A Systematic Study on Factors Impacting GUI Traversal-Based Test Case Generation Techniques for Android Applications

Bo Jiang, *Member, IEEE*, Yaoyue Zhang, *Student Member, IEEE*, Wing Kwong Chan, *Member, IEEE*, and Zhenyu Zhang, *Member, IEEE*

***Abstract*—Many test case generation algorithms have been proposed to test Android apps through their graphical user interfaces. However, no systematic study on the impact of the core design elements in these algorithms on effectiveness and efficiency has been reported. This paper presents the *first* controlled experiment to examine three key design factors, each of which is popularly used in GUI traversal-based test case generation techniques. These three major factors are definition of GUI state equivalence, state search strategy, and waiting time strategy in between input events. The empirical results on 33 Android apps with real faults revealed interesting results. First, different choices of GUI state equivalence led to significant difference on failure detection rate and extent of code coverage. Second, searching the GUI state hierarchy randomly is as effective as searching it systematically. Last but not the least, the choices on when to fire the next input event to the app under test is immaterial so long as the length of the test session is practically long enough such as 1 h. We also found two new GUI state equivalence definitions that are statistically as effective as the existing best strategy for GUI state equivalence.**

***Index Terms*—Android app, controlled experiment, design factor, fuzzing, test case generation, testing.**

NOMENCLATURE

Acronyms

GUI	Graphical user interface.
DOM	Document object model.
BFS	Breadth first search.
DFS	Depth first search.
UI	User interface.
API	Application programing interface.
OS	Operating system.
ANOVA	ANalyses Of VAriances.

Manuscript received August 11, 2018; revised January 14, 2019; accepted March 21, 2019. Date of publication August 2, 2019; date of current version August 29, 2019. This work was supported in part by the National Natural Science Foundation of China of China under Grant 61772056, Grant 61690202, and Grant 61379045, in part by the Research Fund of the MIIT of China under Grant MJ-Y-2012-07, in part by the GRFs of Research Grants Council of Hong Kong under Grant 11201114, Grant 11200015, and Grant 11214116, and in part by an open project from the State Key Laboratory of Computer Science under Grant SYSKF1608. Associate Editor: T. Dohi. *(Corresponding author: Wing Kwong Chan.)*

B. Jiang and Y. Zhang are with the State Key Laboratory of Software Development Environment, School of Computer Science and Engineering, Beihang University, Beijing 100191, China (e-mail: jiangbo@buaa.edu.cn; zhangyaoyue@buaa.edu.cn).

W. K. Chan is with the Department of Computer Science, City University of Hong Kong, Hong Kong (e-mail: wkchan@cityu.edu.hk).

Z. Zhang is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100864, China (e-mail: zhangzy@ios.ac.cn).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2019.2928459

I. INTRODUCTION

ANDROID [37] is the most widely deployed mobile operating system (OS). In early 2017, the number of Android applications (apps) in Google Play is about 2.4 million and in late 2017, almost another 1 million Android apps have been added to Google Play [39]. Owing to the competitions among apps of the same kind, developers are increasingly placing higher priorities to assure their Android apps against bugs.

Program testing is the de facto approach to assure the correctness of Android apps. Many test case generation techniques have been reported in the literature [3], [5], [7], [13], [14], [20], [24]. These techniques can be broadly grouped into several classes. Relatively major classes, include fuzzers (e.g., IntentFuzzer [34] and monkey), GUI traversal-based techniques [1], [5], [13], [24], search-based techniques [25], [30], and concolic testing techniques [31]. In particular, the class for GUI traversal-based techniques (StateTraversal for short) has been studied heavily by many researchers [31].

The workflow of typical StateTraversal techniques runs as follows: In a test session, a StateTraversal technique T begins testing an Android app P from its entry *Activity*. To differentiate different GUI states, it defines both what a GUI state is and a comparison function to determine whether two given GUI states are equivalent. It selects one GUI widget w from the set of all operable and unexplored widgets of the current GUI state, and sends an input event to w. Such an event may or may not cause P to trigger a transition between GUI states. Next, T waits for a short period of time followed by determining Ps current GUI state. The abovementioned procedure repeats until the technique T has interacted with all the operable widgets of each encountered GUI state. After that, T repeats the abovementioned workflow until the time allocated to the test session is over.

In the abovementioned workflow, there are some common configurable parameters for StateTraversal techniques. First, a fundamental concept in such a technique is the definition of GUI state. Once the notion of GUI state is defined, the notion of GUI state equivalence can be further defined. However, do such

choices really make StateTraversal techniques show noticeable differences in testing effectiveness and efficiency?

Another key design factor for such a technique is to determine the ordering of widgets under the current GUI state for *T* to interact with *P*, and this ordering of widgets is defined by the employed search strategy to traverse the current GUI widget tree of *P*. A popular approach is to use a systematic strategy, such as BFS or DFS, to traverse such a tree. We ask: Is a systematic search strategy really more effective than a randomized strategy (Random)?

Another important design factor in designing a StateTraversal technique is the time to wait between two consecutive events. For instance, a rule of thumb is to send the next input event when the GUI state of *P* is "stable." We ask yet another question: Is there really a significant difference in terms of testing effectiveness if technique *T* just waits for a constant amount of time before sending the next input event?

These three design factors are important because they represent the core decisions in a StateTraversal technique [13]. In previous studies on GUI traversal-based testing [3], [5], [13], [22], researchers have used some of the treatments of the three design factors in their own testing frameworks. Their empirical results indicate that the choice of treatment for the three design factors may or may not have an impact of testing effectiveness. However, there is still no systematic study on these three factors within a unified testing framework.

In this paper, to the best of our knowledge, we present the *first* paper that reports a systematic controlled experiment to explore the impact of the abovementioned three key factors configurable for StateTraversal techniques. Our controlled experiment studied the impacts of different factor levels of three factors on 33 real-world Android apps on failure detection rate on real faults and the extent of code coverage.

The controlled experiment selected widely used factor levels for each such factor. For the factor on the criterion for GUI state equivalence, the controlled experiment studied same Activity ID, similar UI hierarchy of a GUI state, and similar GUI hierarchy and partial widget properties (in terms of Cosine similarity, Jaccard similarity, and Hamming distance) as the factor levels. For the factor on search strategy, the controlled experiment studied BFS, DFS, and Random strategies as the factor levels. Regarding the factor on the time period to wait before sending the next input event to the app under test, the controlled experiment studied wait-for-idle and wait-for-a-while (using typical time period used in previous paper) as the factor levels. We have implemented each combination of these three factor levels in our framework. For each such combination, we have executed each app for 1 h to collect the data, meaning that our data analysis were based on the datasets on testing 33 apps for 1980 h in total.

We have obtained interesting results from our controlled experiment. First, we found that the uses of Cosine similarity, Jaccard similarity, and Hamming similarity led to higher failure detection rates and higher code coverage at the 5% significance level than using the other factor levels for the factor on GUI state equivalence, where both Jaccard and Hamming similarities are our newly found GUI state equivalence definitions using the same granularity of GUI state information as Cosine similarity through the controlled experiment. Second, BFS, DFS, and RS were statistically comparable to one another at the 5% significance level in terms of failure detection rate and also in terms of code coverage when the testing time was sufficiently long. Third, the wait-for-while strategy was not statistically less effective than the wait-for-a-idle strategy at the 5% significance level for both effectiveness and code coverage when the period of the test session was set to 1 h. At a closer look, we found that different waiting strategies and search strategies indeed have different convergence speeds in terms of failure detection rate when the testing time is very limited. Finally, if a factor was constrained to be a particular factor level, we found many combinations of factor levels of the other two factors achieving the maximal failure detection rates and maximal code coverage under the above constraint.

Note that our experimental framework mainly focuses on click input events on GUI widgets, the study on other input factors, such as gestures, text inputs, environmental inputs, and thread scheduling, are beyond the scope of this paper.

The main contribution of this paper is threefold as given as follows.

1) This paper summarizes the *first* controlled experiment that systematically examined the influences of selected factors in the design of StateTraversal techniques on testing effectiveness when testing Android apps.
2) The results show that carefully choosing what constitute the definition of GUI state equivalence is important as this definition significantly affects the testing effectiveness of this class of techniques. Through the controlled experiment, we discovered two previously unknown GUI state equivalence definitions that were statistically as effective as the existing best strategy, i.e., Cosine similarity.
3) The results revealed that StateTraversal techniques can be configured to equip with many different combinations of factor levels of these three factors without significantly compromising the testing effectiveness.

This paper extends its conference version [16] in four aspects. First, it identified two new effective definitions of GUI state equivalence, which are as effective as the existing best strategy statistically. Second, it extends the conference version by studying the impact of the waiting time and search strategy on testing effectiveness when the testing time is very limited. Third, it substantially extends the controlled experiment to study these two aspects in detail. Finally, the scale of the controlled experiment is enlarged by 66.7% (from 36 combinations in [16] to 60 combinations in this paper), and thus, all the data analyses reported in this paper were not reported in its conference version.

The rest of this paper is organized as follows. Section II gives an overview of StateTraversal techniques. Section III summarizes the design factors and their levels studied in this paper. Section IV presents the controlled experiment and the data analysis. We review the related work in Section V and Section VI concludes this paper.

