Which Factor Impacts GUI Traversal-Based Test Case Generation Technique Most? *

A Controlled Experiment on Android Applications

Bo Jiang, Yaoyue Zhang
School of Computer Science and Engineering
Beihang University
Beijing, China
{jiangbo, zhangyaoyue }@buaa.edu.cn

W.K. Chan†
Department of Computer Science
City University of Hong Kong
Hong Kong
wkchan@cityu.edu.hk

Zhenyu Zhang
State Key Laboratory of Computer Science
Institute of Software,
Chinese Academy of Sciences
Beijing, China
zhangzy@ios.ac.cn

***Abstract*—There are many research works on automated GUI traversal-based test case generation techniques for Android application. However, the effect of different factors used in a GUI traversal algorithm has not been systematically explored. In this work, we report a controlled experiment on 33 real-world applications to expose their real failures to systematically study three major factors that are commonly observed in testing tools for this class of applications. They include the notion of GUI state equivalence, the state search (or exploration) strategy, and the amount of time to wait between two input events. Our experimental results clearly show that different notions of GUI state equivalences have significantly different effects on failure detection rate and code coverage, randomized search is comparable to systematic search, and different choices of waiting time strategies do not make significant differences in terms of testing effectiveness. We also report other interesting results in this paper.**

Keywords—Android applications; GUI traversal; test case generation; Automatic testing

I. INTRODUCTION

Android is a prominent kind of mobile operating system [28]. As of early 2017, the number of Android applications (excluding those low-quality ones) in Google Play alone has reached 2.4 million [30]. Many of these applications have been extensively used in our digital living environments. In view of the keen competitions among Android applications offering the same kind of functions, developers are increasingly aware of placing a high priority in assuring the quality of their Android applications because applications that crash and/or malfunction frequently are unlikely to be welcomed by end-users. Improving the quality of Android applications as a whole thus leads to an improvement of the quality of their digital life.

Program testing is one of the most widely practiced approaches to assure the correctness of applications in software development projects. Owing to the presence of a large number of Android applications to be developed and released, automated test case generation techniques [3][5][11][12][15][18] are one of the major research focuses in the software engineering research, which not only targets at generating test inputs but also exposes failures efficiently. These techniques can be broadly classified into a few categories, including fuzzers (e.g., monkey and IntentFuzzer [26]), GUI modeling and traversal-based techniques [1][5][11][18], search-based techniques [24] and symbolic execution techniques [2]. Among these classes of techniques, techniques based on traversal of GUI models have been regarded as one of the most promising directions [25]. For ease of our presentation, we refer this class of techniques to as StateTraversal.

Typical StateTraversal techniques follow the following workflow: A StateTraversal technique starts from a given initial GUI state of the Android application under test. To support this process of GUI state identification, it defines what constitutes a GUI state and an objective criterion to determine whether two GUI states are equivalent. Among all operable and unexplored widgets of the current GUI states reached by the technique, it selects one of them, and sends input events to the widget to explore the GUI state space of the application under test. It then waits for a while before extracting the current GUI state and sending the next event. The above procedure repeats until all the operable widgets of all reached GUI states have been explored.

In the above workflow, there are multiple (and major) configurable parameters that each StateTraversal technique can choose to initialize. Since this class of techniques is a state exploration technique, the notion of state and state equivalence are fundamental.

† Correspondence Author
* This research is supported in part by the Key Research Fund of the MIIT of China (project no. MJ-Y-2012-07), the Research Grants Council of HKSAR (project nos. 11201114, 11200015, 11214116), an open project from the State Key Laboratory of Computer Science (project no. SYSKF1608), and the National Natural Science Foundation of China (project no. 61379045).

978-1-5386-0592-9/17 $31.00 © 2017 IEEE
DOI 10.1109/QRS.2017.12

Once the notion of state and state equivalence are decided, the next important design of such a technique is to decide the search strategy to be used for exploration the state space based on the encountered GUI states.

Also, the time to wait between two input events is also an interesting design factor. For instance, one popular intuition is that the next event should be sent after the GUI state has become stable. Is there really a significant difference to test effectiveness if not following this intuition in tool design such as by waiting just for a certain amount of time between two events?

In this paper, to the best of our knowledge, we present the first work in reporting a large-scale controlled experiment that systematically studies the above important parameters configurable for StateTraversal techniques using 33 real-world Android applications to expose real and critical failures (e.g., crash). We have studied the effects on failure detection ability of different factor levels of the following three factors: GUI state equivalence criterion, search strategy over the GUI state transition graph, and the waiting time strategy described above. Our controlled experiment has chosen to use a number of representative factor levels of each of these factors to gain significance and relevance to the industry and research rigor. Specifically, we have chosen to use identical Activity ID of a widget, identical UI hierarchy of a GUI state, and similar GUI hierarchy and attributes (in the sense of cosine similarity) as the factor levels for the state equivalence criterion, use breadth first search, depth first search, and randomized search as the factor levels for the search strategy, and use wait-for-idle and wait-for-time-period (with more than 10 fold difference in timing period among concrete time period used) as the factor levels of the waiting time factor. To support the data analysis, we have implemented all combinations of these factor levels in our full factorial design of the experiment, and executed each of these applications for 3600 seconds for each such combination. In total, we have executed these 33 applications for 1188 hours.

Our controlled experiment revealed interesting results. First, we found that using Cosine similarity as the notion of state equivalence resulted in higher failure detection ability and statement coverage achieved by the corresponding test executions both in a statistically meaningful way at the 5% significance level. Second, interestingly, the randomized search strategy was statistically comparable to other systematic exploration strategies at the 5% significance level in both failure detection rate and statement code coverage. Third, the strategy of waiting for all activities idle before sending the next input event to the application under test was not statistically more effective than the strategy of waiting for a fixed time interval at the 5% significance level in terms of both failure detection rate and statement code coverage. Moreover, by fixing the factor level of each factor one at a time, we also identified many combinations of two factors (at the factor levels) resulted in highest failure detection rates and statement code coverage.

The contribution of this paper is threefold: (1) To the best of our knowledge, this paper reports the *first* experimental study that investigates the impact of different levels and treatments of StateTraversal techniques on test effectiveness systematically. (2) It reveals that the notion of state equivalence is a *significant design factor* of StateTraversal technique. Moreover, randomized search is surprisingly effective compared to systematic search. It also clarifies the popular misunderstanding that waiting until GUI state becomes stable before sending the next event is crucial for effective test case generation of Android applications. (3) It shows that there are many combinations of factor levels can attain the same high level of failure detection rate and high statement code coverage in the experiment, indicating that there could be many good configurations in configuring StateTraversal. It points to the research direction of studying the Pareto efficiency in test case generation for this class of techniques.

