud	TestCreateLink	10	3	11	13		TestPostNavigation	9	4	4	21
	TestDelete	6	2	3	13		TestViewSite	2	1	0	21
	TestFileDetail	9	2	3	8	Writeily Pro	TestCreateFolder	5	1	0	17
	TestRename	7	1	8	18		TestCreateNote	6	1	0	39
	TestSearch	3	1	0	11		TestDeleteNote	4	1	0	3
	TestSearchDetail	4	1	0	5		TestEditNote	4	1	0	25
	TestUpload	5	1	0	0		TestMoveNote	6	1	3	17
	TestViewStorage	2	1	0	11		TestRenameNote	5	1	3	6
							TestSearch	3	1	3	28
Total								377	86	17	38

#Evt: number of events. #Asst: number of assertions. %Mutant Killed: percentage of killed mutants. Orig: original test suite. Aug'd: augmented test suite.

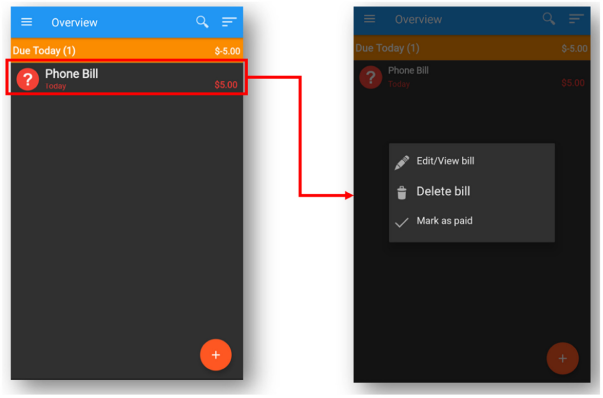
n in Algorithm 3, we configured $n = 3$, because in our experiments, we found that a value higher than 3 results in diminishing returns in terms of the fault detection effectiveness of generated tests and a value substantially lower than 3 sacrifices the ability to detect many defects. We evaluate and discuss how variable n influences the fault detection effectiveness of the augmented test suites as part of RQ4 in Section 5.5.

Finally, in our experiments, we used VirtualBox VM [39] with Android-x86 7.1 OS [1] installed. The experiments were conducted on a Windows laptop with 2.8 GHz Intel Core i7 CPU and 32 GB RAM. Our experimental data is publicly available [3].

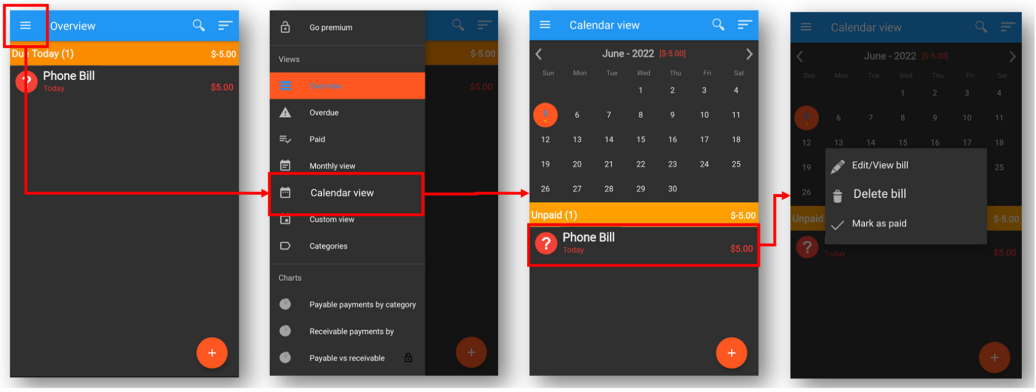
5.2 RQ1: Quality of the Generated Tests

We executed Route to generate 219 new tests (with $n = 3$) from the 73 original tests in Table 1. During the execution, App Exploration was the most time-consuming phase. On average, it took five hours to finish the exploration for an original test (with a median of 4.4 hours and a standard deviation of 3.8 hours). This is as expected, because the GUI states of the AUT (i.e., snapshots of VM) were repeatedly restored and restarted in this phase. Nevertheless, this phase can be accelerated several times by performing the exploration in parallel with multiple emulators.

To investigate whether Route is able to generate quality tests that verify the same features as the original tests, we manually examined the behavior of the 219 generated tests and categorized them into four groups.



(a) Original: click the bill from Overview to delete it



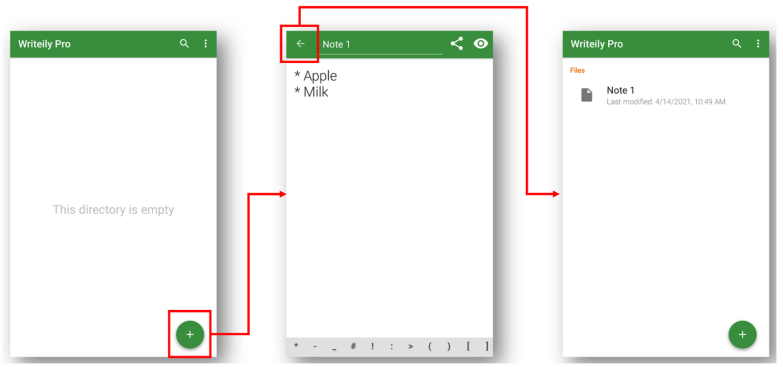
(b) Generated: click the bill from Calendar View to delete it

Fig. 5. Deleting a bill in Easy Bills Reminder with different controllers.