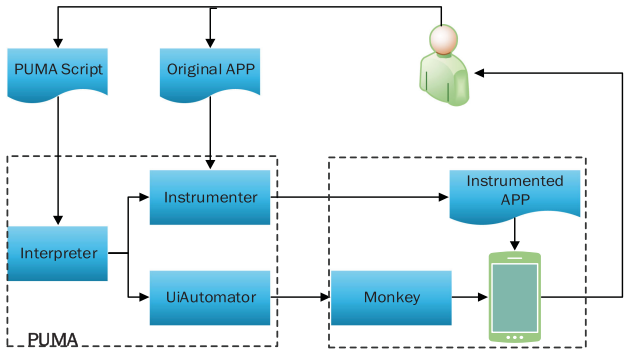

Fig. 1. Overall workflow of PUMA.

TABLE I
GENERIC GUI EXPLORATION-BASED TEST CASE GENERATION FRAMEWORK OF PUMA

```
1: foreach app under test
2:   start a new app under test
3:   S←empty stack
4:   push entry activity to S
5:   while S is not empty do
6:     pop an unfinished activity s_i from S
7:     examine activity s_i
8:     select next clickable UI widget from s_i
       // Factor 2: Search strategy
9:     perform the click
10:    wait for next acitivty s_j to load
       // Factor 3: Waiting time
11:    flag←s_j is equivalent to an explored activity
       // Factor 1: State equivalence
12:    if not flag then
13:      push s_j to S
14:      update finished clicks for s_i
15:    if all widgets in s_i are clicked then
16:      pop s_i from S
17:    if S is empty then
18:      stop testing the app
```

II. GUI Traversal-Based Test Case Generation

A. PUMA Framework Revisited

PUMA [13] is a framework to support testing of Android apps. It includes a component for dynamic analysis and a component for GUI traversal-based test case generation. Fig. 1 shows its overall workflow. PUMA instruments the app under test, and uses its enhanced monkey tool to execute the instrumented app. When the test session ends, PUMA generates log files for subsequent data analysis and dynamic analysis.

B. Algorithm

Table I presents the skeletal test case generation algorithm of PUMA. The underlined comments highlight the code configurable by client tools within the PUMA framework. In total, there are three factors highlighted and explored in this paper.

In the algorithm, the variable s represents a GUI state, each associating with a set of clickable UI elements. If not every clickable UI element of s received at least one input event, the state s is called *unfinished*, otherwise *finished*. The variable S represents the set of GUI states. S is set to be empty when the algorithm starts.

The algorithm starts an Android app P. It pushes the *entry Activity* of P into S. It pops an unfinished state from S, chooses a clickable UI widget w based on a search strategy, and sends an input event to w. After that, the algorithm waits for a while or waits until the next UI activity is loaded. It then checks whether there is any state kept in S equivalent to the current GUI state. If there is no such equivalent state in S, the algorithm pushes the current GUI state into S. If all the UI widgets on a GUI state in S have been explored, the algorithm removes that GUI state from S. If S is nonempty, the abovementioned process repeats.

Lines 8, 10, and 11 are marked with underlined comments, indicating the positions of the studied design factors, which we are going to present in Section III.

The algorithm only shows the key design considerations of the PUMA framework while some of the details are omitted for brevity. In particular, during GUI state exploration, if the application under test get stuck in a state or the testing reaches another application, the algorithm will send the *back* button pressed event or restart the application (i.e., stop application and re-execute actions to reach next unexplored state) to resume exploration.

III. Summary of Design Factors and Factor Levels

A. GUI State Equivalence

This design factor is to examine what should constitute an abstraction of a GUI state and what levels of equivalence between two GUI states to be used. We studied five factor levels in our controlled experiment. Three of them are the factor levels proposed in previous work while two others are factor levels newly proposed by this paper.

Note that when we discuss the factor levels for GUI state equivalence, we abbreviate the names of the factor levels for ease of presentation. For example, the factor level "*ActivityID*" is the abbreviated name for "GUI state equivalence definitions based on *ActivityID*." The full name for *UI Hierarchy*, *Cosine*, *Jaccard*, and *Hamming* can be interpreted similarly.

1) Factor Levels Adopted From Previous Work: In previous work, the ActivityID (see A3E [3]), the UI hierarchy (see SwiftHand [5]), and the Cosine similarity (see DECAF [22] and PUMA [13]) have been used to define GUI state equivalence. We also adopted these three factor levels in this paper.

Factor level *ActivityID*: If the ActivityIDs of two GUI states are the same, then the two states are deemed equivalent. ActivityID is the identifier of each activity in Android OS and this notion of GUI state equivalence is easy to implement. For instance, the ActivityID of an activity can be returned by *getCurrentActivityName*() in PUMA, and a string comparison on two ActivityIDs can determine the equivalence.

Factor level *UI Hierarchy*: A GUI state is represented by the hierarchical structure of its widget tree without considering

the detailed properties of GUI widgets. Each GUI widget in a UI Hierarchy is modeled by its widget type. For instance, a textbox widget located at the third level in the DOM tree of a UI Hierarchy is represented as "TextBox@3." Two GUI states are equivalent iff their corresponding widgets tree representations are identical.

Factor level *Cosine*: In DECAF [22], the tool models a GUI state by a feature vector. Such a vector is in the form of a map, and the keys are of type *String* and the values are of type *Integer*. Given a UI hierarchy, the tool extracts the class name of each visible widget, the level in the DOM tree that the widget located at, as well as the text of the widget, and then it concatenates all these information into a string to represent a key in the map. The value of the map is 1 or 0, where 1 means such a key exists and vice versa. For instance, a checkbox at level 3 can be modeled as a binary tuple ⟨checkbox@3@"Close", 1⟩ in the feature vector, which represents that the UI element is a check box with text "Close" at the level 3 of the DOM tree in the UI hierarchy. This binary tuple represents one element of the feature vector, and the binary tuples of all GUI elements in the current GUI hierarchy forms the feature vector of the corresponding GUI state.

With such feature vector representation of a GUI state, one can calculate the distance between two GUI states using *Cosine* similarity. Given two feature vector $V = \langle v_1, v_2, \ldots, v_n \rangle$ and $U = \langle u_1, u_2, \ldots, u_n \rangle$, their Cosine similarity is defined as follows:

$$\text{Cosine}\,(V,U) = \sum_{i=1}^{n} v_i u_i / \left(\sqrt{\sum_{i=1}^{n} v_i^2} * \sqrt{\sum_{i=1}^{n} u_i^2} \right).$$

The *Cosine* similarity ranges from -1 to 1, where -1 means the opposite while 1 means exactly the same. PUMA uses a default threshold of 0.95 for *Cosine* similarity to determine whether two GUI states are equivalent. In our controlled experiment, we also adopted the same default threshold of 0.95 for consistency.

We note that, if the used feature vectors are bit vectors (note that the values of the map in our experiment are either 0 or 1), the *Ochiai* similarity coefficient will be the same as the Cosine similarity coefficient in this scenario. So, a further study on *Ochiai* similarity coefficient as a new factor level is not included in our controlled experiment.

2) New Factor Levels Proposed in This Paper: In our preliminary empirical study [16], we concluded that using a finer notion of state equivalence statistically resulted in higher failure detection ability and code coverage rate. However, it is still unknown whether this conclusion is a general rule or not. Therefore, in this paper, we apply this rule to design two new GUI state equivalence definitions to further evaluate the generality of this rule. The Jaccard and Hamming distance definitions use the same granularity of GUI state information as Cosine, but with different distance calculation formulas. Therefore, adding Jaccard and Hamming in our study can help evaluate the validity and generality of our conclusion in a more convincing way.

Factor level *Jaccard*: Based on the feature vector representation of the GUI state stated in Section III-A1, we can calculate the distance between two GUI states with *Jaccard* similarity. Given two feature vector $V = \langle v_1, v_2, \ldots, v_n \rangle$ and $U = \langle u_1, u_2, \ldots, u_n \rangle$, the *Jaccard* similarity is defined as

$$\text{Jaccard}\,(V,U) = |V \cap U| \,/\, |V \cup U|.$$

The *Jaccard* similarity gives a value in the range from 0 to 1. The value 1 means the two vectors are exactly the same. In our controlled experiment, we also set the threshold value as 0.95.

Factor level *Hamming*: Given the feature vector representation of the GUI state, we can calculate the distance between two GUI states using the *Hamming* distance. Given two feature vector $V =< v_1, v_2, \ldots, v_n >$ and $U =< u_1, u_2, \ldots, u_n >$, their *Hamming* distance is the number of positions where the two feature vectors are different. We normalized the *Hamming* distance in between 0 to 1 after distance calculation. We set the threshold value as 0.95 for Hamming distance in this paper.

TABLE II
BFS-BASED SEARCH ALGORITHM

```
1:  obtain the root UI widget from the current GUI state
2:  Q←empty queue
3:  ret←empty list  //return the list of clickable UI widget
4:  enque root into Q
5:  while Q is not empty do
6:    x ←get Q's head
7:    insert x to clickable list ret
8:    enque all child widgets of x into Q
9:  end-while
10:  return ret
```

B. Search Strategy

Search strategy determines the order of widgets within a GUI state to interact with. We can construct the widget tree by collecting all clickable widgets reachable from the root widget. As presented in Table I, different orders of interacting with these widgets in the same widget tree determines the traversal order of the GUI state model. For instance, upon clicking the chosen widget, the app under test may transit to a new or an existing state, which will be the next GUI state to explore.