The organization of the rest of the paper is as follows. In Section II, we introduce the generic GUI traversal-based test case generation framework. In Section III, we present the details of each design factors studied in our controlled study. In Section IV, we present our controlled experiment as well as the results analysis. Then we describe the related work in Section V. Finally, we conclude our work in Section VI.

II. A GUI Traversal-Based Test Case Generation Framework

A. Overview of the PUMA Framework

PUMA [11] is an extensible framework for dynamic analysis and GUI traversal-based test case generation. Both its dynamic analysis component and its component for exploration of a UI transition model can be customized.

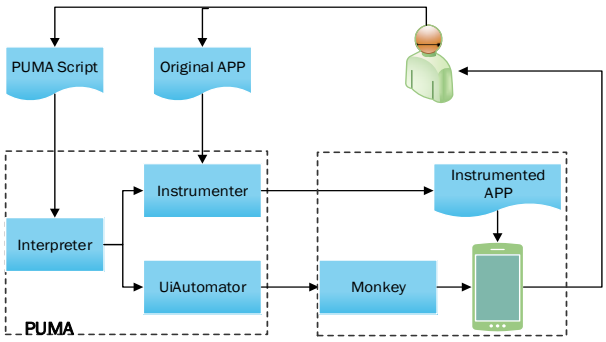

Fig. 1. Overview of PUMA

Fig. 1 shows the overview of the PUMA workflow. Developers should firstly provide a PUMAScript code and the binary code of an Android application to PUMA where PUMAScript is a language implemented as a Java extension. Next, the PUMA interpreter interprets the given PUMAScript code, and translates the code instructions into monkey-specific directives (via UIAutomator) and app-specific directives. PUMA's app instrumenter statically analyzes the application to determine the parts of the code relevant to analysis and instruments the application. The output is an instrumented version of the given application that satisfies the app-specific directives specified through the given PUMAScript code. Finally, a programmable monkey configured with the monkey-specific directives specified in the PUMAScript code executes the instrumented version of the application. Upon the

completion of the program execution, PUMA generates logs which contain outputs specified in the app-specific directives, as well outputs generated by the programmable monkey.

B. The Generic GUI Exploration-based Test Case Generation Framework

TABLE I presents the pseudo-code of the PUMA. The underlined part is the configuration points (i.e., parameters stated in Section I), which can be extended in the PUMA framework.

In the algorithm, *s* represents a GUI state and *S* represents the set of GUI states. (We note that in Section III, we will present the notion of GUI state.) Each state is associated with a set of clickable UI elements. If there is any clickable UI element not yet receiving an input (click) event, the state is called *unfinished*, otherwise, *finished*.

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

```
1: while not all apps have been explored do
2:    pick a new app and start the app
3:    S←empty stack
4:    push initial page to S
5:    while S is not empty do
6:      pop an unfinished page s_i from S
7:      go to page s_i
8:      pick next clickable UI element from s_i
        // Factor 2: Search strategy
9:      perform the click
10:     wait for next page s_j to load
        // Factor 3: Waiting time
11:     flag←s_j is equivalent to an explored page
        // Factor 1: State equivalence
12:     if not flag then
13:       add s_j to S
14:       update finished clicks for s_i
15:   if all clicks in s_i are explored then
16:     remove s_i from S
17:   if S is empty then
18:     terminate this app
```

The algorithm firstly selects an application from the application set under test and starts the application. Then, it puts the initial page of the application into the GUI state set, which is empty initially. Next, it selects an unfinished state from the GUI state set, picks a clickable UI element, and clicks on it. Third, it waits for a certain period of time so that next UI page can be loaded, then compares the new state with those explored states one by one to determine whether the new state is equivalent to an explored state. If there is no match, the algorithm puts this new state into the GUI state set. If all the clickable UI elements have been explored by clicking on them, the finished state is removed from the GUI state set. The above procedure then repeats until the state set is empty.

The code lines (lines 8, 10, and 11) with underlined comments in the algorithm are the locations of three major factors to be studied in our controlled experiment. In the next section, we describe our design of the factor levels of these design factors at these three configuration points.

III. DESIGN FACTORS

In this section, we present the three design factors to be studied in our controlled experiment, and the factor levels therein.

A. Characterization of GUI State and State Equivalence

The first design factor to be studied is how to characterize a GUI state and how to consider two GUI states to be equivalent.

We aim to explore the factor levels that have been proposed separately in different previous work. The main purpose is to critically examine whether there is any significant difference in test effectiveness, which, to the best of our knowledge, the present work is the first one to report it.

Specifically, three state equivalence criteria chosen in our controlled experiment as three factor levels of State Equivalence are as follows: the cosine similarity used by the DECAF [17] and PUMA [11], the UI hierarchy used by SwiftHand [5], and ActivityID used in A3E [3].

Factor level *Cosine*: In DECAF [17], a feature vector is used to represent a UI hierarchy. This feature vector extracts the type, the level in the DOM tree and the text from each visible UI element in the DOM tree of the UI hierarchy. For instance, a button can be expressed as (Button@2, "red", "Dial") in the feature vector, which represents that the UI element is a red button with text "Dial" at the level 2 of the DOM tree of the UI hierarchy. A state is a set of UI hierarchies that every pair of UI hierarchies in the same state are similar to one another based on the cosine similarity coefficient with a default threshold (0.95) used by PUMA. In this paper, the cosine similarity is expressed by the eigenvectors of the UI widgets. We also adopt the same default threshold in our controlled experiment.

Factor level *UI Hierarchy*: In this factor level, each GUI widget in a UI Hierarchy is mapped to the GUI type of that GUI widget and the same structure of the UI Hierarchy is maintained to connect these GUI types. For example, this factor level represents a button at the level 2 in the DOM tree of the UI Hierarchy as (Button@2). In our controlled experiment, we use the Widgets tree structure to represent the UI hierarchy, and use the following criterion to determine state equivalence: two GUI states are equivalent if and only if the widgets trees are the same.

Factor level *ActivityID*: In this factor level, if the activity identifiers of two sets of activities are the same, then the two sets of activities refer to the same GUI state. ActivityID is the only and intrinsic identity of each activity. It is coarser in granularity than the cosine similarity and UI hierarchy. ActivityID can be obtained by calling UIAutomator's API getCurrentActivityName(). Then we compare the two states' ActivityID using string comparison.

B. Search Strategy

Search strategy is the second design factor to be studied in our controlled experiment. Based on the root widget of a GUI state, we can get the set of clickable widgets reachable from

this root widget. The different orders of clicking these elements affect the traversal path in the GUI model. We propose to study three basic search strategies in the experiment: *Breadth First Search* (BFS) [11], *Depth First Search* (DFS) [3], and *Randomized Search* (Random for short) [36].