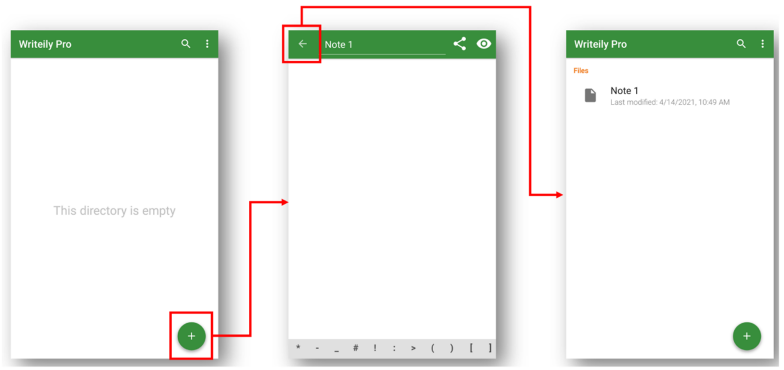
5.2.1 Different Controllers. In this category, the generated test performs the same task through different GUI controllers. Examples include Figures 1(b) and 1(c). Instead of the pop-up menu in the original test, these two tests reach the timetable management screen by the selection dialog and menu options in Settings, respectively. Another example is illustrated in Figure 5. In the original test, a bill is selected from the Overview screen and then deleted. Alternatively, the deletion can be achieved through the Calendar View screen, as performed in the generated test.

5.2.2 Different Input Data. In this category, the generated test performs the same task through the same controllers, but with different input data. For example, Figure 6 illustrates that the task of creating a note is tested the same way in both the original and generated tests, i.e., using the same controllers. However, the created notes are different in terms of their content (i.e., input data). Figure 7 provides another example in this category. An original test depicted in Figure 7(a) views the details of a photo, but in the generated test the photo is deleted and the details of another photo are displayed (Figure 7(b)). In other words, the task of viewing photo details is performed the same way, i.e., via the same menu option, in these two tests, but the photos (input data) are different.

5.2.3 Different Control Flow. The generated test performs the same task through the same controllers and with identical input data. However, it adds additional steps to the original control flow.



(a) Original: create a note with title and content



(b) Generated: create a note with title only

Fig. 6. Creating a note in Writeily Pro with different input data.

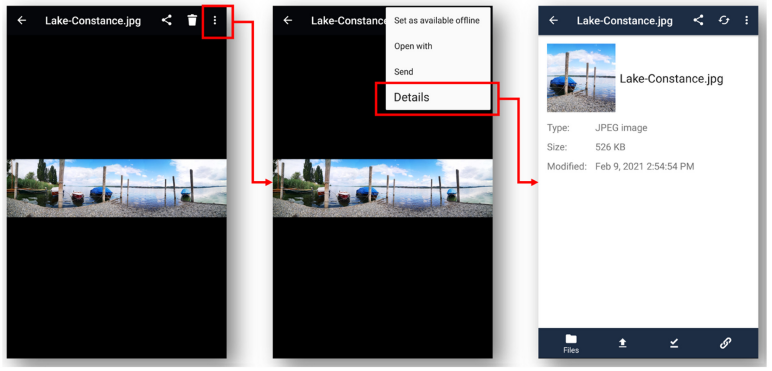
For instance, Figure 8(a) shows that the generated test switches from table view to list view before managing timetables. Similarly, Figure 8(b) illustrates that an additional circuit is added to the original control flow by opening and closing a dialog.

5.2.4 Deviated. Here, the original assertions accidentally pass in the generated test, but the desired functionality is not actually performed. For example, Figure 9(a) illustrates that an original test for the *page search* feature in WordPress first performs a search with a keyword and then checks if a specific post title is displayed. However, as shown in Figure 9(b), one of the generated tests directly navigates to the post list screen and performs the existence check of the post title. Such a test is not considered to be performing the same task as the original test.

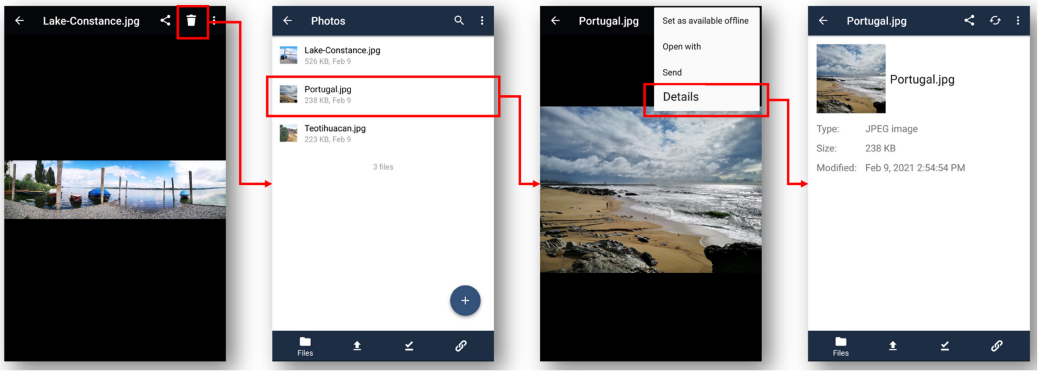
The number of generated tests for each category is listed in Table 2. 92% (201/219) of the tests generated by Route fall in the first three categories; these are high-quality tests that can be used to augment an existing test suite. Moreover, for 96% (70/73) of the original tests, Route was able generate at least one feature-based test. In the next research question, we further investigate why the deviated tests were generated and how the effectiveness of Route can be improved.