Note that the target of the whole test case generation algorithm is the traversal of all clickable widgets. Different search strategies (BFS, DFS, and Random) will only affect the order of clickable widgets to traverse. All clickable widgets will be traversed based on the test case generation algorithm, finally.

In this paper, we investigate three search strategies: BFS [13], DFS [3], and *Random Search* (Random) [46].

Table II presents the BFS search strategy. In this paper, our tool traverses the GUI widget tree with a queue data structure. First, the algorithm enques the root node into an empty queue. Next, it deques the first widget and insert it into the list *ret*. Then, it enques the children of this widget into the queue. The process repeats until the queue becomes empty. Finally, the returned ordered list determines the order of the widgets to interact with the corresponding GUI state. Fig. 2 presents an exemplified GUI widget tree where nodes denote clickable widgets while edges denote their parental relationship. For BFS traversal, n_0 enques first. Then, it is moved from the queue to the returned list. Next, n_1 and n_2 are put into the queue and then moved to the list in turn.

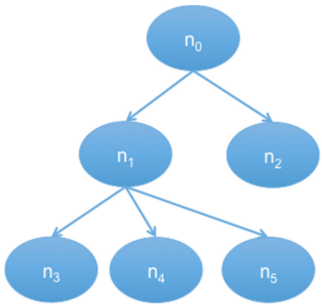

Fig. 2. Exemplified GUI Widget Tree.

TABLE III
DFS-BASED SEARCH ALGORITHM

```
1:  get root GUI widget in the current GUI state
2:  S←empty stack
3:  ret←empty list //return the list of clickable UI Widget
4:  push root into S
5:  while S is not empty do
6:    x← S.pop()
7:    Insert x into ret
8:    push the child widgets of x into S
9:  end-while
10: return ret
```

Finally, n_3, n_4, and n_5 are put into the queue and then moved to the list in turn.

The DFS next-click algorithm is presented in Table III. This algorithm is similar to the BFS algorithm, except that it uses a stack instead of a queue. We also use the tree in Fig. 2 for illustration purpose: it first places n_0 into the stack. Then, it clicks on n_0 and pops it out from the stack. Next, it places n_1 into the stack followed by clicking n_1 and popping it out from the stack. The processing on n_3, n_4, and n_5 are similar. Finally, the algorithm places n_2 into the stack and clicks n_2.

The abovementioned two algorithms get the next widget to click from the ordered clickable list. In contrast, *Random* just randomly shuffles the clickable widgets within the widget tree and insert them into the list to return.

C. Waiting Time

The waiting time factor determines the time to wait between sending two consecutive input events. Different waiting times may result in different following up GUI states to interact with. One option is to wait until the next GUI state is stable, while the other option is to set a fixed time period for waiting. The choice of waiting time is a tradeoff. On one side, a short waiting time may lead to unnecessary application not responding or unexpected application behaviors. On the other side, a long waiting time may waste precious testing time.

In this empirical study, we choose four typical factor levels of waiting time because they were used in existing GUI traversal-based test case generation tools. The first one is to use the waitForIdle() method of the UIAutomator API in PUMA [13], whose functionality is to wait until the execution of the app becomes idle. This API call checks nothing notable on the GUI happening for a certain amount of time. We refer to this strategy as *wait-for-idle*.

Other Android testing tools adopt different strategies for timing control. In our controlled experiment, we studied the following waiting time periods: 200 ms as in Shauvik *et al.* [31] to control the Monkey tool in their experiment, and 3000 ms as in ACTEve [2], and 5000 ms as in SwiftHand [5]. We refer to them as *wait200ms*, *wait3000ms*, and *wait5000ms*, respectively, and collectively call them as *wait-for-a-while*.

TABLE IV
THREE FACTORS AND THEIR LEVELS

Factor Level	State Equivalence	Search Strategy	Waiting Time
0	Cosine	BFS	waitForIdle
1	UI Hierarchy	DFS	wait200ms
2	ActivityID	Random	wait3000ms
3	Jaccard	—	wait5000ms
4	Hamming	—	—

D. Summary of Factors and Factor Levels

Table IV presents a summary of the factor levels for each factor studied in this paper. For GUI state equivalence, the five levels are Cosine, UI hierarchy, ActivityID, Jaccard, and Hamming. For search strategy, the three levels are BFS, DFS, and Random. For waiting time, the four levels are waitForIdle, *wait200ms*, *wait3000ms*, and *wait5000ms*.

IV. CONTROLLED EXPERIMENT

In this section, we report our controlled experiment and its data analysis.

A. Research Questions

We aim to answer the following six research questions.

1) **RQ1**: Is there any difference in choosing different definitions of state equivalence in terms of testing effectiveness?
2) **RQ2**: Is systematic search strategy superior to random search in terms of testing effectiveness?
3) **RQ3**: Does the choice of waiting time strategy have a significant impact on testing effectiveness when the time for testing is fixed to a relatively long period such as 1 h?
4) **RQ4:** Does the choice of waiting time strategy have any significant impact on rate of fault detection and code coverage if the time for testing is very limited?
5) **RQ5:** Does the choice of search strategy have any significant impact on rate of fault detection and code coverage if the time for testing is very limited?
6) **RQ6**: Is there any good combination of these factor levels in terms of failure detection rate and code coverage?

B. Benchmarks

A total of 33 real-life open-source Android apps are included as our benchmarks. They were also used in four previous works: sixteen for Dynodroid [24], three for A3E [3], five for ACTEve

TABLE V
LIST OF APPS USED IN THIS PAPER

Android apps	Version	Category	Also Used by
BookCatalogue	1.6	Tool	A^{3}E
TomdroidNotes	2.0a	Social	
Wordpress	0.5.0	Tool	
SpriteMethodTest	-	Example	ACTEve
RandomMusicPlayer	1	Music	
CountdownTimer	1.1.0	Tool	
Ringdroid	2.6	Media	
Translate	3.8	Tool	
Nectroid	1.2.4	Media	DynoDroid
MunchLife	1.4.2	Game	
Addi	1.98	Tool	
Photostream	1.1	Media	
SyncMyPix	0.15	Media	
aLogCat	2.6.1	Tool	
Multi SMS	2.3	SMS.	
BaterryDog	0.1.1	Tool	
NetCounter	0.1.4	Tool	
DivideAndConquer	1.4	Game	
HotDeath	1.0.7	Card	
Bomber	1 1	Game	
Auto Answer	1.5	Tool	
PasswordMakerPro	1.1.7	Tool	
K-9 Mail	3.512	Email.	
AardDictionary	1.4.1	Reference	
LearnMusicNotes	1.2	Fun	SwiftHand
MiniNoteViewer	0.4	Tool	
TippyTipper	1.1.3	Finance	
WeightChart	1.0.4	Health	
Sanity	2.11	Tool	
Mileage	3.1.1	Finance	
MyExpenses	1.6.0	Finance	
Whohasmystuff	1.0.7	Tool	
DalvikExplorer	3.4	Tool	

[2], and nine for SwiftHand [5]. Table V presents the descriptive statistics of these benchmarks.

C. Experimental Setup

We have realized all the factor levels listed in Table IV by extending the PUMA framework, and used our extended PUMA tool to test each benchmark under each combination of factor levels.

We used two virtual machines (VMs) configured with Ubuntu 14.04 as the client OS to perform our experiment. To manage the VMs, we used VirtualBox [47]. VirtualBox allows us to manage the client OSs of multiple VMs independently. For each VM, we configured it with a 2-core processor and 6 GB of memory. We used the Vagrant tool to help manage the VirtualBox images.

D. Experimental Procedure

By combining different factor levels among the three factors, we generated 60 (i.e., $5 \times 3 \times 4$) combinations in total. Each combination of them was a complete configuration of our enhanced framework. For each such configuration, each benchmark was tested for 60 min. In other words, we assume a 1 h test session for data analysis to study RQ1, RQ2, RQ3, and RQ6. We gathered the execution results and code coverage information after each execution. In total, we collected the dataset Z based on the abovementioned experiment with 1980 testing hours. For code coverage information, there are 24 new treatments for 33 apps, meaning that there are about 792 or more data points to evaluate each research question.

For RQ1–RQ3, we study the impacts of different levels for each factor on testing effectiveness (i.e., code coverage and failure detection rate in this paper). To analyze the influence of different factor levels of the same factor, we partitioned the dataset Z into equivalence classes (i.e., one equivalence class for one distinct factor level for that factor). For instance, to study the impacts of different levels of the factor *state equivalence*, we partitioned the dataset Z according to their state equivalence strategies and compared the data among the corresponding equivalence classes using a statistical test. For RQ1 to RQ3, one-way ANalyses Of VAriances (ANOVAs) was employed to check whether there were any significant differences in the median values of the data in these equivalence classes followed by a pair-wised comparison on their mean values through the multiple comparison procedure. In the entire experiment, we set the significance level of each statistical test to the 5%.