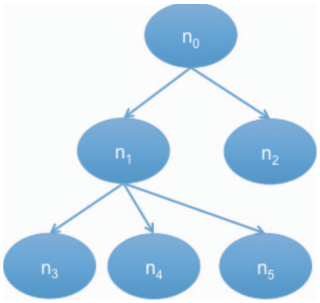

Fig. 2. A Sample GUI Widget Tree

The BFS algorithm is shown in TABLE II. In our controlled experiment, we get all the clickable widgets in the form of widgets tree and transform the tree into a queue. Firstly, it en-queues the root node in an empty queue. Next, it de-queues the first widget and puts the widget into the list *ret*, then puts the children of this widget into the queue. The loop continues until the queue is empty.

TABLE II SEARCH STRATEGY-BFS ALGORITHM

```
1:  get root clickable UI element in current app state
2:  Q←empty queue
3:  ret←empty list as clickable UI element list
4:  put root into Q
5:  while Q is not empty do
6:    qto←queue Q's head element
7:    take qto, add it to clickable list ret
8:    put the children clickable UI elements of qto into Q
9:  end-while
10: return current clickable list ret
```

The DFS algorithm is shown in TABLE III. In this paper, we get all the clickable events in the form of widgets tree and transform the tree into a stack. As shown in TABLE III, the algorithm is similar to that presented in TABLE II, except that it uses a stack instead of a queue for implementation. Fig. 2 shows a sample GUI widget tree where each node represents a clickable widget and each edge represents their parent-child relationship. For DFS traversal, n_0 is put into the stack firstly, and then n_0 is clicked and popped out from the stack. Next, n_1 is put into the stack. Then n_1 is clicked and taken out from the stack, then n_3, n_4 and n_5 are put into the stack in turn and clicked one by one. Finally, n_2 is put into the stack and clicked.

Both BFS and DFS get the next element from the ordered clickable **list** returned in each step. On the other hand, the randomized strategy just randomly chooses one widget *w* from the clickable widget **set**, and puts the children widget of *w* in the clickable widget set, then clicks on *w*, and so on.

TABLE III NEXT CLICK STRATEGY-DFS ALGORITHM

```
1:  get root clickable UI element in current app state
2:  S←empty stack
3:  ret←empty list as clickable UI element list
4:  push root into S
5:  while S is not empty do
6:    sto←the top element of stack S
7:    pop sto, add it to clickable list ret
8:    push the children clickable UI elements of sto into S
9:  end-while
10: return current clickable list ret
```

C. Waiting Time

The waiting time is the third factor to be studied in our controlled experiment. It determines the time for a testing tool to wait for the next GUI state to finish loading after event. Different waiting times may lead to different GUI states sampled from the program execution. An intuition is that a testing tool should wait until the next GUI state is completely rendered, or the time period should be long enough to ensure the next GUI state is ready to accept next event.

In our controlled experiment, we choose four different factor levels to evaluate the above intuition.

PUMA [11] uses a waiting time policy called waitForIdle() invoked through its UIAutomator API, which waits for the current program execution of the application to become idle. This API call ensures that a full GUI state is loaded.

Apart from the synchronization approach taken by PUMA, another popular strategy is to use timing control. One intuition stated above is that a longer waiting time seems to be more desirable, therefore, we consider multiple waiting time periods, which differs by 10 folds as a whole. Shauvik et al. set the delay time of 200ms in the use of Monkey tools for Android application stress test [25]. ACTEve [2] keeps the waiting time for the next state to be 3000ms, whereas, SwiftHand [5] sets it to 5000ms. We denote these three factor levels as wait200ms, wait3000ms, and wait5000ms, respectively.

D. Summary of Factors and Factor Levels

TABLE IV THREE FACTORS AND THEIR LEVLES

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

TABLE IV summarizes the factor levels for each factor studied in the experiment. The three levels for the factor state equivalence are cosine-similarity metric, UI hierarchy and ActivityID. The three levels for factor search strategy are BFS, DFS and Random. The four levels for waiting time are waitForIdle, wait 200ms, wait 3000ms and wait 5000ms.

TABLE V LIST OF APPS USED IN OUR STUDY

	Android apps	Version	Category	Previously Used by
1	BookCatalogue	1.6	Utility	A3E
2	TomdroidNotes	2.0a	Social	A3E
3	Wordpress	0.5.0	Productiv	A3E
4	SpriteMethodTest	-	Sample	ACTEve
5	RandomMusicPlayer	1	Music	ACTEve
6	CountdownTimer	1.1.0	Utility	ACTEve
7	Ringdroid	2.6	Media	ACTEve
8	Translate	3.8	Utility	ACTEve
9	Nectroid	1.2.4	Media	DynoDroid
10	MunchLife	1.4.2	Entertain	DynoDroid
11	Addi	1.98	Utility	DynoDroid
12	Photostream	1.1	Media	DynoDroid
13	SyncMyPix	0.15	Media	DynoDroid
14	aLogCat	2.6.1	Tools	DynoDroid
15	Multi SMS	2.3	Comm.	DynoDroid
16	BaterryDog	0.1.1	Utility	DynoDroid
17	NetCounter	0.1.4	Utility	DynoDroid
18	DivideAndConquer	1.4	Casual	DynoDroid
19	HotDeath	1.0.7	Card	DynoDroid
20	Bomber	1 1	Casua	DynoDroid
21	Auto Answer	1.5	Utility	DynoDroid
22	PasswordMakerPro	1.1.7	Utility	DynoDroid
23	K-9 Mail	3.512	Comm.	DynoDroid
24	AardDictionary	1.4.1	Reference	DynoDroid
25	LearnMusicNotes	1.2	Puzzle	SwiftHand
26	MiniNoteViewer	0.4	Utility	SwiftHand
27	TippyTipper	1.1.3	Finance	SwiftHand
28	WeightChart	1.0.4	Health	SwiftHand
29	Sanity	2.11	Comm.	SwiftHand
30	Mileage	3.1.1	Finance	SwiftHand
31	MyExpenses	1.6.0	Finance	SwiftHand
32	Whohasmystuff	1.0.7	Utility	SwiftHand
33	DalvikExplorer	3.4	Utility	SwiftHand

IV. CONTROLLED EXPERIMENT

In this section, we describe our controlled experiment and present the evaluation results.

A. Research Questions

RQ1: Does choosing different notions of state equivalence have significant impact on the test effectiveness in terms of failure detection rate and code coverage?