5.3 RQ2: Analysis of Deviated Tests

In the deviated tests generated by Route, the oracles in the original tests were satisfied without performing the desired functionality. We inspected all of the deviated tests to understand the



(a) Original: view the details of a photo via a menu option



(b) Generated: delete the photo and then view the details of another photo via the same menu option

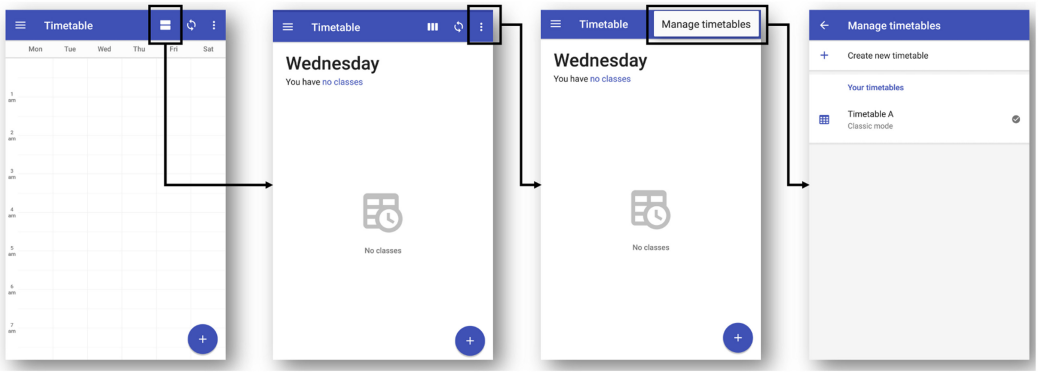
Fig. 7. Viewing photo details in OwnCloud with different input data.

Table 2. Categories of Tests Generated by ROUTE

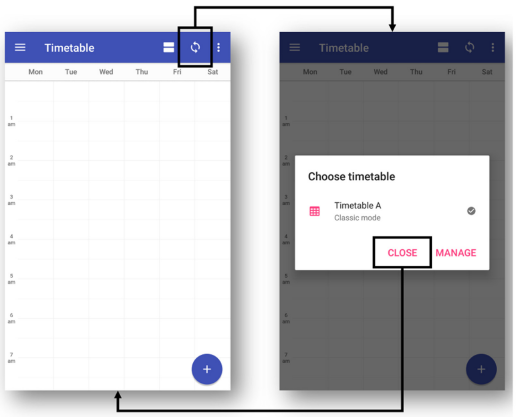
Category	#Generated Tests
Different Controllers	73 (33%)
Different Input Data	21 (10%)
Different Control Flow	107 (49%)
Deviated	18 (8%)
Total	219 (100%)

reasons behind their generation, as summarized in Table 3. As shown in Table 3, most of the deviated tests resulted from imprecise oracles. A factor contributing to imprecise oracles is duplicate widget ID. For example, in *TestDelete* for OwnCloud, an existence check of the TextView with ID *empty_list_view* is performed after deleting a file. However, this widget ID is used in multiple Activities (FileDisplayActivity and UploadListActivity), which resulted in a test that does not perform deletion but with a passed oracle. Using a unique widget ID may prevent such a situation.

Another example of imprecise oracles is *TestPageSearch* for WordPress, shown in Figure 9. In this test, an oracle verifies a post titled "sour candy" to confirm the search feature works. Nevertheless, the oracle can be satisfied by directly navigating to the post list screen without a search. The same case happened when ROUTE tried to augment *TestSearch* for Shopping List and Writeily Pro. To



(a) Switch from table view to list view before managing timetables



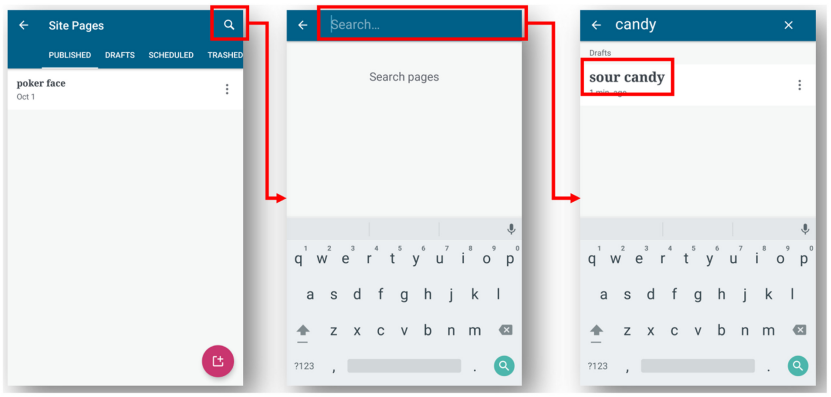
(b) Open and close a dialog before managing timetables

Fig. 8. Timetable management in School Planner with different control flows.

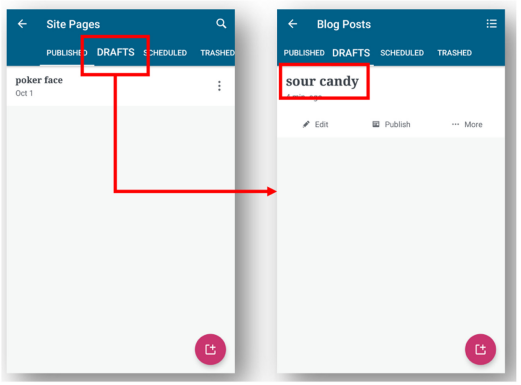
Table 3. List of Deviated Tests Generated by Route

Test	App	#Generated Deviated Test	Reason
TestAddBill	Easy Bills Reminder	3	Imprecise oracle
TestSearch	Easy Bills Reminder	2	Imprecise oracle
TestDelete	OwnCloud	2	Imprecise oracle
TestSearch	Shopping List	2	Imprecise oracle
TestPageSearch	WordPress	1	Imprecise oracle
TestDeleteNote	Writeily Pro	2	Replaced action
TestMoveNote	Writeily Pro	3	Imprecise oracle
TestSearch	Writeily Pro	3	Imprecise oracle
Total		18	

avoid this situation, test engineers may provide oracles that more precisely describe the expected program state of the AUT. For example, in addition to specific text about search results such as post title, the test can further check the screen title or Activity name to ensure that the correct screen is displayed.



