Randomized Event Sequence Generation Strategies for Automated Testing of Android Apps

David Adamo*, Renée Bryce
University of North Texas
3940 North Elm Street
Denton, TX 76205, USA
DavidAdamo@my.unt.edu, Renee.Bryce@unt.edu

Tariq M. King
Ultimate Software Group, Inc.
2250 North Commerce Parkway
Weston, FL 33326, USA
Tariq_King@ultimatesoftware.com

***Abstract*—Mobile apps are often tested with automatically generated sequences of Graphical User Interface (GUI) events. Dynamic GUI testing algorithms construct event sequences by selecting and executing events from GUI states at runtime. The event selection strategy used in a dynamic GUI testing algorithm may directly influence the quality of the test suites it produces. Existing algorithms use a uniform probability distribution to randomly select events from each GUI state and they are often not directly applicable to mobile apps. In this paper, we develop a randomized algorithm to dynamically construct test suites with event sequences for Android apps. We develop two frequency-based event selection strategies as alternatives to uniform random event selection. Our event selection algorithms construct event sequences by dynamically altering event selection probabilities based on the prior selection frequency of events in each GUI state. We compare the frequency-based strategies to uniform random selection across nine Android apps. The results of our experiments show that the frequency-based event selection strategies tend to produce test suites that achieve better code coverage and fault detection than test suites constructed with uniform random event selection.**

***Index Terms*—GUI Testing, Automated testing, Mobile apps, Android**

I. Introduction

Mobile devices are increasingly powerful resources that enable individuals to perform computing tasks anywhere at anytime. Google's Android holds the largest share of the mobile Operating System (OS) market worldwide [8]. Mobile apps now provide services to end users in critical domains such as e-commerce, banking and healthcare. Faulty mobile apps may lead to devastating consequences. Only about 16% of users are likely to try a failing app more than twice [15]. By 2017, the mobile app market will be a $77 billion industry [6]. Therefore, the success of a mobile app may depend on how thoroughly it is tested during its development life cycle.

The services provided by a mobile app are often accessed by executing GUI events such as clicking a button or entering data into text boxes. Mobile apps are Event Driven Systems (EDSs) that are often tested with automatically generated GUI event sequences. Model-based testing techniques use an abstract model of the Application Under Test (AUT) to automatically generate event sequences. Model construction is an expensive process and model-based testing approaches may generate infeasible test cases since the model may not always accurately reflect the runtime state of the AUT. Dynamic GUI testing techniques are often based on dynamic event extraction and do not require an abstract model of the AUT [3]. Dynamic event extraction-based algorithms generate event sequences through repeated selection and execution of available events from GUI states at runtime. Existing algorithms and techniques for dynamic test suite construction focus on other types of GUI-based software (such as desktop and web applications), and are often not effective for mobile apps because they are not adapted to GUI patterns that are unique to mobile applications. Android devices have "home" and "back" navigation buttons that exhibit domain-specific behavior and are available in every GUI state of an Android app. These navigation events often close the AUT and are disproportionately more likely to be selected during dynamic test suite construction, compared to other events that are only available in specific GUI states. Algorithms that do not compensate for the ubiquity of these events may produce test suites that contain too many short event sequences that do not adequately explore the AUT's GUI. Dynamic GUI testing techniques identify potential events from GUI states at runtime and use an event selection strategy to choose events to execute. The choice of event selection strategy may influence the quality of dynamically constructed test suites. While there is a significant body of work on automated GUI testing tools and model-based techniques [1], [10], [11], [13], [16], [17], prior research gives little attention to event selection strategies for dynamic GUI testing of mobile apps. Uniform random selection with pseudorandom numbers may generate test suites in which certain GUI events are executed often, while some others are executed rarely or not at all. Thus, the test suites may fail to exercise parts of the GUI that cover significant amounts of code or expose faults.

In this work, we develop a randomized dynamic event extraction-based algorithm to automatically construct test suites for Android apps. The algorithm allows specification of fixed probability values to prevent disproportionate selection of navigation events. We also develop two frequency-based event selection strategies and empirically compare them to uniform random selection on nine Android apps. Our frequency-based event selection algorithms maintain a history of event selection frequencies and use the frequency information to dynamically alter event selection probabilities during test suite construction.

The objective is to generate test cases that are biased toward infrequently selected events. The first algorithm (frequency weighted) assigns weights to each event in a GUI state such that the weight of an event is inversely proportional to the number of times it has been previously selected. The greater the weight of an event, the more likely it is to be selected. The second algorithm (minimum frequency) always chooses an event that has been selected least frequently in a GUI state.