RQ2: Is choosing a systematic search strategy (DFS or BFS) superior to choosing the randomized strategy (Random) in terms of failure detection rate and code coverage?

RQ3: Is there any significant difference between Wait-for-Idle and Wait-for-a-While strategies in the factor of Waiting Time? Moreover, within the group of Wait-for-a-While strategies, is it true that a longer waiting time leads to a higher failure detection rate or higher code coverage?

RQ4: Is there any particularly effective treatment observed in the controlled experiment in terms of failure detection rate and code coverage?

B. Benchmarks

We selected 33 real-world open-source mobile apps used by four previous tool projects as our benchmark suite. Sixteen (16) of these apps were taken from Dynodroid[18], 3 from A3E[3], 5 from ACTEve[2], and 9 from SwiftHand[5]. TABLE V lists the benchmarks with version number, application category, and other tools that have previously evaluated them.

C. Experimental Setup

To evaluate the effects of the factor levels and their applicable combinations (known as *treatments*) of these three factors, we implemented all above-mentioned factor levels in the PUMA framework. Then we set up the PUMA framework to test the 33 Android applications.

The controlled experiment was carried out on two virtual machines installed with Ubuntu 14.04 operating systems. We used the open source virtualization software named Oracle VirtualBox. The Oracle VirtualBox can install multiple client operating systems and each client system can be opened, suspended and stopped independently. Each virtual machine was configured with dual-core processor and 6GB memory. We selected Vagrant to build virtual testing environment and manage these virtual machines.

D. Experimental Procedure

There were in total 36 (*i.e., 3*3*4*) combinations of factor levels for the three factors (state equivalence, search strategy and waiting time). Each combination is a configuration in the PUMA framework. Therefore, we ran the 33 benchmarks with the PUMA tool under each of the 36 configurations for 1 hour each. Then, we collected the code coverage and failure information at the end of each execution. The whole process took 1188 testing hours in total on our 2 virtual machines.

For RQ1 to RQ3, we aim to study and compare the effects of different levels of each factor on test effectiveness (failure detection rate and code coverage). When we analyzed the data for the levels of one factor, we aggregated the results for all the levels of the other two factors. For example, when we studied the effects of different levels for the factor *state equivalence*, we grouped all the results with the same level together and compared their populations statistically.

For RQ4, we aim to study the impact of different treatments (i.e., combination of levels) on testing effectiveness. When we studied the impact of one treatment, we grouped the results for other factor levels of the same factor together and compared their statistical populations among these groups. For example, when we compared different treatments having two factor levels fixed to <*state equivalence*, *search strategy*>, we grouped all results of *Waiting Time* for each treatment and then compared their populations.

For all research questions, we conducted the one-way ANalyses Of VAriances (ANOVAs) (e.g., also used in [16]) to compare the distributions of groups of data to check whether their means differed significantly from one other. We then used the multiple comparison procedures to perform pair-wised comparison.

In this experiment, failure detection rate and statement code coverage were chosen as the two metrics to evaluate the test effectiveness. These two metrics are also used in previous testing research to evaluate test effectiveness (e.g., [25]). We did not use any seeded faults in the experiment. In other words,

all the failures detected are also real failures in real-world applications.

To collect the code coverage of each application under each treatment, we used the Emma [31] to generate coverage reports and collect the line coverage. To gather the failure information of each program execution, we collected the system *logcat* file of each application and extracted the exceptions and errors with stack trace information. We wrote a script to parse exceptions and errors triggered in the testing process. Because some failures occurred repeatedly during the testing process, we considered two failures as the same if they had the same stack traces and produced the same kind of error messages upon failure. In this way, we were able to measure the number of distinct failures within a 1-hour testing to evaluate the failure detection rate.

E. Results and Analysis

In this section, we present our data analysis to answer each research question stated in Section V.A.

1) Ansering RQ1

Fig. 3 shows the box-whisker plot representing the distribution of number of distinct failures detected within the 1-hour testing period for each notion of state equivalence. We can see that the notches of *Cosine*, UI Hierarchy and ActivityID do not overlap with each other. Moreover, *Cosine* achieves a higher median number of failures than UI Hierarchy, which in turn is higher than ActivityID. This shows the median values of the 3 definitions of state equivalence differ significantly from each other.

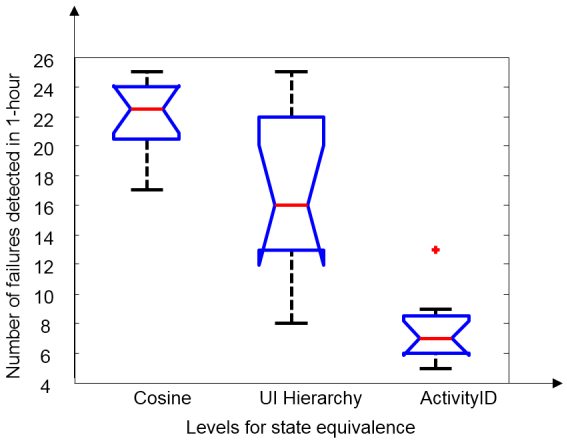

Fig. 3. Comparison of three definitions of state equivalence for failure detection rate.

We further perform the multiple comparisons to see whether the means of different notions of *state equivalence* differ significantly from each other at the 5% significance level. The result is shown in Fig. 4, in which we can see that there is no overlap between the three state equivalence definitions. This result confirms that *Cosine* is significantly more effective than *UI Hierarchy*, which is in turn more effective than *ActivityID* in detecting failures.

We further check the test cases generated from different levels of the factor State Equivalence. We found that the Cosine was a finer notion of state equivalence, which in turn made the GUI state model more fine-grained. As a result, within the same testing period, the PUMA tool with Cosine as the configuration parameter may have a higher chance to visit more transitions between distinct state pairs. Similarly, the UI Hierarchy is coarser than Cosine distance but finer than ActivityID. Therefore, adopting UI Hierarchy is more effective than ActivityID but less effective than Cosine.

The result seems indicating that the ability to explore more dynamic states is a superior design option in engineering a test case generation tool. Nonetheless, a question is whether the same observation can be made using typical code coverage as a measure.

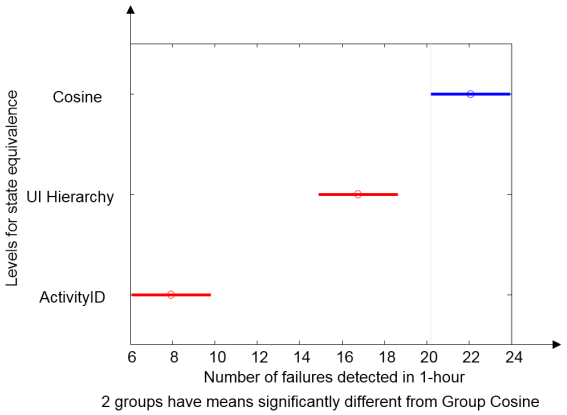

Fig. 4. Multiple comparison results on three notions of state equivalence for failure detection rate.