(a) Original: search for a post and verify the search result



(b) Generated: the specific post can be accessed via post list

Fig. 9. Example deviated test in WordPress.

Another reason for generation of deviated tests is *replaced action*. For instance, in *TestDeleteNote* for Writeily Pro, the test first deletes a note and then checks if the TextView with ID *empty_hint* (hint text for empty folder) appears. However, the existence check can also be satisfied by moving the note elsewhere. To address this situation, we can allow the user of ROUTE to annotate core test steps (e.g., clicking the delete button) in the original tests. Such annotated events are then guaranteed to be included in Test Generation to ensure the generated tests are feature-based.

Although deviated tests are not as useful as other types of generated tests in augmentation of a test suite, they can be insightful for developers to improve the quality of assertions in original tests, e.g., the fact that an assertion passes on multiple screens can potentially indicate a problem with the original test.

5.4 RQ3: Fault Detection Effectiveness

To investigate whether the generated tests are advantageous in terms of **fault detection effectiveness (FDE)**, we created mutants for the apps under test with MutApk [17], an open-source mutation testing tool for APK files. It supports 35 mutation operators designed for Android apps [30] and performs the mutation on intermediate representations of the code. We created the mutants with MutApk's default strategy, in which both the mutation operators and locations of mutated code were picked randomly. The number of mutants created for each app is shown in Table 4.

Table 4. Number of Mutants Generated for the Subject Apps

App	#Mutants	$\|T_o\|$	$\|F_o\|$	$\|T_{n=3}\|$	$\|F_{n=3}\|$
Astro	44	6	8 (18%)	24	14 (32%)
Easy Bills Reminder	32	7	6 (19%)	28	14 (44%)
Fuelio	41	7	13 (32%)	28	21 (51%)
Money Tracker	33	8	9 (27%)	32	15 (45%)
OwnCloud	38	8	6 (16%)	32	16 (42%)
School Planner	36	8	5 (14%)	32	11 (31%)
Shopping List	39	7	3 (8%)	28	9 (23%)
To Do List	38	7	6 (16%)	28	14 (37%)
WordPress	28	8	2 (7%)	32	8 (29%)
Writeily Pro	36	7	3 (8%)	28	16 (44%)
Total	365	73	61 (17%)	292	138 (38%)

T_o: original test suites. $T_{n=3}$: augmented test suites. F_o: mutants killed by T_o. $F_{n=3}$: mutants killed by $T_{n=3}$.

Table 5. Types of Generated Mutants

Mutation Operator	Description (from Reference [30])	#Mutant Generated	# Mutant Killed by T_o	# Mutant Killed by $T_{n=3}$
ActivityNotDefined	Delete an activity <android:name="Activity"/>entry in the Manifest file	54	14	21
ClosingNullCursor	Assign a cursor to null before it is closed	8	1	4
DifferentActivityIntentDefinition	Replace the Activity.class argument in an Intent instantiation	8	1	7
FindViewByIdReturnsNull	Assign a variable (returned by Activity.findViewById) to null	17	7	13
InvalidActivityPATH	Randomly insert typos in the path of an activity defined in the Manifest file	17	7	14
InvalidFilePath	Randomly mutate paths to files	2	1	2
InvalidIDFindView	Replace the id argument in an Activitity.findViewById call	28	19	25
InvalidKeyIntentPutExtra	Randomly generate a different key in an Intent.putExtra(key, value)	8	0	2
InvalidLabel	Replace the attribute "android:label" in the Manifest file with a random string	16	0	0
InvalidSQLQuery	Randomly mutate a SQL query	11	0	6
InvalidViewFocus	Randomly focus a GUI component	23	0	0
LengthyGUICreation	Insert a long delay (i.e., Thread.sleep(..)) in the GUI creation thread	11	1	5
LengthyGUIListener	Insert a long delay (i.e., Thread.sleep(..)) in the GUI listener thread	5	0	1
MissingPermissionManifest	Select and remove an <uses-permission />entry in the Manifest file	14	1	1
NullInputStream	Assign an input stream (e.g., reader) to null before it is closed	11	1	1
NullIntent	Replace an Intent instantiation with null	22	3	8
NullMethodCallArgument	Randomly set null to a method call argument	18	2	8
NullValueIntentPutExtra	Replace the value argument in an Intent.putExtra(key, value) call with new Parcelable	6	0	3
ViewComponentNotVisible	Set visible attribute (from a View) to false	11	0	4
WrongMainActivity	Randomly replace the main activity definition with a different activity	1	0	1
WrongStringResource	Select a <string />entry in /res/values/strings.xml file and mutate the string value	74	3	12
Total		365	61	138

T_o: original test suites. $T_{n=3}$: augmented test suites.

The percentages of mutants killed by the original test suite and the augmented test suite are shown in Table 1. Note that we generated three more tests for each original test, so the size of augmented test sets is four (one original plus three generated). Table 1 shows that the fault detection effectiveness of the augmented test sets improved on 73% (53/73) of the subject tests and by up to 39% (in the case of *TestCreateNote* for Writeily Pro). The mean, median, and standard deviation of the FDE for the original test suites were 4.5%, 3%, and 4.9%, respectively. However, the mean, median, and standard deviation of the FDE for the augmented test suites were 13.3%, 12%, and 8.4%, respectively. In total, the augmented tests were able to kill 38% (138/365) of the mutants, while the original tests were only able to kill 17% (61/365) of them. Another perspective from each app regarding the improved fault detection effectiveness is reported in Table 4. Both perspectives indicate that the generated tests are not redundant. They covered additional parts of the AUT that the original tests missed. In other words, Route is capable of augmenting test suites to detect latent faults.