For RQ4 and RQ5, we aim to investigate whether there is any difference between different waiting time strategies as well as search strategies in terms of testing effectiveness when the time allowed for testing is limited. To analyze the rate of fault detection, we logged the code coverage information and the fault exposure information at the end of every minute (i.e., at the end of testing for 1 min, 2 min, until 20 min). We compared the rate of the fault detection and code coverage within such time interval for different waiting time factor levels.

For RQ6, we study the impact of different treatments (i.e., different combinations of factor levels among the three factors) on testing effectiveness. When we investigated the impact of one treatment, we fixed the corresponding factor levels, and compared the results of each combination of unfixed factor levels. For example, when we compared different treatments with two factor levels fixed to ⟨*search strategy, waiting time*⟩, we partitioned the dataset Z into equivalence classes where one such class for one distinct definition of *state equivalence.*

In our controlled experiment, we used two evaluation metrics to measure testing effectiveness. They are failure detection rate and code coverage [6]. These two metrics were frequently used in existing studies (see, e.g., [31]).

Our tool was equipped with *Emma* [41] to gather code coverage data. To obtain the testing result of each run, the tool parsed the *logcat* file for the run and extracted the exceptions and error messages and the stack trace information. Some failures may occur more than once in one or more runs. Therefore, for each equivalence class of dataset Z, we considered two failures the same if they associated with the same stack traces and produced the same type of error messages. We consolidated the same failures together and only counted the consolidated failure as one failure when measuring the failure detection rate. To differentiate the failures during different testing periods, we checked the timestamps of their corresponding logs for grouping.

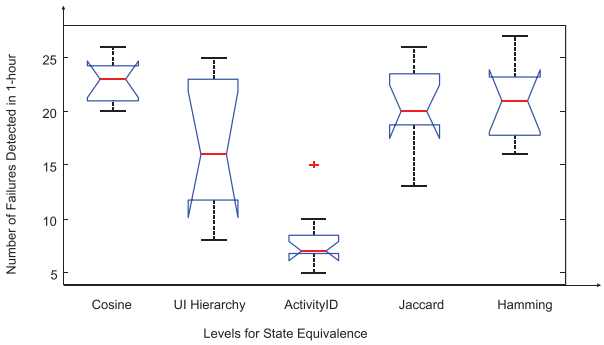

Fig. 3. Comparison on failure detection rate (state equivalence).

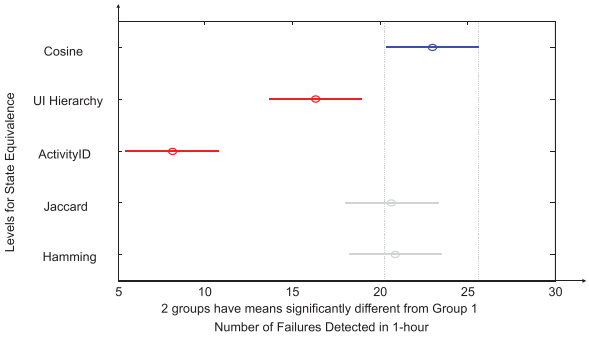

Fig. 4. Multiple comparison on failure detection rate (state equivalence).

E. Results and Analysis

In this section, we analyze our experimental result and answer the research questions raised in Section IV-A.

1) Answering RQ1: The box-whisker plot in Fig. 3 presents number of detected distinct failures within the 1 h test session for the factor levels of state equivalence. Within the box-whisker plot, the (red) horizontal line represents the median value, and the upper and lower box borders represent the values at 25% and 75% quartiles. Finally, the whiskers are the two lines outside the box that extend to the highest and lowest observations. When the notches of two boxes overlap with each other, it roughly represents that there is no statistical difference between the median values of the two groups of data, and vice versa.

The notches of UI Hierarchy and ActivityID do not overlap with each other. This indicates that the median values of UI Hierarchy and ActivityID differ significantly from each other. However, Cosine and UI Hierarchy do have small overlapping. Moreover, we observed that in terms of median value, Cosine is located higher than UI Hierarchy, and the latter is located higher than ActivitiyID. The notches of Cosine, Jaccard, and Hamming overlap with each other, but the median value of Cosine is higher than that of Hamming, which in turn is higher than that of Jaccard.

In Fig. 4, we further perform multiple comparisons among different groups of data in terms of their mean values. Nonoverlapped lines represent groups of values that are significantly different from each other in terms of mean values whereas

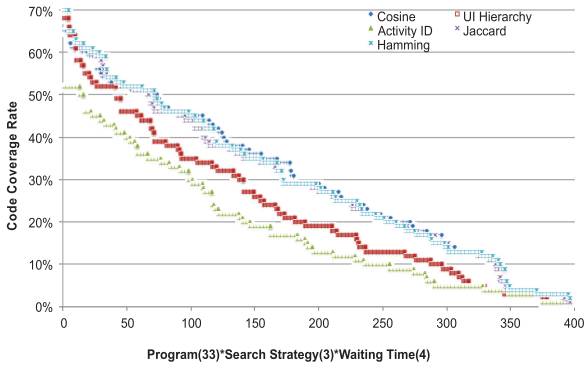

Fig. 5. Distribution of code coverage (state equivalence).

overlapped lines represent groups of values having no significant difference from each other. The blue line represents the group to compare with (i.e., Cosine), the grey lines represent groups that have no significant difference from the group Cosine (i.e., Jaccard and Hamming) in terms of mean values, the red lines represent groups that are significantly different from group Cosine (i.e., ActivityID and UI Hierarchy) in terms of mean values. The two dotted lines help reader to understand the overlapping relationships between different groups.

We observed that the line representing Cosine, UI Hierarchy, and ActivityID do not overlap with one another. This result shows that Cosine, UI Hierarchy, and ActivityID were decreasingly less effective in detecting failures in a statistically meaningful way. We also observed that the two newly proposed state equivalence definitions Jaccard, and Hamming overlap a little with the UI Hierarchy. In fact, Jaccard and Hamming lies in between Cosine and UI Hierarchy observably.

Fig. 5 shows the distributions of the code coverage for the five GUI state equivalence definitions on all combinations in a scatter plot. The use of 33 benchmarks, three search strategies, and four waiting time as a whole generates 396 combinations in total for each GUI state equivalence definition. We show the data based on their corresponding code coverage rates in the scatter plot. The *x*-axis shows the 396 combinations, whereas the *y*-axis shows their corresponding code coverage rate. In general, the area below each dotted line in Fig. 5 represents the accumulated code coverage achieved by each GUI state equivalence definitions. The larger the area, the better code coverage achieved by the corresponding GUI state equivalence definition. We can see clearly that Cosine, Jaccard, and Hamming have similar areas and they all have larger area than Activity ID and UI Hierarchy. Therefore, we can conclude that the code coverage rates of *Cosine*, *Hamming*, and *Jaccard* are very close to each other. Moreover, they consistently perform better than *UI Hierarchy* and *ActivityID*.

Fig. 6 further shows that the ANOVA results for the five definitions of state equivalence in terms of code coverage. We can find that the *Cosine*, *Jaccard*, and *Hamming* fall into the top tier group. The notches of their box-whisker do not overlap with *UI Hierarchy* and *ActivityID*, which show their median values differ significantly from each other. As shown in Fig. 7, the

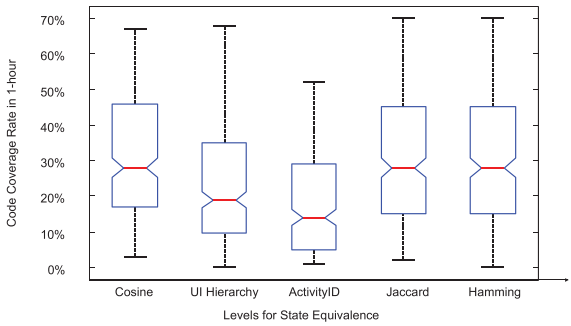

Fig. 6. Comparison on code coverage (state equivalence).

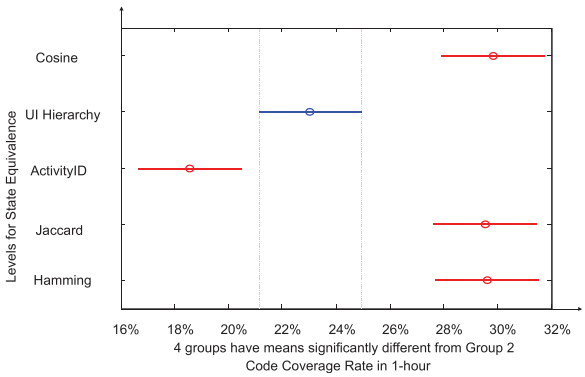

Fig. 7. Multiple comparisons on code coverage (state equivalence).

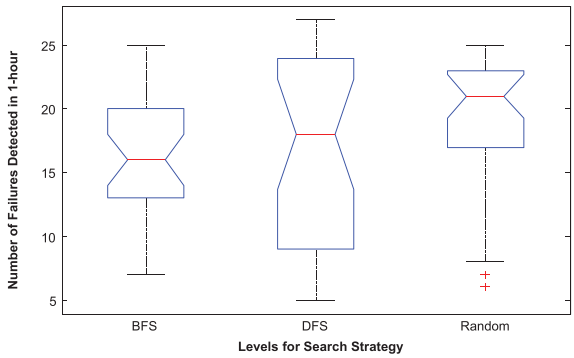

Fig. 8. Comparison on failure detection rate (search strategy).