Fig. 5 shows that the distributions of the three notions of state equivalences in terms of statement code coverage. We find again that the notches of different state equivalence definitions do not overlap with each other, which shows their median values differ significantly from each other. Furthermore, the multiple comparison results of Fig. 6 show that, *Cosine* achieves a higher code coverage rate than UI Hierarchy, which in turn performs better than *ActivityID* at a 5% significance level.

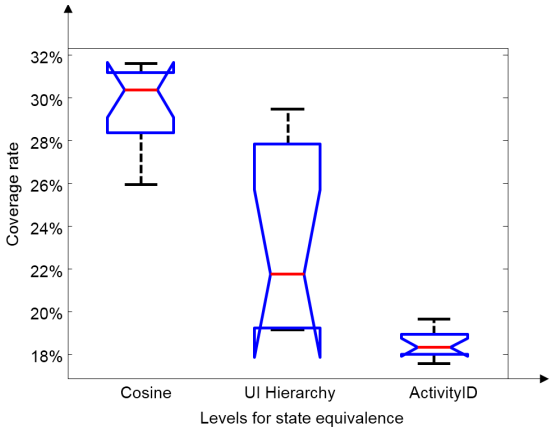

Fig. 5. Comparison of three notions of state equivalence for code coverage

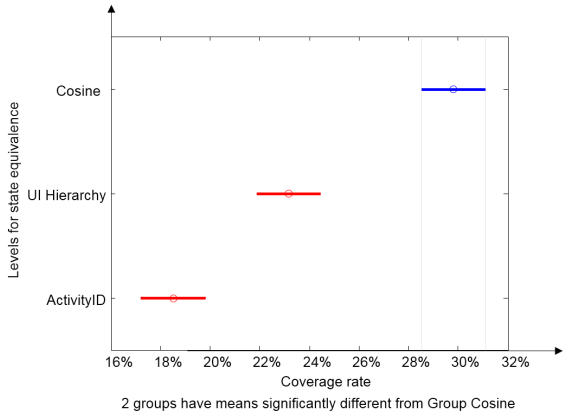

Fig. 6. Multiple comparison results on three notions of state equivalence for code coverage

Combining the two aspects (failure detection rate and code coverage), it appears to us that there is a significant difference in the testing effect when using different notions of state equivalence in the test case generation process. It seems the finer the state equivalence definition, the better the failure detection and code coverage results.

Finding 1: Using a finer notion of state equivalence (Cosine Similarity > UI Hierarchy > ActivityID) resulted in higher failure detection ability and code coverage rate as achieved by the corresponding test executions in a statistically meaningful way at the 5% significance level.

2) Ansering RQ2

The result on the failure detection rates for the factor *Search Strategy* is shown in Fig. 7 and the corresponding multiple mean comparison result is shown in Fig. 8. We can see from Fig. 7 the median values of different search strategy do not differ significantly from each other. From Fig. 8, we find that the difference in their mean values is also not significant at the 5% significance level. This result is interesting. Random requires less design effort than BFS and DFS, but it can achieve comparable results in detecting failures.

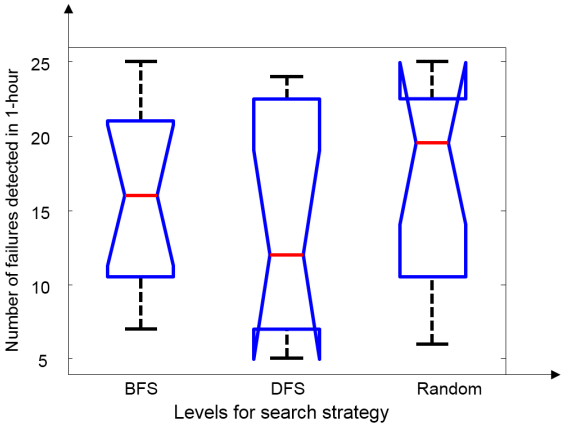

Fig. 7. Comparison of three search strategies for failure detection rate.

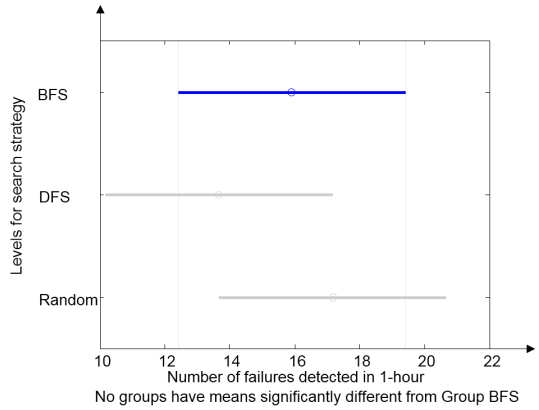

Fig. 8. Multiple comparison results on three search strategies for failure detection rate.

Indeed, from the code coverage perspective, Fig. 9 shows that BFS, DFS and Random all achieve similar code coverage, which is further confirmed by the multiple comparisons shown in Fig. 10.

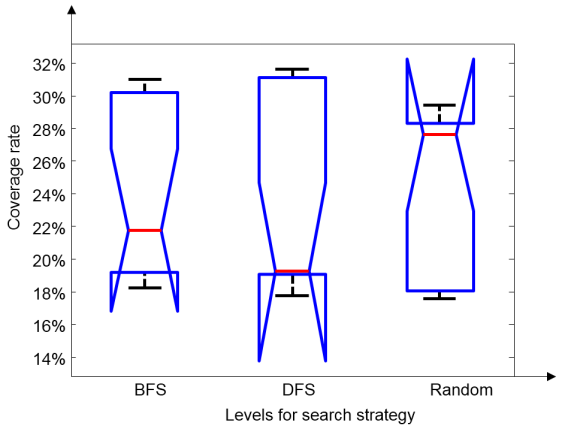

Fig. 9. Comparison of the three search strategies for code coverage.