To further understand the types of faults that Route can help identify, we have listed the mutation operators used in our experiment in Table 5. Out of 21 mutation types created with the help of MutApk [17], the original test suites were able to cover 13 of them. However, the test suites

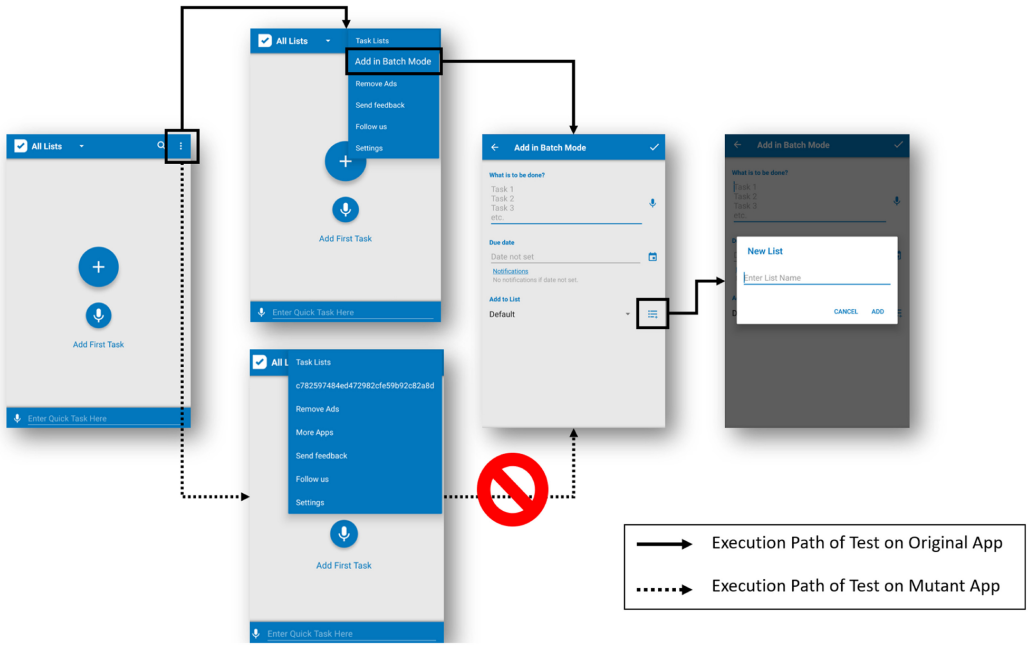


Fig. 10. Mutant of the To Do List app killed by the generated test to create a new list.

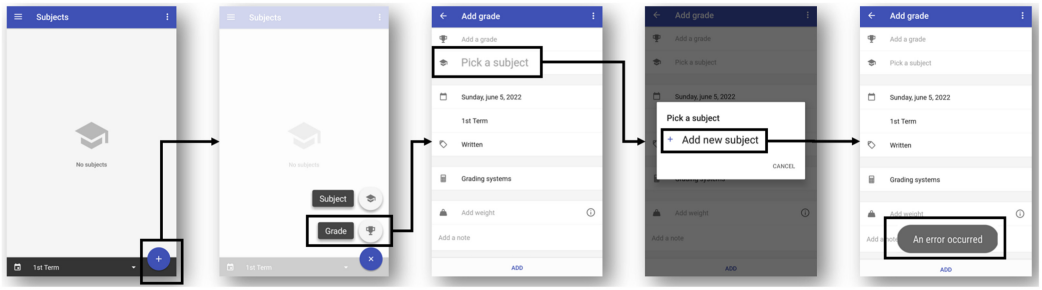


Fig. 11. Mutant of the School Planner app killed by the generated test to add a subject.

augmented by ROUTE outperformed the original suites in terms of both the types of mutants covered and the number of mutants killed. The augmented suites exclusively covered six more mutant types: *InvalidKeyIntentPutExtra*, *InvalidSQLQuery*, *LengthyGUIListener*, *NullValueIntentPutExtra*, *ViewComponentNotVisible*, and *WrongMainActivity*. Furthermore, the number of mutants killed by the augmented suites was significantly (over three times) more than the original suites for the following mutant types: *ClosingNullCursor*, *DifferentActivityIntentDefinition*, *LengthyGUICreation*, *NullMethodCallArgument*, and *WrongStringResource*.

Note that the behaviors manifested by the mutants were not necessarily app crashes. For instance, Figure 10 illustrates an alternative way found by ROUTE to create a list in the To Do List app, which is through adding tasks in batch. Here, the mutant created with operator *WrongStringResource* was exclusively killed by a test generated by ROUTE. The mutation operator modified a menu option that was clicked in one of the tests generated by ROUTE, resulting in the test to fail when executed on the mutant app. Figure 11 depicts another example of a non-crashing failure due to a

Table 6. Growth of Test Suite Size, Number of Killed Mutants, and Number of Types of Covered Mutation Operators

	T_o	$T_{n=1}$	$T_{n=2}$	$T_{n=3}$
Test Suite Size	73	146	219	292
#Mutant Killed	61	112	126	138
#Types Covered MutantOp	13	17	19	19

T_o: original test suite. $T_{n=x}$: augmented test suite by setting $n = x$.

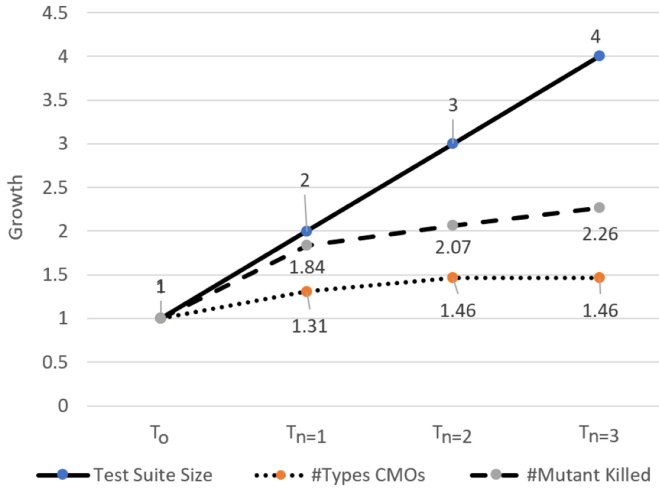


Fig. 12. Normalized growth of test suite size, number of killed mutants, and number of types of covered mutation operators. T_o: original test suite. $T_{n=x}$: augmented test suite by setting $n = x$.

mutant created with the *DifferentActivityIntentDefinition* operator. In this case, Route generated a test to add a subject in the School Planner app through adding a Grade, which resulted in exclusively killing the mutant. These examples demonstrate that Route was able to generate additional tests that exposed non-crashing failures.