This work makes the following contributions: (i) we develop a randomized test suite construction algorithm for dynamic event extraction-based testing of Android apps (ii) we develop two frequency-based event selection algorithms as alternatives to uniform random event selection (iii) we empirically evaluate the frequency-based and uniform random event selection strategies on nine Android apps in terms of code coverage and fault detection. The results of our experiments show that the frequency-based strategies tend to produce test suites that achieve better code coverage and fault detection than test suites constructed with uniform random event selection.

The rest of this paper is organized as follows. Section II discusses related work on dynamic GUI testing. Section III introduces our test suite construction algorithm for Android apps. We present our frequency-based event selection strategies in Section IV and empirically evaluate them in Section V. Section VI discusses the results of our empirical study, Section VII discusses threats to validity and Section VIII concludes the paper.

II. Related Work

Bae et al. [3] perform an empirical comparison of model-based and dynamic event extraction-based GUI testing techniques. They evaluate dynamic event extraction-based testing with uniform random event selection on Java desktop applications. The study shows that a dynamic event extraction-based approach achieves better code coverage and has a lower tendency to generate nonexecutable event sequences compared to a model-based approach. Adaptive random testing (ART) may improve the effectiveness of random testing by evenly spreading test cases across the input domain. Chen et al. [5] show that ART can reduce the number of test cases needed to find the first fault by up to 50%. However, the majority of work in ART focuses only on programs with numeric inputs. Liu et al. [10] adapt ART to mobile app testing and show that an ART approach is more effective for fault detection than random testing. ART is inefficient even for trivial problems because it repeatedly calculates distances among test cases and generates multiple test case candidates, most of which are discarded [2]. Carino [4] develops algorithms to generate and execute single fixed-length event sequences for Java desktop applications. Dynodroid [11] automatically generates a single sequence of inputs for Android apps. In this paper, we develop an algorithm to generate test suites with multiple distinct event sequences of varying length for Android apps. We use the algorithm to evaluate two frequency-based event selection strategies that dynamically alter event selection probabilities based on the prior selection frequency of events in each GUI state.

III. Dynamic Event Sequence Generation

Figure 1 shows an algorithm to automatically construct test suites with GUI event sequences for Android apps. The algorithm takes the following input: (i) the application under test (ii) number of event sequences (test cases) to generate (iii) the probability of selecting the *back* event, and (iv) the probability of selecting the *home* event.

Input: application under test, AUT
Input: number of test cases, n
Input: back button probability, p_{back}
Input: home button probability, p_{home}
Output: test suite, T

```
1:  T ← φ
2:  testCaseCount ← 0
3:  while testCaseCount < n do
4:      clear app data and start AUT
5:      testCase ← φ
6:      repeat
7:          if random(0, 1) ≤ p_back then
8:              selectedEvent ← createBackEvent()
9:          else if random(0, 1) ≤ p_home then
10:             selectedEvent ← createHomeEvent()
11:         else
12:             events ← getAvailableEvents()
13:             selectedEvent ← selectEvent(events)
14:             execute selectedEvent
15:             testCase ← testCase ∪ {selectedEvent}
16:         end if
17:     until application exits
18:     if testCase ∉ T then
19:         T ← T ∪ {testCase}
20:         testCaseCount ← testCaseCount + 1
21:     end if
22: end while
```

Fig. 1. Randomized test suite construction algorithm

Android devices have "home" and "back" navigation buttons that are available in every GUI state of an Android app. The "home" event always closes the AUT regardless of its GUI state. The "back" event often closes the AUT, but its behavior in an event sequence also depends on the GUI state of the AUT and the set of states explored by preceding events. The presence of these navigation events in every GUI state makes them disproportionately more likely to be selected, compared to other events that are only available in specific GUI states. The algorithm in Figure 1 allows specification of fixed probability values for the "home" and "back" events and uses the specified value to probabilistically terminate test cases. A single test case ends when an event closes the AUT. The algorithm begins with an empty test suite and generates multiple event sequences as test cases. Before construction of each event sequence, the algorithm clears all data created by the previous session and initializes an empty event sequence. After restarting the AUT, the algorithm identifies the events that are available in the AUT's initial GUI state and uses a selection strategy to choose an event. The algorithm executes the selected event and adds it to the event sequence. Selecting and executing an event leads to a new GUI state. This event selection and execution process is repeated in each GUI state

until the algorithm selects an event that closes the AUT. The algorithm also ensures that duplicate event sequences are discarded.