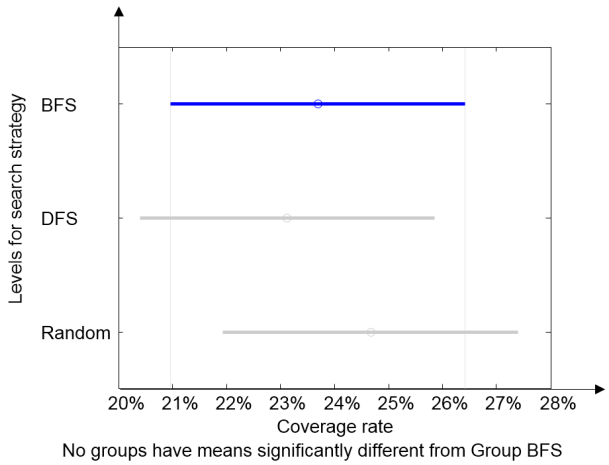

Fig. 10. Multiple comparison results on three search strategies for code coverage.

The overall results show that there is no significant difference between the systematic and randomized search strategies. The result also indicates that Random is a surprisingly good search strategy for test case generation for Android application due to its simplicity.

Finding 2: Interestingly, the randomized search strategy was statistically comparable to other systematic exploration strategies at the 5% significance level in both failure detection rate and statement code coverage. It indicates that the additional time overhead incurred by a systematic search strategy (BFS and DFS) is not paid off.

3) Ansering RQ3

For the factor *Waiting Time*, in terms of failure detection rate, we can see from Fig. 11 that the median values of using different waiting time do not differ significantly from each other.

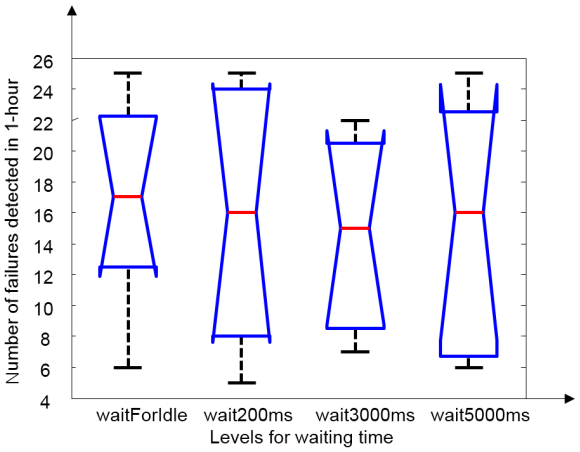

Fig. 11. Comparison of the waiting time levels for failure detection rate.

The result for the multiple comparisons in Fig. 12 further shows that the distributions of the mean values of these four strategies do not differ in a statistical meaningful way.

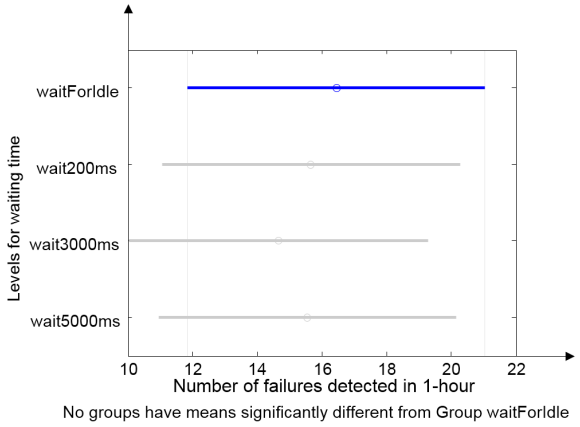

Fig. 12. Multiple comparison results on the four waiting time strategies for failure detection rate.

In terms of code coverage, we observe from their boxplots shown in Fig. 13 and the multiple mean comparisons shown in Fig. 14 that they are almost the same statistically. It indicates that waiting time is not an important design factor for code coverage. Thus, testers of Android applications can safely set this factor as a low priority in making decisions in configuring a test case generation tool for improved code coverage.

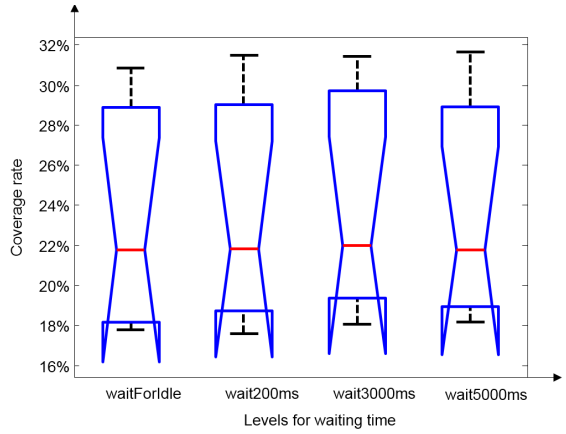

Fig. 13. Comparison of the four waiting time strategies for code coverage.

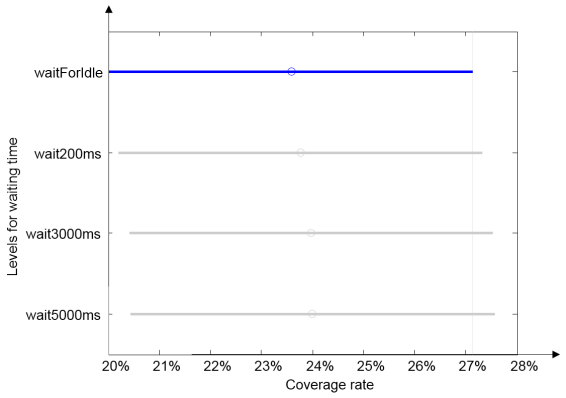

Fig. 14. Multiple comparison results on four waiting time strategies for code coverage.

Finding 3: The strategy to wait until GUI state is stable before sending the next input event is not statistically more effective than the strategy of waiting for a fixed time interval at the 5% significance level in terms of both failure detection rate and statement code coverage.

4) Ansering RQ4

In this research question, we aim to explore whether there are particularly effective combinations of design factor levels. For completeness purpose, we show the best treatment in terms of failure detection rate when none or one factor level is fixed in TABLE VI.

Specifically, we encode each treatment as follows: we use the triple (*i, j, k*) to represent a specific treatment, where *i* represents levels for *state equivalence*, *j* represents levels for

search strategy; and *k* represents levels for *waiting time*, where the levels of *i, j,* and *k* can be found in TABLE IV.

From TABLE VI, when none of the factor levels is fixed, treatment *(0,0,3)* is the best. In other words, *<Cosine Similarity, BFS, wait5000ms>* is the best configuration among all combinations of levels in terms of failure detection rate in our experiment. It has detected 25 failures in the whole experiment.

For the row indicated with a fixed factor level (e.g., Cosine in the first column), the second column shows the best treatment observed when the factor level shown in the first column is used. We can see that for 7 out of 10 studied factor levels, we can find a treatment that results in the highest number of detected distinct failures.