5.5 RQ4: Size of the Augmented Test Suites

As mentioned previously, the number of tests generated by Route can be configured by setting the variable n. While expanding the test suites may improve their **fault detection effectiveness (FDE)**, it introduces additional costs in terms of test execution and maintenance. We analyzed how the size of augmented test suites influence the FDE in our experiments and listed the results in Table 6. The results indicate that when we grew the test suites linearly, the improvement was sub-linear in terms of both the FDE and the types of **covered mutation operators (CMOs)**. For example, when we expanded the test suites by 2 fold (from 73 tests to 146 tests, by setting $n = 1$), the FDE improved by a factor of 1.84 (112/61). Moreover, in the case that the size of test suites was increased by 4 fold (by setting $n = 3$), the FDE improved by a factor of 2.26 (138/61). Finally, increasing n from 2 to 3 did not improve the number of covered mutant types. We depict the normalized growth of FDE and CMOs along with the suite size in Figure 12. From this experiment, we conclude configuring Route with a value for n higher than 3 is likely to result in diminishing returns. Developers may choose to set n in between 1 and 3, depending on their desire to balance between test suite size and FDE.

5.6 Threats to Validity

The major external threat to validity of our results is the generalization to other subject test cases and apps. To mitigate this threat, we constructed our dataset by including different categories of apps. Another external threat to validity is using mutants in place of real faults in the experiments. Nevertheless, prior empirical study [26, 40] indicates a statistically significant correlation between mutant detection and real fault detection. Moreover, we reported the mutation operators covered by both the original and augmented test suites to provide more insights regarding what types of real bugs ROUTE may be good at revealing. The main internal threat to validity of the proposed approach is the possible mistakes involved in our implementation and experiments. When ROUTE identifies GUI states, dynamic content such as timestamps or ads may introduce inaccuracy, i.e., the XML layout is slightly changed but should be considered as the same screen. We leveraged virtualization to mitigate this issue and did not observe any instance of inaccuracies caused by dynamic content in our experimental results. That said, to prevent such potential inaccuracy caused by dynamic content, one can adopt more sophisticated state abstraction models such as the ones used in prior work [10, 21, 47] to determine which screens are equivalent. Moreover, we manually inspected all of our results to increase our confidence in their correctness. The experimental data is also publicly available for external inspection.

6 RELATED WORK

6.1 Test Augmentation

Test augmentation techniques [6, 11, 13, 18, 23, 27, 35, 41, 44, 46, 49, 50] create new tests from existing ones to achieve a given engineering goal, such as improving coverage according to a criterion. Pezze et al. [41] proposed to leverage existing unit tests to construct more complex tests that focus on class interactions to reveal more faults. Yoo and Harman [49] introduced a search-based technique that can generate additional test data from existing test data to improve the input space coverage. Harder et al. [23] presented a test augmentation technique based on *operational abstraction*, which is a formal specification of program's runtime behavior that can be dynamically mined. A test suite can be augmented by adding test cases until the operational abstraction stops changing. Tillmann and Schulte [46] proposed to use symbolic execution and constraint solving to help increase code coverage by finding relevant variations of existing unit tests. Similarly, Bloem et al. [13] used symbolic execution and model checking techniques to alter path conditions of existing tests and generate new tests that enter uncovered features of the program. Starting from concrete unit tests, Fraser and Zeller [18] presented an approach to generate parameterized unit tests containing symbolic pre- and post-conditions to achieve higher code coverage. To improve the mutation score of an existing test suite, Baudry et al. [11] introduced a genetic algorithm to guide the search for test cases that kill more mutants. Focusing on the context of regression testing, the work by Santelices et al. [44] adopted dependency analysis and symbolic execution to synthesize new tests with respect to the code changes not covered by existing tests. Another work considering test-suite augmentation for code changes by Kim et al. [27] leverages different test generation algorithms dynamically, since different algorithms have different strengths. Finally, Zhang and Elbaum [50] developed a solution to amplify a test suite for finding bugs in exception handling code.

ROUTE is different from the prior work because it is for GUI tests, while all of the above augmentation techniques are for unit tests. Furthermore, ROUTE aims to generate tests that verify the same functionality as the original tests, which is not the focus of prior work. Finally, unlike ROUTE, none of the above-mentioned techniques target Android apps.

Our work is more related to Thor, proposed by Adamsen et al. [6]. Thor takes existing UI tests for Android apps and injects neutral event sequences to see if the original assertions still pass.

Neutral event sequences are a series of operations that are not expected to affect the outcome of the injected test cases, such as rotating the phone or turning the screen off and on. Unlike Route, Thor is not able to find alternative tests for verifying the same functionality, because its goal is to simply expose the AUT to adverse conditions. In other words, Thor cannot generate the types of tests discussed in Section 5.2 in which the same functionality of the AUT is examined with different controllers, input data, or control flows.

Another work related to ours is Testilizer for *web applications*, proposed by Fard et al. [35]. Testilizer first infers a state flow graph from an existing web test, dynamically explores the graph, and then generates new tests from the updated graph. The goal of Testilizer is to explore new states and apply new and generic assertions learned from existing ones. In other words, the augmentation by Testilizer is not feature-based. It is also not applicable to Android apps.