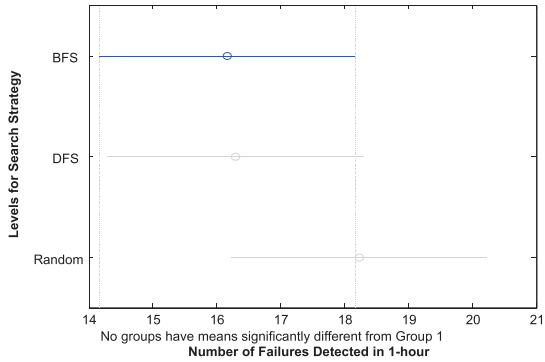

Fig. 9. Multiple comparisons on failure detection rate (three search strategies).

multiple comparison procedure also confirmed that the former three achieved higher code coverage than the latter two, and *UI Hierarchy* achieved higher code coverage *ActivityID* as well.

The definitions of Cosine, Hamming, and Jaccard use both the GUI structure information and GUI widget properties. They are finer in granularity than UI Hierarchy and ActivityID, meaning that the former three can produce finer GUI state models to be explored than the latter two. As a result, within the same testing period, the GUI traversal-based algorithms configured with the former three can have a better odd to explore more GUI states, leading to better code coverage and fault detection ability. UI Hierarchy is coarser than Cosine (and Hamming and Jaccard as well) but finer than ActivityID. This aligns with the empirical findings on the relative order of their testing effectiveness.

Based on the analysis findings on failure detection rate and code coverage, we found that using different definitions of state equivalence for GUI traversal-based algorithms have significant impact on the testing effectiveness of these algorithms. Moreover, the use of finer the state equivalence definition can achieve higher testing effectiveness.

We can answer RQ1 that using a finer definition of state equivalence can lead to a statistically higher failure detection rate and higher code coverage. Furthermore, observably, *Cosine* performs best in terms of failure detection rate, but *Hamming* and *Jaccard* are statistically comparable to *Cosine*. In terms of code coverage, these three definitions of state equivalence performed better than *UI Hierarchy* and *ActivityID* and were comparable to each other.

2) Answering RQ2: Fig. 8 shows the ANOVA result on the failure detection rates for the factor *Search Strategy*, and the multiple comparison results are depicted in Fig. 9.

In Fig. 8, the median values of DFS and Random do not differ significantly from each other. Furthermore, the notches of Random and BFS do not overlap with each other, indicating that Random performs better than BFS significantly in terms of median values.

Fig. 9 presents the results of the multiple comparison procedures for different search strategy. Each horizontal line represents groups of values for a corresponding level. Nonoverlapped lines represent groups of values that are significantly different from each other in terms of mean values, whereas overlapped lines represent groups of values having no significant difference from each other. From Fig. 9, we can find the mean values of different search strategies do not differ significantly as they overlap with each other.

Nonetheless, Random is observably more effective than DFS and BFS in terms of mean values. Random is simpler than BFS and DFS to implement, but can attain statistically comparable failure detection rates, which is interesting.

Fig. 10 shows that the three factor levels for *Search Strategy* attained comparable median code coverage, which was confirmed by the multiple comparison procedure as shown in Fig. 11.

The overall results show that Random is a surprisingly good search strategy for GUI traversal-based test case generation due to its simplicity.

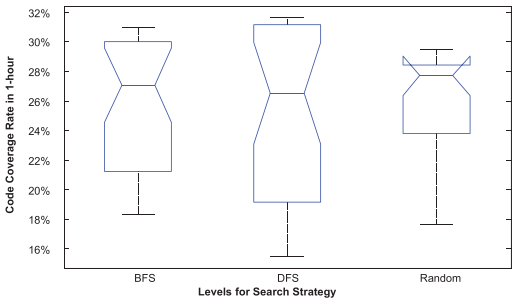

Fig. 10. Comparison on code coverage (search strategy).

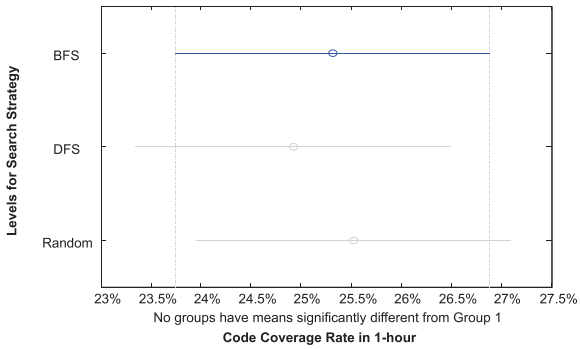

Fig. 11. Multiple comparisons on code coverage (search strategy).

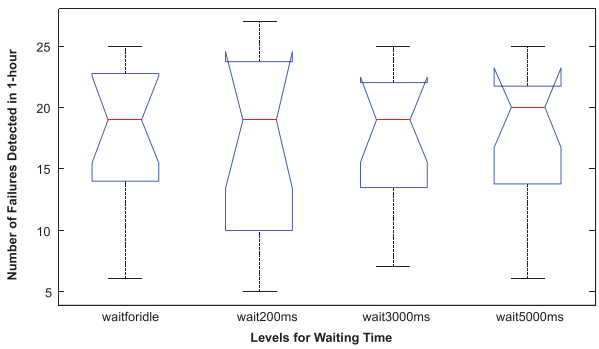

Fig. 12. Comparison on failure detection rate (waiting time strategy).

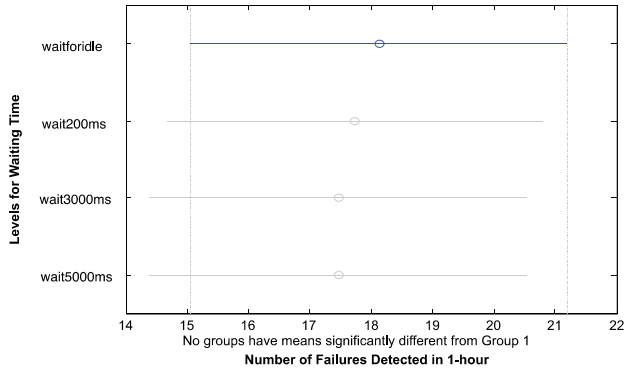

Fig. 13. Multiple comparisons on failure detection rate (waiting time strategy).

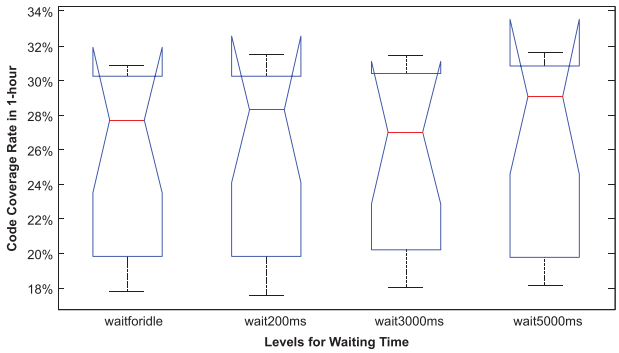

Fig. 14. Comparison on code coverage (waiting time strategy).

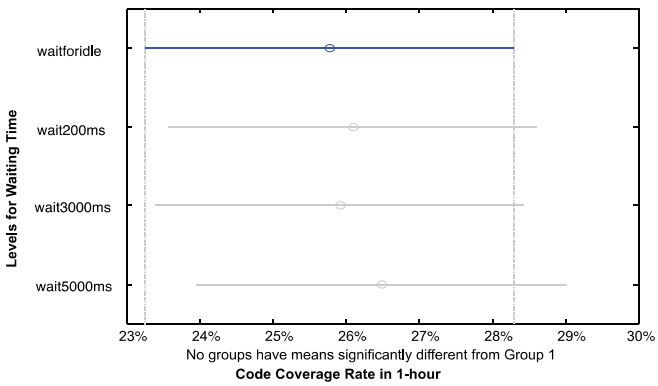

Fig. 15. Multiple comparisons on code coverage (waiting time strategy).

We can answer RQ2 that Random is statistically comparable to the two systematic search strategies (DFS and BFS) in terms of testing effectiveness.

3) Answering RQ3: We can see from Fig. 12 that the median values of using different waiting time do not differ significantly from each other in terms of failure detection rate.

The result of the multiple comparison procedure shown in Fig. 13 confirms that the mean values of these four strategies in terms of failure detection rate have no significant difference at the 5% significance level.

From Fig. 14, the median values of using different waiting time do not differ significantly in terms of code coverage. Similarly, the multiple comparison results in Fig. 15 confirm that they also have comparable mean values statistically at the 5% significance level. They show that waiting time strategy is not an important design factor for the metric code coverage when the testing time is long enough, which is a 1 h test session in our controlled experiment.

We can answer RQ3 that when the testing time is long enough (1 h in the experiment), different waiting time factor levels have no significant impact on testing effectiveness.