TABLE VI BEST TREATMENT IN FAILURE DETECTION RARE

Fixed level	Treatment and Metric	
	Best Strategy	*# of Detected Failures*
None is fixed	(0,0,3)	25
Cosine	(0,0,3)	25
UI Hierarchy	(1,2,0)	25
ActivityID	(2,0,0)	13
BFS	(0,0,3)	25
DFS	(0,1,3)	24
Random	(1,2,0)	25
waitForIdle	(0,0,0)	25
wait200ms	(0,0,1)	25
wait3000ms	(0,1,2)	22
wait5000ms	(0,0,3)	25

The best treatment in terms of code coverage when none or one factor level is fixed is shown in TABLE VII. Note that we use the same triple (*i,j,k*) as above to encode each treatment.

TABLE VII BEST TREATMENT IN CODE COVERAGE

Fixed dimension	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)	19.67	15.82
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

When none of the factor level is fixed, treatment *(0,1,3)* is the best (i.e., *<Cosine Similarity, DFS, wait5000ms>*). Its mean statement coverage is 31.64%. When one dimension is fixed, the best strategy for code coverage can be interpreted similarly in other rows.

Finding 4: There were many combinations of factor levels can attain the same high level of failure detection rate and high level of statement code coverage in the experiment. It indicates that there could be many good configurations in configuring StateTraversal. The findings point to the research direction of more comprhensive study in the Pareto efficiency of test case generation for the techniques.

F. Threats to Validity

The first factor affecting the threat to validity is the correctness of our tools. We implemented those factor levels within the PUMA tool and the virtualized experiment framework was adapted from [25]. To reduce the threats due to bugs in our implementation, we had carefully examined our source code and repeated the experiment results of [25] for double checks.

We used 33 subjects studied and evaluated in previous work. An experimental study on other subjects may result in different results.

There may be other different factors affecting the effectiveness of GUI exploration-based test case generation techniques. And there may be different factor levels for the factors studied in this work. It is interesting to extend our work to take those factors and levels into consideration in the future.

Similar to previous work [13][29][48], we used code coverage and failure detection rate to evaluate different factor levels and treatments. An experimental study on other metrics such as the time to the first failure may show different results.

V. RELATED WORK

In this section, we will briefly review some closely related work.

A. Test Development Platform for Android Application

As Glenford J.Myers [22] described it, software testing is the process of executing a program for the purpose of discovering errors. Automated testing is the process of controlling the execution of tests using special software and comparing the actual results to the expected results [35]. At present, there are several test development platforms for Android application, such as MonkeyRunner [33], Robotium [34], and UIAutomator [29].

MonkeyRunner [33] is a testing tool provided by Android SDK. The user uses the testing API interfaces provided by the tool to write Python scripts. The script is sent to MonkeyRunner as a test case for execution. In addition, It also has a screen capture and image comparison mechanism, which can serve as test oracles. Robotium [34] is another popular automated application testing tool. It is based on Android's instrumentation framework, supporting the black box automatic testing. UIAutomator [29] is Google's new automated testing tool for android applications. It requires the users to create a test project for the application under test. Users can write test scripts to simulate UI events such as tap, drag, and text input.

B. Test Case Generation Technique for Android

Hu et al. [13] classified android application errors into activity error, event error, dynamic type error, API error, I/O error and concurrency error. They proposed an event-based testing tool for Android application. Starting with the source code of the application, they used the Java test case generation tool JUnit to generate user test case. For each test case, they used the automated event generation tool Monkey to add some events to simulate the user interaction. In the implementation of test cases, the system log file records the application details.

When the testing has finished, the tool will analyze the log files for potential errors.

Model based testing technology [9] must first construct the model of the android applications under test. Then test case generation algorithm will traverse the model in a systematic manner to generate test cases.

Amalfitano et al. [1] proposed a crawler-based testing technique for Android application. This technique first uses crawler-based technique to generate the GUI model of the application under test, then it further generates event sequences based on the generated model. Hao et al. [11] presented a tool named PUMA, which is a generic dynamic analysis framework and test case generation tool for Android application. The strength of PUMA is not in its strategy of exploring apps, but in its generic design. PUMA is a framework that can be extended according to tester's needs. Users can easily perform all kinds of dynamic analysis by extending the basic exploration strategy, such as accessibility violation detection and Ad fraud detection.

Choi et al. [5] implemented a tool named SwiftHand. The tool aims to maximize the code coverage of app under test. It learns a dynamic finite state model while testing the app. And it uses the learned model to generate user inputs. Moreover, it can further improve the learned model during the GUI model exploration process. A key feature of the algorithm is to minimize the number of restarts during exploration, which can save a lot of testing time.

A3E [3] is uses two different and complementary strategies to implement the exploration. The first strategy is *A3E-Depth-First*, which implements a depth first search on the dynamic model of the app. The dynamic model abstracts each activity into a single state, without considering the different states the widgets of the activity. This approach could lead to more efficient exploration of the behavior of an activity. The second strategy is *A3E-Targeted*, which can construct a static activity transition graph of the app under test via taint analysis. Such graph allows the tool to cover activities more efficiently by generating intents.

Monkey [36] is an application automated testing tool provided by Google. It is mainly used for stress testing and reliability testing. Monkey runs in emulator or a device and generates pseudo-random streams of user events (key input, touch screen input, gesture input, etc.) to the application under test. Due to its simplicity and applicability, it is widely adopted in industry.

Dynodroid [18] implements a random exploration strategy similar to Monkey, but it is more effective than Monkey based on the following features. First, it can also generate system events in addition to user events by examining which ones are related to the application. Second, its exploration strategy is more flexible. It can either select the events that have been least frequently selected (Frequency strategy) and can take into account the context (BiasedRandom strategy), that is, events that are relevant in more contexts will be selected more often.

Sapienz [19] is multi-objective search-based automated testing tool for Android applications. It sets several goals for its search-based generation process: higher failure detection rate, higher code coverage, and smaller test case size. Their experimental results show that the technique is competitive when compared with existing techniques.

ACTEve [2] is a concolic-testing tool that symbolically tracks events from the point in the framework where they are generated up to the point where they are handled in the app. For this reasons, ACTEve needs to instrument both the framework and the app under test. The limitation is in its scalability: it can only generate test cases with 4 touch events at most.