6.2 Automated Test Generation for Android Apps

App crawling based on GUI hierarchy information. Many automated test generation techniques [5, 7, 8, 14, 15, 22, 32–34, 36, 38, 45, 48] leverage GUI hierarchy information of the AUT and apply different exploration strategies to create and execute GUI events to improve code coverage. For example, given an Android app, CrashScope [38] explores the app by dispatching randomly generated events, which can be system events or GUI events extracted from the GUI hierarchy. While we also leverage GUI hierarchy information for test generation, what distinguishes Route from prior work, including CrashScope, is that our exploration strategy is directed by existing test suites to achieve feature-based augmentation. Note that Route is not another record-and-replay tool, such as Reference [24], because our goal is not to faithfully replay existing tests, but to diverge from them in a manner that allows us to meaningfully examine the same features.

Systematic testing. To perform systematic testing of Android apps, prior work [8, 9, 19, 25, 37] used techniques such as targeted event sequence generation and symbolic execution. Tanzirul and Neamtiu proposed A3E [9], an app crawler that aims to increase app coverage (Activity and method coverage) by prioritizing events that lead to undiscovered states in the app. Mirzaei et al. [37] and Anand et al. [8] introduced concolic testing in Android to automatically explore obscured parts of an app by finding proper events and contextual settings. Jensen et al. [25] introduced a two-phase technique (generating event-handler summaries and concolic testing) that produce backward event sequences from a target line of code to the entry point of the app. Xiang et al. [19] tried to address the environmental challenges of symbolic execution in the presence of various Android SDKs and libraries by deducing a representation of libraries on-the-fly.

Similarly, Route tries to find an event sequence to reach a specific part of the app. However, unlike the above tools, our goal is not to increase the code coverage by reaching *unexplored* parts of an app. In contrast, we try to find a new path covering the *already visited* states in existing test cases through different sequence of events. In addition, previous tools primarily focus on achieving certain objectives at the source-code level (i.e., to reach specific lines of code), while Route focuses on covering the same behavior and feature as the original test (i.e., to find a path that covers the terminal and oracle GUI states). Finally, it is worth noting that symbolic execution can be used as a complementary approach together with Route to find proper input values to increase the exploration performance.

7 CONCLUSION

There are often several ways of invoking the core features of an app. Due to the manual effort of writing tests, developers tend to consider only the typical way of invoking a feature when creating the tests. However, the alternative ways of invoking a feature are as likely to be faulty, which would go undetected without proper tests. This article presented Route, an automated

tool for feature-based UI test augmentation for Android apps. ROUTE creates high-quality tests, consisting of both inputs and assertions, to verify the features tested by existing tests in alternative ways. ROUTE relies on several novel heuristics to guide the search for new tests and leverages virtualization technology to save the visited UI states, such that the states can be fully restored later for exploration.

Experimental results using real-world subjects have demonstrated the effectiveness of ROUTE, as it successfully generated alternative tests for 96% of the existing test cases in our experiments. Moreover, the fault detection effectiveness of augmented test suites in our experiments showed substantial improvements of up to 39% over the original test suites.

In our future work, we aim to investigate the applicability of techniques described here in other computing domains (e.g., web applications) and other testing paradigms (e.g., unit tests). Moreover, we plan to conduct sensitivity analysis to discuss the tradeoff between the performance and fault detection ability of ROUTE with larger lookahead steps and the support of system events in the app exploration phase. Finally, we plan to develop an adapter that translates the JSON-formatted outputs to executable tests in a programming language such as Java and conduct user studies with practitioners to validate the utility of tests generated using ROUTE in practice.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers of this article for their detailed feedback, which helped us improve the work.

REFERENCES

[1] Chih-Wei Huang. 2022. *Android-x86 - Porting Android to x86*. Retrieved from https://www.android-x86.org/.

[2] OpenJS Foundation. 2022. *Appium*. Retrieved from https://github.com/appium/appium.

[3] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. 2022. *Route Project Website*. Retrieved from https://sites.google.com/view/route.

[4] Andrea Dal Cin. 2022. *School Planner*. Retrieved from https://play.google.com/store/apps/details?id=daldev.android.gradehelper.

[5] Google LLC. 2022. *UI Application Exerciser Monkey*. Retrieved from https://developer.android.com/studio/test/monkey.

[6] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of Android test suites in adverse conditions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'15)*. Association for Computing Machinery, New York, NY, 83–93. DOI: https://doi.org/10.1145/2771783.2771786

[7] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Softw.* 32, 5 (Sept. 2015), 53–59. DOI: https://doi.org/10.1109/MS.2014.55

[8] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE'12)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2393596.2393666

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

[10] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. Association for Computing Machinery, New York, NY, 238–249. DOI: https://doi.org/10.1145/2970276.2970313

[11] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. 2005. From genetic to bacteriological algorithms for mutation-based testing. *Softw. Test. Verif. Reliab.* 15, 2 (2005), 73–96.

[12] F. Behrang and A. Orso. 2019. Test migration between mobile apps with similar functionality. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. 54–65.

[13] Roderick Bloem, Robert Koenighofer, Franz Röck, and Michael Tautschnig. 2014. Automating test-suite augmentation. In *Proceedings of the 14th International Conference on Quality Software*. 67–72. DOI: https://doi.org/10.1109/QSIC.2014.40

[14] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming*

Systems Languages & Applications (OOPSLA'13). ACM, New York, NY, 623–640. DOI: https://doi.org/10.1145/2509136.2509552

[15] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of Android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*. Association for Computing Machinery, New York, NY, 481–492. DOI: https://doi.org/10.1145/3377811.3380402

[16] Paul M. Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk*. Pearson Education.