4) Answering RQ4: In many production scenarios, mobile applications update frequently, which makes testing time allowed in each development cycle limited. Therefore, we want to

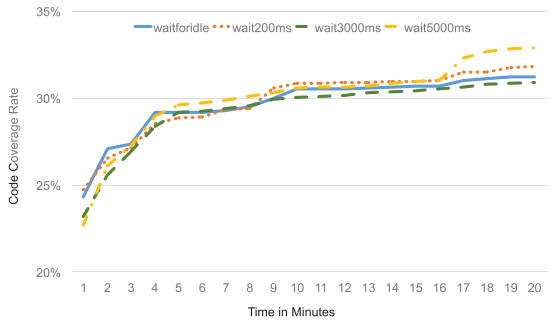

Fig. 16. Impact of waiting time strategy on code coverage with limited testing time.

Fig. 17. Impact of waiting time strategy on failure detection with limited testing time.

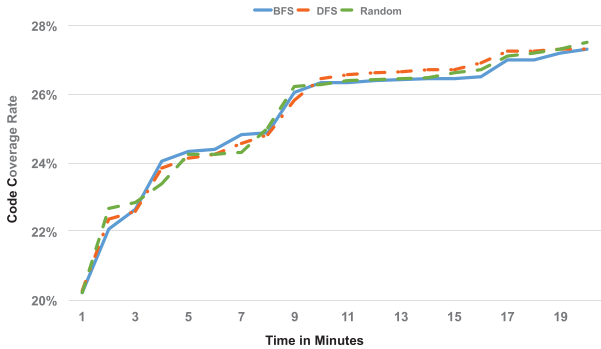

Fig. 18. Impact of search strategy on code coverage with limited time.

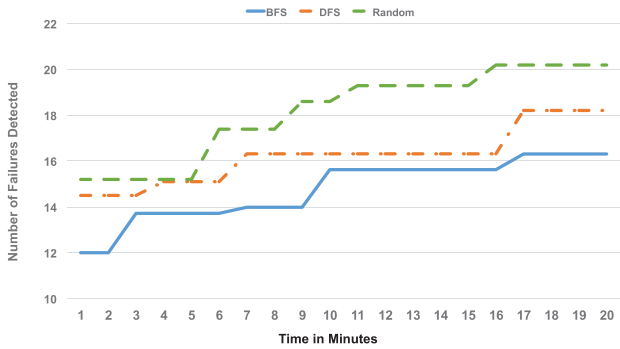

Fig. 19. Impact of search strategy on failure detection with limited time.

further understand the impact of waiting time on testing effectiveness when the testing time is constrained in this section. To be specific, we want to study whether different waiting time strategies have any significant differences in testing effectiveness if the time for testing is limited.

Fig. 16 shows the impact of different waiting time strategies on testing effectiveness when the testing time is limited. The *x*-axis presents the testing time in minutes, and the *y*-axis shows the percentage of code coverage achieved. Thus, the figure reflects the speed of different testing strategies to achieve code coverage. The code coverage over 20 min is not shown as all techniques have converged already.

When the testing time is limited, we can see that the code coverage rates of different waiting time strategies are close to one another: There is no consistent winner.

We then further studied the number of failures detected when the testing time is limited, where the results are as shown in Fig. 17. The *x*-axis shows the testing time in minutes, and the *y*-axis shows the number of failures detected. We can see that *waitForIdle* consistently performs the best followed by *wait5000ms*. Shorter waiting time appears to detect less failures than *wait5000ms* after the first few minutes. Therefore, if the testing time is limited, the data shows that the *waitForIdle* strategy have the best failure detection rate among the strategies studied.

We can answer RQ4 that the *waitForIdle* is a more effective strategy in terms of failure detection rate, but nonetheless, there is no consistent winning strategy in terms of code coverage.

The combined results of *RQ3* and *RQ4* show that different waiting time strategies have different convergence speeds in terms of failure detection rate (but not code coverage). The choice of best strategy depends on the testing time available in practice.

5) Answering RQ5: In this section, we want to further understand the impact of search strategy on testing effectiveness when the testing time is limited.

Fig. 18 shows the impact of different search strategies on testing effectiveness when the testing time is limited. The *x*-axis presents the testing time in minutes, and the *y*-axis shows the percentage of code coverage achieved. Thus, the figure reflects the speed of different search strategies in terms of code coverage. The code coverage over 20 min is also not shown as all techniques have converged. When the testing time is limited, we can see that the code coverage rates of different waiting time strategies are close to each other.

We then further studied the number of failures detected when the testing time is limited, where the results are as shown in Fig. 19. The *x*-axis shows the testing time in minutes, and the *y*-axis shows the number of failures detected. We can see that *Random* consistently performs better than DFS, which in turn is better than BFS. Therefore, if the testing time is limited, the data shows that Random has the best failure detection rate among the strategies studied.

We can answer RQ5 that Random is a more effective strategy than DFS or BFS in terms of failure detection rate when the testing time is limited. However, there is no consistent winning strategy in terms of code coverage.

TABLE VI
BEST TREATMENT IN FAILURE DETECTION

Fixed level	Treatment and Metric	
	Best Strategy	*# of Detected Failures*
None is fixed	(4,1,1)	27
Cosine	(0,1,1)	26
UI Hierarchy	(1,2,0)	25
ActivityID	(2,0,1)	15
Jaccard	(3,1,1)	26
Hamming	(4,1,1)	27
BFS	(0,0,3)	25
DFS	(4,1,1)	27
Random	(1,2,0)	25
waitForIdle	(0,0,0)	25
wait200ms	(4,1,1)	27
wait3000ms	(0,1,2)	22
wait5000ms	(0,0,3)	25

6) Answering RQ6: In this section, we report our analysis on studying whether there are any good treatments among the three factors. We first summarize the best treatment(s) in Table VI for failure detection rate and in Table VIII for code coverage.

We use a triple (*i, j, k*) to encode each treatment, where the levels of *i, j,* and *k* are as given in Table IV. In such a triple, the *i* component refers to a particular factor level of *state equivalence*, the *j* component refers to a particular factor level of *search strategy*, and the *k* component refers to a particular factor level of *waiting time*.

As given in Table VI, in terms of failure detection rate, the treatment (*4,1,1*), which is ⟨*Hamming, DFS, wait200ms*⟩ the best treatment if no factor is fixed to a particular level. It has exposed 27 failures with 1 h testing. For the rows with one fixed factor level (e.g., *Cosine* in the first column), the second column shows the best treatment if that factor level shown in the first column is fixed. For instance, if the GUI state equivalence is fixed to *Cosine* (0,1,1), which means ⟨*Cosine, DFS, wait200ms*⟩ is the best treatment. The other rows can be interpreted similarly.

Note that the result of the best treatment is in fact consistent with the result of RQ1 on the best choice of GUI state equivalence definition. The choice of best GUI state equivalence definition is found by performing statistical comparison among all groups of results with their GUI state equivalence definition fixed. In another word, we are comparing groups of values $\langle 0, X, X\rangle$, $\langle 1, X, X\rangle$, $\langle 2, X, X\rangle$, $\langle 3, X, X\rangle$, and $\langle 4, X, X\rangle$ statistically, where X represents any value for a factor. In contrast, the results for the best treatment are in fact comparing the effectiveness of each fixed combination of factor levels (i.e., treatments). In another word, we are comparing all 60 possible combinations: ⟨0,0,0⟩, ⟨0,0,1⟩, ⟨0,0,2⟩, ⟨0,0,3⟩, ⟨1,0,0⟩, ⟨1,0,1⟩, ⟨1,0,0⟩, etc. Even if the group of values for $\langle 0, X, X\rangle$ is statistically better than the groups of values for $\langle 4, X, X\rangle$, it is normal for one specific treatment (i.e., ⟨4,1,1⟩) in $\langle 4, X, X\rangle$ to be better than any specific treatment in $\langle 1, X, X\rangle$.

Moreover, based on Fig. 3, Cosine is only observably better than Hamming. Statistically, there is no significance between Cosine and Hamming. From the upper whisker of Cosine and Hamming in Fig. 3, we can find there are indeed combinations of factors with Hamming distance performs better than Cosine, which is consistent with the results of the current research question.

In practice, if the test engineers have the freedom to configure all three factors with factor levels studied in this paper for their testing tool, they can adopt the results of RQ6 to use the best combinations of factor levels (e.g., choosing ⟨*Hamming, DFS, wait200ms*⟩). On the other side, if the factor levels used for search strategy and waiting time is unknown yet (i.e., possibly any values studied this paper) or new (i.e., factor levels not studied in this paper), we suggest the test engineers choose *Cosine* since it has a higher chance to get better testing effectiveness.

We further classified the 27 failures found by the best treatment as given in Table VII. Each row represents one type of failure based on their exception class name. The column *Count* represents the number of failures of that type and the last column shows the apps where such failures were found.