VI. Conclusion

GUI traversal-based test case generation techniques are promising for effective Android application testing. However, there are several configurable steps (factors) within the test case generation process, whose impact on test case generation effectiveness has not been studied systematically. In this work, we report a controlled experiment on 33 real-world applications to study 3 factors in a full-factorial design setting resulting in 36 treatments. Our experimental results show that different notions of state equivalence will significantly affect the failure detection rates and the code coverage, baseline systematic search strategies (BFS and DFS) are comparable to the randomized search strategy (Random) in both failure detection rate and code coverage, and waiting for idle and waiting for a fixed time period have no significant difference in these two metrics either. We have also observed that Cosine notion of state equivalence is statistically superior in our controlled experiment. Among all the 36 treatments, the treatment <*Cosine Similarity, BFS, wait5000ms*> is the best configuration for GUI traversal-based test case generation technique in terms of both failure detection rate. And the treatment <*Cosine Similarity, DFS, wait5000ms*> is the best configuration for code coverage. Moreover, with respect to the same factor level of each factor, there are many good treatments that achieved the highest failure detection rates and statement code coverage.

This work points to the research direction of studying the Pareto efficiency in test case generation for this class of techniques. It is interesting to find out the underlying reasons on why state equivalence in a finer granularity could improve failure detection rate but not affect the code coverage and why longer waiting time negatively correlates to the failure detection rate.

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 Software, 32(5):1-1, vol. PP, no. 99, 2014.

[2] S. Anand, M. Naik, M. J. Harrold, and H. Yang, Automated Concolic Testing of Smartphone Apps, in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE2012), NY, USA: ACM, pp. 59:1–59:11, 2012.

[3] T. Azim and I. Neamtiu, Targeted and Depth-first Exploration for Systematic Testing of Android Apps, in Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA2013), New York, NY, USA: ACM, pp. 641–660, 2013.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. P. Scout: analyzing the Android permission specification, in Proceedings of ACM Conference on Computer and Communications Security (CCS2012), pp. 217-228, 2012.

[5] W. Choi, G. Necula, and K. Sen, Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning, in Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA2013), New York, NY, USA: ACM, 2013, pp. 623–640, 2013.

[6] J. Crussell, C. Gibler, and H. Chen. Attack of the Clones: DetectingCloned Applications on Android Markets." in Proceedings of European Symposium on Research in Computer Security (ESORICS2012), 81(13):2454-2456, 2012.

[7] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: DetectingPrivacy Leaks in iOS Applications, in Proceedings of National Down Syndrome Society (NDSS2011), 2011.

[8] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones, in Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI2010), Berkeley, CA, USA, 2010, pp. 1–6, 2010.

[9] M. C. Gaudel, Testing can be formal, too, P.D. Mosses, M. Nielsen, M.I. Schwartzbach (Eds.), tapsoft '95: Theory and Practice of Software Development, Lecture Notes in Computer Science, number 915, Springer-Verlag, Heidelberg, pp. 82-96, 1995.

[10] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller, Checking app behavior against app descriptions, in Proceedings of the 36th International Conference on Software Engineering (ICSE2014), New York, NY, USA: ACM, June 2014, pp. 1025–1035, 2014.

[11] 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 Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys2014), New York, NY, USA: ACM, 2014, pp. 204–217, 2014.

[12] C. Hu and I. Neamtiu, Automating GUI Testing for Android Applications, in Proceedings of the 6th International Workshop on Automation of Software Test (AST 2011), New York, NY, USA: ACM, pp. 77–83, 2011.

[13] C. Hu and I. Neamtiu, Testing of Android Apps, in Proceedings of ACM Object-Oriented Programming, Systems, Languages & Applications (OOPSLA2013), 2013.

[14] M. Kechagia, D. Mitropoulos, and D. Spinellis, Charting the API minefield using software telemetry data, Empirical Software Engineering (ESE2014), pp. 1–46, 2014.

[15] Xiujiang Li, Yanyan Jiang, Yepang Liu, Chang Xu, Xiaoxing Ma and Jian Lu. User Guided Automation for Testing Mobile Apps. In Proceedings of the 21st Asia-Pacific Software Engineering Conference (APSEC 2014), pp. 27-34, Jeju, Korea, Dec 2014.

[16] Z. Li, M. Harman and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization, in IEEE Transactions on Software Engineering, 33(4):225-237, 2007.

[17] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. in Proceedings of the National Spatial Data Infrastructure (NSDI2014), pp. 57-70, 2014.

[18] A. Machiry, R. Tahiliani, and M. Naik, Dynodroid: An Input Generation System for Android Apps, in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013), New York, NY, USA: ACM, 2013, pp. 224–234, 2013.

[19] K. Mao, M. Harman, and Y. Jia. Sapienz: multi-objective automated testing for Android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA2016), ACM, New York, NY, USA, pp. 94-105, 2016.

[20] A. P. Mathur, Foundations of Software Testing. Pearson Education India, 2008.

[21] A. Memon, I. Banerjee, and A. Nagarajan, GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing, in Proceedings of the 10th Working Conference on Reverse Engineering (WCRE2003), Washington, DC, USA: IEEE Computer Society, 2003, pp. 260–, 2003.

[22] G. J. Myers, C. Sandler and T. Badgett, The art of software testing. John Wiley & Sons, pp. 7-8, 2011.

[23] D. Octeau, S. Jha, and P. McDaniel, Retargeting Android Applications to Java Bytecode, in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE2012), New York, NY, USA: ACM, 2012, pp. 6:1–6:11, 2012.

[24] R. Rakesh, P. Rabins and M. S. Chandra, Review of Search based Techniques in Software Testing. International Journal of Computer Applications (IJCA2012), 51(6):42-45, 2012.

[25] R. C. Shauvik, G. Alessandra and Oo. Alessandro, Automated Test Input Generation for Android: Are We There Yet? in Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering (ASE2015), Lincoln, Nebraska, USA, pp. 429-440, 2015.

[26] K. Yang, J. Zhuge, Y. Wang, L. Zhou, and H. Duan. 2014. IntentFuzzer: detecting capability leaks of android applications. In Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIA CCS2014). ACM, NY, USA, pp. 531-536, 2014.

[27] A. Zeller, R. Hildebrandt, Simplifying and isolating failure-inducing input. Software Engineering (TSE2002), IEEE Transactions on, 28(2):183-200, 2002.

[28] Android.http://www.android.com, last access on Jan, 2017.

[29] Android uiautomator. http://developer.android.com/tools/help/uiautomator/index.html.

[30] AppBrain. The number of applications in Google Play. https://www.appbrain.com/stats/number-of-android-apps

[31] Emma. http://emma.sourceforge.net/.

[32] Gartner. http://www.199it.com/archives/408226.html, last access on November28, 2015.

[33] Monkeyrunner.http://www.android-doc.com/tools/help/monkeyrunner_concepts.html.

[34] Robotium. https://github.com/RobotiumTech/robotium

[35] Software Testing Research Survey Bibliography. http://web.engr.illinois.edu/~taoxie/testingresearchsurvey.htm.

[36] The Monkey UI android testing tool, http://developer.android.com/tools/help/monkey.html.