[17] Camilo Escobar-Velásquez, Michael Osorio-Riaño, and Mario Linares-Vásquez. 2019. MutAPK: Source-codeless mutant generation for Android apps. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19)*. 1090–1093. DOI: https://doi.org/10.1109/ASE.2019.00109

[18] Gordon Fraser and Andreas Zeller. 2011. Generating parameterized unit tests. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'11)*. Association for Computing Machinery, New York, NY, 364–374. DOI: https://doi.org/10.1145/2001420.2001464

[19] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE'18)*. IEEE, 419–429.

[20] Google. 2021. *Create and Manage Virtual Devices*. Retrieved from https://developer.android.com/studio/run/managing-avds.

[21] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE'19)*. 269–280. DOI: https://doi.org/10.1109/ICSE.2019.00042

[22] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. ACM, New York, NY, 204–217. DOI: https://doi.org/10.1145/2594368.2594390

[23] Michael Harder, Jeff Mellen, and Michael D. Ernst. 2003. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, 60–71.

[24] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for Android. In *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 349–366.

[25] Casper S. Jensen, Mukul R. Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *Proceedings of the International Symposium on Software Testing and Analysis*. 67–77.

[26] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. Association for Computing Machinery, New York, NY, 654–665. DOI: https://doi.org/10.1145/2635868.2635929

[27] Yunho Kim, Zhihong Zu, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. 2014. Hybrid directed test suite augmentation: An interleaving framework. In *Proceedings of the IEEE 7th International Conference on Software Testing, Verification and Validation*. 263–272. DOI: https://doi.org/10.1109/ICST.2014.39

[28] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, Vol. 10. 707–710.

[29] J. Lin, N. Salehnamadi, and S. Malek. 2020. Test automation in open-source Android apps: A large-scale empirical study. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE'20)*.

[30] Mario Linares-Vásquez, Gabriele Bavota, Michele Tufano, Kevin Moran, Massimiliano Di Penta, Christopher Vendome, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2017. Enabling mutation testing for Android apps. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. Association for Computing Machinery, New York, NY, 233–244. DOI: https://doi.org/10.1145/3106237.3106275

[31] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. 2017. How do developers test Android applications? In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*. 613–622. DOI: https://doi.org/10.1109/ICSME.2017.47

[32] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for Android apps. In *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*. ACM, New York, NY, 224–234. DOI: https://doi.org/10.1145/2491411.2491450

[33] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented evolutionary testing of Android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, New York, NY, 599–609. DOI: https://doi.org/10.1145/2635868.2635896

[34] 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'16)*. ACM, New York, NY, 94–105. DOI: https://doi.org/10.1145/2931037.2931054

[35] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*. ACM, 67–78. DOI: https://doi.org/10.1145/2642937.2642991

[36] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek. 2015. SIG-Droid: Automated system input generation for Android applications. In *Proceedings of the IEEE 26th International Symposium on Software Reliability Engineering (ISSRE'15)*. 461–471. DOI: https://doi.org/10.1109/ISSRE.2015.7381839

[37] Nariman Mirzaei, Sam Malek, Corina S. Păsăreanu, Naeem Esfahani, and Riyadh Mahmood. 2012. Testing Android apps through symbolic execution. *ACM SIGSOFT Softw. Eng. Notes* 37, 6 (2012), 1–5.

[38] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017. CrashScope: A practical tool for automated testing of Android applications. In *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C'17)*. IEEE, 15–18.

[39] Oracle. 2021. *Oracle VM VirtualBox*. Retrieved from https://www.virtualbox.org/.

[40] Mike Papadakis, Donghwan Shin, Shin Yoo, and Doo-Hwan Bae. 2018. Are mutation scores correlated with real fault detection? A large scale empirical study on the relationship between mutants and real faults. In *Proceedings of the IEEE/ACM 40th International Conference on Software Engineering (ICSE'18)*. 537–548. DOI: https://doi.org/10.1145/3180155.3180183

[41] M. Pezze, K. Rubinov, and J. Wuttke. 2013. Generating effective integration test cases from unit ones. In *Proceedings of the IEEE 6th International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 11–20. DOI: https://doi.org/10.1109/ICST.2013.37

[42] Xue Qin, Hao Zhong, and Xiaoyin Wang. 2019. TestMig: Migrating GUI test cases from iOS to Android. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19)*. ACM, New York, NY, 284–295. DOI: https://doi.org/10.1145/3293882.3330575

[43] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy Branham, and Sam Malek. 2021. Latte: Use-case and assistive-service driven automated accessibility testing framework for Android. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*. 1–11.

[44] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2008. Test-suite augmentation for evolving software. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*. 218–227. DOI: https://doi.org/10.1109/ASE.2008.32

[45] 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 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. ACM, New York, NY, 245–256. DOI: https://doi.org/10.1145/3106237.3106298

[46] N. Tillmann and W. Schulte. 2006. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Softw.* 23, 4 (2006), 38–47. DOI: https://doi.org/10.1109/MS.2006.117

[47] Wenyu Wang, Wei Yang, Tianyin Xu, and Tao Xie. 2021. Vet: Identifying and avoiding UI exploration tarpits. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. Association for Computing Machinery, New York, NY, 83–94. DOI: https://doi.org/10.1145/3468264.3468554

[48] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *Fundamental Approaches to Software Engineering*. Springer Berlin, 250–265.

[49] S. Yoo and M. Harman. 2012. Test data regeneration: Generating new test data from existing test data. *Softw. Test. Verif. Reliab.* 22, 3 (May 2012), 171–201. DOI: https://doi.org/10.1002/stvr.435

[50] Pingyu Zhang and Sebastian Elbaum. 2012. Amplifying tests to validate exception handling code. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 595–605. DOI: https://doi.org/10.1109/ICSE.2012.6227157

Received 4 December 2021; revised 31 July 2022; accepted 12 October 2022