We can see that NullPointerException and RuntimeException are the two major types of failures detected by the test case

TABLE VII
CLASSIFICATION OF FAILURES OF THE BEST TREATMENT

No.	Exception in stack trace	Count	Applications
1	java.lang.NullPointerException	13	LearnMusicNotes, MiniNoteViewer, Nectroid,MunchLif, Addi,Mileage, SyncMyPix, MyExpenses, aLogCat, NetCounter, Bomber, DalvikExplorer, AardDictionary
2	java.lang.RuntimeException	9	SpriteMethodTest, Addi, TomdroidNotes, Photostream, aLogCat, Ringdroid, hotdeath, DivideAndConquer, AardDictionary
3	java.lang.Exception	1	BaterryDog
4	java.lang.ArrayIndexOutOfBoundsException	1	BaterryDog
5	android.content.ActivityNotFound Exception	1	MiniNoteViewer
6	java.io.IOException	1	RandomMusicPlayer
7	android.database.StaleData Exception	1	Ringdroid

TABLE VIII
BEST TREATMENT IN CODE COVERAGE

Fixed level	Combination and Metrics		
	Best Strategy	*Average (%)*	*Variance (%)*
None is fixed	(0,1,3)	31.64	17.79
Cosine	(0,1,3)	31.64	17.79
UI hierarchy	(1,2,2)	29.48	18.35
ActivityID	(2,0,2)	20.56	14.86
Jaccard	(3,1,1)	31.24	17.34
Hamming	(4,1,1)	31.12	17.45
BFS	(0,0,3)	31	18.35
DFS	(0,1,3)	31.64	17.79
Random	(1,2,2)	29.48	18.35
waitForIdle	(0,1,0)	30.88	17.44
wait200ms	(0,1,1)	31.52	17.64
wait3000ms	(0,1,2)	31.42	17.67
wait5000ms	(0,1,3)	31.64	17.79

generation tool. This indicates that mobile developers should be more aware of such types of failures during development.

Table VIII presents the best treatments in terms of code coverage. If no factor is fixed to a particular factor level, treatment (*0,1,3*), meaning ⟨*Cosine, DFS, wait5000ms*⟩ is the best treatment. The corresponding mean code coverage is 31.64% with a variance of 17.19%. If one factor level is fixed, the corresponding rows can be interpreted similarly.

We observed that the best treatments for failure detection rate and code coverage rate vary. The result shows that the best choice of factor levels should be determined by the actual constraints in practice.

F. Threats to Validity

We used 33 apps with real faults to perform the experimental study. These apps are also frequently used in existing paper. The use of other apps as benchmarks may lead to different observations.

The correctness of our tools was one potential issue affecting the validity of our controlled experiment. We have realized all the three factors and their levels within the same framework and the virtual-box based experiment framework was adapted from [31]. We have reviewed and tested our code thoroughly to reduce possible bugs in our implementation.

The factors studied in this paper are based on the abstraction of the PUMA framework. In addition, there may be different factors not considered in this paper that may also affect testing effectiveness. We will leave the exploration of these factors in future work.

Following the existing works [13], [29], [48], we used code coverage rate and failure detection rate to evaluate testing effectiveness. An experimental study on other metrics may lead to different observations.

Within our experimental framework, the GUI state is considered stable when nothing notable happens in the UI for a certain amount of time. In addition, the waiting time factor is defined as the time to wait between two input events. Inserting system events when the application is stable can trigger more callback code within the application under test. However, system events are also considered input events within our testing framework. Therefore, within our testing framework, stability is an attribute observed rather than controlled during testing. On the other hand, inserting system events (as well as other types of inputs) in addition to clicks are surely interesting factor to explore. We will leave it in the future as it is beyond the scope of this paper.

V. Related Work

A. Test Development Platform for Android Apps

There are many popular test development environments and tools, such as MonkeyRunner [43], Robotium [44], and UIAutomator [38]. These test development platforms usually provide a set of APIs for the testers to write test scripts based on their own test requirements.

MonkeyRunner [43] is a testing tool within the Android SDK. It provides API interfaces in Python for the testers to interact with the device, to simulate user inputs, and to get testing results for verification. The script is interpreted by the MonkeyRunner to interact with the application under test. Moreover, it also provides a mechanism to capture screen and compare image to support the test oracle procedure. Robotium [44] uses the Android instrumentation framework to support black-box automated testing by extending those instrumentation classes. UIAutomator [38] allows testers to craft test scripts to send input events to UI widgets rather than screen coordinates. Appium [40] uses the WebDriver protocol to test iOS, Android, and Windows apps. It also supports several programming languages so that testers can easily adopt it for use. In [17], we reported a test development platform for Android-based smart TV applications focusing on stress testing. It also supports the automatic testing of voice control applications.

B. Test Case Generation Techniques for Android Application

Hu and Neamtiu [15] presented an event-based test case generation tool for Android apps. They used the JUnit tool to help generate initial test cases based on the source code of the app. Starting from each test case, their tool used the Monkey tool to insert target events to simulate user interactions. When the test execution has finished, the tool checks logs to identify failures.

Model-based testing technique [11] constructs the model of an app followed by traversing the model systematically for test case generation. The tool proposed by Amalfitano *et al.* [1] uses a crawler-based technique to build a GUI model of an app, and then generates input event sequences by traversing the GUI model. Hao *et al.* [13] proposed the PUMA tool, which can perform dynamic analysis and test case generation for Android apps and allows users to flexibly extend the PUMA framework to incorporate different kinds of dynamic analyses.

A3E [3] is a GUI traversal-based test generation tool with two complementary exploration strategies: *A3E-Depth-First* and *A3E- Targeted*. The former strategy performs DFS on a dynamic model of the app where the dynamic model defines each activity as a distinct GUI state. The latter strategy is more complicated. It applies static taint analysis to build an activity transition graph of the app, and then uses the graph to efficiently generate intents.

Choi *et al.* [5] proposed the SwiftHand tool, which aims to maximize the code coverage of the application under test while minimizing the number of restarts during testing to save testing time. It builds a finite state model of the app for exploration to generate inputs dynamically. During the testing process, the tool will also dynamically improve the model based on the testing data.

In [32], Su *et al.* proposed a guided, stochastic model-based GUI testing technique for Android application. It combines static analysis and dynamic analysis to first build the GUI model of the application. Then, it mutates and refines the stochastic model to guide the test generation process. The results show that the technique is effective to achieve high code coverage and to trigger crash for Android application testing.

Dynodroid [24] realized a randomized exploration strategy. It is able to generate system events relevant to the app through

analysis. It can also either generate events that have been least frequently used or take the contexts into account. In other words, it can bias toward those events that are relevant in selected contexts.

Monkey [46] is a widely used stress-testing tool within the Android platform. It can generate pseudorandom sequences of user events to an app under test. Furthermore, many improved fuzzing tools over Monkey are also widely used. They either use widget level interaction or have enhanced failure diagnosis ability.

Sapienz [25] is a search-based test case generation tool. It takes failure detection rate, code coverage, and test case size as factors in optimizing its test cases during the search process. Their evaluation results show that Sapienz can be effective in generating smaller test cases and these test cases can result in high failure detection rate and high code coverage.

ACTEve [2] is a testing tool based on dynamic symbolic execution. Based on source code instrumentation, it performs concrete and symbolic execution to systematically generate test inputs. However, the requirement to instrument the application limits its applicability to application with source code only. Furthermore, it can only output test cases with four input events at most due the state explosion problem.

Crowdsourcing Android application testing is another popular approach in industry. In [36], Zhang *et al.* proposed an effective approach to help guide the testers on crowd-sourced platform to build high quality tests for Android application.

Inputting the appropriate text information to the text input widget of Android application is also important to effectively test Android applications. In [23], Liu *et al.* proposed to use deep learning to generate the input texts, which was empirically found to be effective.

There are also works focusing on other factors that may affect the testing effectiveness of test case generation techniques for Android apps. In [19], Li *et al.* proposed to generate input events and thread schedules at the same time to detect concurrency bugs in apps. In [33], Wu *et al.* proposed to generate relevant gesture events to effectively test Android application. It would be interesting to study these factors within our experimental framework in the future.

VI. Conclusion

In designing StateTraversal techniques for testing Android apps, there were several common configurable factors. However, the impact of factor levels for these factors on testing effectiveness has not been systematically studied. In this paper, we reported the *first* work to fill this gap by reporting a controlled experiment to study three such factors in a full factorial design setting. Our experiments revealed several interesting results. First, the choice of state equivalence can cause StateTraversal techniques to produce significantly different testing effectiveness. Second, applying Random and applying BFS or DFS on the same widget trees can be comparable in terms of testing effectiveness. Third, if a test session was long enough (i.e., 1 h in our controlled experiment), the choice between the wait-for-idle strategy and the wait-for-a-while strategy was immaterial. Also interestingly, if the test session was more limited in length, applying the wait-for-idle strategy and Random was observed to achieve the best failure detection rate in our study. Moreover, we have also proposed two new GUI state equivalence strategies (*Jaccard* and *Hamming*) that were statistically as effective as the existing best GUI state equivalence strategy. Finally, we have found that in the controlled experiment, configuring a StateTraversal technique with *Hamming*, *DFS,* and *wait200ms* detected the greatest number of failures, and configuring the technique with the treatment $\langle$*Cosine, DFS, wait5000ms*$\rangle$ achieved highest code coverage, which were more effective than the previous representative technique PUMA using the treatment $\langle$*Cosine, BFS, waitForIdle*$\rangle$.

In future, we would plan to intensively study the notions of GUI state equivalences. We would also work on the test generation for other types of Android application components, such as services and content providers. Randomized search strategy was a surprisingly effective strategy. We would also like to investigate the reasons behind such findings in future work.

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR—A tool for automated model-based testing of mobile apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sep./Oct. 2015.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, New York, NY, USA, 2012, pp. 1–11.

[3] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Programm. Syst. Lang. Appl.*, New York, NY, USA, 2013, pp. 641–660.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "P. Scout: Analyzing the Android permission specification," in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 217–228.

[5] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of android apps with minimal restart and approximate learning," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Programm. Syst. Lang. Appl.*, New York, NY, USA, 2013, pp. 623–640.

[6] M. H. Chen, M. R. Lyu, and W. E. Wong, "Effect of code coverage on software reliability measurement," *IEEE Trans. Rel.*, vol. 50, no. 2, pp. 165–170, Jun. 2001.

[7] T. Y. Chen and R. Merkel, "Quasi-random testing," *IEEE Trans. Rel.*, vol. 56, no. 3, pp. 562–568, Sep. 2007.

[8] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2012, vol. 81, no. 13, pp. 2454–2456.

[9] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: DetectingPrivacy leaks in iOS applications," in *Proc. Nat. Down Syndrome Soc.*, 2011, pp. 1–15.

[10] W. Enck *et al.*, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, Berkeley, CA, USA, 2010, pp. 1–6.

[11] M. C. Gaudel, "Testing can be formal, too," in *Tapsoft '95: Theory and Practice of Software Development* (Lecture Notes in Computer Science), P. D. Mosses, M. Nielsen, M. I. Schwartzbach, Eds. Heidelberg, Germany: Springer-Verlag, 1995, pp. 82–96.

[12] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, "Checking app behavior against app descriptions," in *Proc. 36th Int. Conf. Softw. Eng.*, New York, NY, USA, Jun. 2014, pp. 1025–1035.

[13] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst., Appl. Services*, New York, NY, USA, 2014, pp. 204–217.

[14] C. Hu and I. Neamtiu, "Automating GUI testing for android applications," in *Proc. 6th Int. Workshop Autom. Softw. Test*, New York, NY, USA, 2011, pp. 77–83.

[15] C. Hu and I. Neamtiu, "Testing of android apps," in *Proc. ACM Object-Oriented Programm., Syst., Lang. Appl.*, 2013, pp. 1–10.
[16] B. Jiang, Y. Y. Zhang, Z. Y. Zhang, and W. K. Chan, "Which factor impacts GUI traversal-based test case generation technique most? A controlled experiment on android applications," in *Proc. IEEE Int. Conf. Softw. Qual., Rel. Secur.*, Prague, Czech Republic, 2017, pp. 21–31.
[17] B. Jiang, P. Chen, W. K. Chan, and X. Zhang, "To what extent is stress testing of android TV applications automated in industrial environments?" *IEEE Trans. Rel.*, vol. 65, no. 3, pp. 1223–1239, Sep. 2016.
[18] M. Kechagia, D. Mitropoulos, and D. Spinellis, "Charting the API minefield using software telemetry data," *Empirical Softw. Eng.*, vol. 20, no. 6, pp. 1785–1830, 2015.
[19] Q. Li *et al.*, "Effectively manifesting concurrency bugs in android apps," in *Proc. 23rd Asia-Pac. Softw. Eng. Conf.*, Hamilton, New Zealand, 2016, pp. 209–216.
[20] X. Li, Y. Jiang, Y. Liu, C. Xu, X. Ma, and J. Lu, "User guided automation for testing mobile apps," in *Proc. 21st Asia-Pac. Softw. Eng. Conf.*, Jeju, South Korea, 2014, pp. 27–34.
[21] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 225–237, Apr. 2007.
[22] B. Liu, S. Nath, R. Govindan, and J. Liu, "DECAF: Detecting and characterizing Ad fraud in mobile apps," in *Proc. Nat. Spatial Data Infrastructure*, 2014, pp. 57–70.
[23] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proc. 39th Int. Conf. Softw. Eng.*, Piscataway, NJ, USA, 2017, pp. 643–653.
[24] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, New York, NY, USA, 2013, pp. 224–234.
[25] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for Android applications," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, New York, NY, USA, 2016, pp. 94–105.
[26] A. P. Mathur, *Foundations of Software Testing*. London, U.K.: Pearson Education, 2008.
[27] A. Memon, I. Banerjee, and A. Nagarajan, "GUI ripping: Reverse engineering of graphical user interfaces for testing," in *Proc. 10th Working Conf. Reverse Eng.*, Washington, DC, USA, 2003, pp. 260–2003.
[28] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*. Hoboken, NJ, USA: Wiley, 2011, pp. 7–8.
[29] D. Octeau, S. Jha, and P. McDaniel, "Retargeting android applications to java bytecode," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, New York, NY, USA, 2012, pp. 6:1–6:11.
[30] R. Rakesh, P. Rabins, and M. S. Chandra, "Review of search based techniques in software testing," *Int. J. Comput. Appl.*, vol. 51, no. 6, pp. 42–45, 2012.
[31] R. C. Shauvik, G. Alessandra, and O. O. Alessandro, "Automated test input generation for android: Are we there yet?," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Lincoln, NE, USA, 2015, pp. 429–440.
[32] T. Su *et al.*, "Stochastic model-based GUI testing of android apps," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, New York, NY, USA, 2017, pp. 245–256.
[33] X. Wu, Y. Jiang, C. Xu, C. Cao, X. Ma, and J. Lu, "Testing android apps via guided gesture event generation," in *Proc. 23rd Asia-Pac. Softw. Eng. Conf.*, Hamilton, New Zealand, 2016, pp. 201–208.
[34] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan, "IntentFuzzer: Detecting capability leaks of android applications," in *Proc. 9th ACM Symp. Inf., Comput. Commun. Secur.*, New York, NY, USA, 2014, pp. 531–536.
[35] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.
[36] X. Zhang, Z. Chen, C. Fang, and Z. Liu, "Guiding the crowds for android testing," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion*, Austin, TX, USA, 2016, pp. 752–753.
[37] Android. [Online]. Available: http://www.android.com. Accessed: Sep. 2017.
[38] UIAutomator. [Online]. Available: http://developer.android.com/tools/help/uiautomator/index.html. Accessed: Feb. 2018.
[39] AppBrain. "The number of applications in Google Play." [Online]. Available: https://www.appbrain.com/stats/number-of-android-apps. Accessed: Nov. 2017.
[40] Appium. [Online]. Available: https://www.appium.io/. Accessed: Sep. 2017.
[41] Emma. [Online]. Available: http://www.emma.sourceforge.net/. Accessed: Oct. 2017.
[42] Gartner. [Online]. Available: http://www.199it.com/archives/408226.html. Accessed: Nov. 2017.
[43] Monkeyrunner. [Online]. Available: http://www.android-doc.com/tools/help/monkeyrunner_concepts.html. Accessed: 2017.
[44] Robotium. [Online]. Available: https://github.com/RobotiumTech/robotium. Accessed: 2017.
[45] Software Testing Research Survey Bibliography. [Online]. Available: http://web.engr.illinois.edu/~taoxie/testingresearchsurvey.htm. Accessed: Aug. 2017.
[46] The Monkey UI Android Testing Tool. [Online]. Available: http://developer.android.com/tools/help/monkey.html. Accessed: 2017.
[47] VirtualBox. [Online]. Available: https://www.virtualbox.org/. Accessed: Sep. 2017.

Bo Jiang (M'08) received the Ph.D. degree in software engineering from The University of Hong Kong, Hong Kong, in 2011.

He is currently an Associate Professor of Software Engineering with the School of Computer Science and Engineering, Beihang University, Beijing, China. He also has extensive experience in research with industry partners. His current research interests include software testing, debugging, and blockchain technology. His research has been reported in leading journals and conferences, such as *Automated Software Engineering*, Fast Software Encryption Conference, International Conference on Web Services, International Conference on Software Quality, Reliability, and Security, the IEEE TRANSACTIONS ON SERVICES COMPUTING, IEEE TRANSACTIONS ON RELIABILITY, *Journal of Systems and Software*, *Information and Software Technology*, and Society of Plastics Engineers.

Dr. Jiang is a PC member for many conferences and a Guest Editor for the *Journal of Systems and Software*. He was the recipient of the four best paper awards from his conference publications.

Yaoyue Zhang (S'17) received the M.Sc. degree in software engineering from the School of Computer Science and Engineering, Beihang University, Beijing, China, in 2017.

She is currently with Baidu Inc., Beijing, China. Her research interest focuses on software testing.

Wing Kwong Chan (M'04) received the B.Eng., M.Phil., and Ph.D. degrees from The University of Hong Kong, Hong Kong.

He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong, Hong Kong. His research results have been reported in more than 100 major international journals and conferences. His current research interests include addressing the software testing and program analysis challenges faced by developing and operating large-scale applications.

Dr. Chan is an Editor for the *Journal of Systems and Software* and *International Journal of Creative Computing*, and a Guest Editors for several international journals. He was a PC/RC member of many international conferences and was the Program Chair of several conferences and workshops.

Zhenyu Zhang (M'06) received the Ph.D. degree in software engineering from The University of Hong Kong, Hong Kong, in 2009.

He is currently an Associate Professor of Software Engineering with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. He has published research results in venues, such as the IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, *Computer*, International Conference on Software Engineering, Fast Software Encryption Conference, *Automated Software Engineering*, and *World Wide Web*. His current research interests include program debugging and testing for software and systems.