The implementation of the $selectEvent$ function call on line 13 determines the strategy used to select events from each GUI state. A uniform random strategy uses a uniform probability distribution over the event sequence space to randomly select an event in each GUI state. Each event in a GUI state is equally likely to be selected and the probability distribution never changes. Uniform random selection is often implemented with pseudorandom number generators that select a random event from the set of available events in each GUI state. We discuss frequency-based alternatives to uniform random selection in section IV.

IV. Frequency-based Event Selection Strategies

A. Frequency Weighted Selection

In this strategy, event selection probabilities are determined by the number of times each event has been previously selected. Similar to uniform random selection, each event in a GUI state may be selected. Unlike uniform random selection, every event in a GUI state is *not equally likely* to be selected. In any given GUI state, events that have been previously selected fewer times relative to other events, have a higher likelihood of selection. This selection strategy dynamically changes event selection probabilities during test suite construction by keeping track of the selection frequency of each event.

Input: set of available events in GUI state, $events$
Output: selected event, $selectedEvent$
1: **function** FREQWEIGHTEDSELECTION($events$)
2: $totalWeight \leftarrow 0.0$
3: **for** $event$ in $events$ **do**
4: $totalWeight \leftarrow totalWeight$ + weight($event$)
5: **end for**
6: $selectedEvent \leftarrow$ *first event in set of events*
7: $selectionWeight \leftarrow random(0,1) \times totalWeight$
8: $weightCnt \leftarrow 0.0$
9: **for** $event$ in $events$ **do**
10: $weightCnt \leftarrow weightCnt$ + weight($event$)
11: **if** $weightCnt \geq selectionWeight$ **then**
12: $selectedEvent \leftarrow event$
13: updateSelectionCount($selectedEvent$)
14: **return** $selectedEvent$
15: **end if**
16: **end for**
17: updateSelectionCount($selectedEvent$)
18: **return** $selectedEvent$
19: **end function**

Fig. 2. Frequency weighted event selection algorithm

Figure 2 shows the frequency weighted selection algorithm. The algorithm takes the set of available events in a GUI state as input. The weight of each event in a GUI state is given by:

$$weight(e) = \frac{1}{N(e)+1} \quad (1)$$

where e is an event and $N(e)$ is the number of times the event has been previously selected. The algorithm makes a random selection biased by the weight of each available event.

B. Minimum Frequency Selection

This strategy considers only events that have been selected least frequently in a given GUI state. Unlike uniform random and frequency weighted selection, there are instances in which some events in a GUI state have no chance of selection. This strategy gives exclusive consideration to the least frequently selected events in a GUI state.

Input: set of available events in GUI state, $events$
Output: selected event, $selectedEvent$
1: **function** MINIMUMSELECTION($events$)
2: $candidates \leftarrow \phi$
3: $minCount \leftarrow \infty$
4: **for** $event$ in $events$ **do**
5: $timesSelected \leftarrow$ getSelectionCount($event$)
6: **if** $timesSelected < minCount$ **then**
7: $candidates \leftarrow \phi$
8: $candidates \leftarrow candidates \cup \{event\}$
9: $minCount \leftarrow timesSelected$
10: **else if** $timesSelected == minCount$ **then**
11: $candidates \leftarrow candidates \cup \{event\}$
12: **end if**
13: **end for**
14: $selectedEvent \leftarrow$ selectRandom($candidates$)
15: updateSelectionCount($selectedEvent$)
16: **return** $selectedEvent$
17: **end function**

Fig. 3. Minimum frequency event selection algorithm

Figure 3 shows the minimum frequency selection algorithm. The algorithm takes the set of available events in a GUI state as input. It iterates through the set of available events and identifies the subset of events that have been selected the least number of times. All events that are not in this subset are discarded. If there is more than one event that has been selected the least number of times, the algorithm makes a uniform random selection (i.e. random tie breaking). Finally, the algorithm updates the selection count of the selected event.

V. Evaluation

A. Research Questions

We conduct an empirical study to address the following research questions:
RQ1: Do the frequency-based event selection strategies generate test suites that achieve higher code coverage than those generated with uniform random event selection?
RQ2: Do the frequency-based event selection strategies generate test suites that detect more faults than those generated with uniform random event selection?

B. Subject Apps

We evaluate each event selection strategy on nine real-world Android apps. Each app is publicly available in the F-droid

TABLE I
CHARACTERISTICS OF SELECTED ANDROID APPS

App Name	Domain	Downloads	# Bytecode blocks	# SLOC
Tomdroid v0.7.2	Productivity	10,000 - 50,000	22,169	5,736
Droidshows v6.5	Entertainment	50,000 - 100,000	16,244	3,322
Loaned v1.0.2	Lifestyle	100 - 500	9,781	2,837
Budget v4.0	Finance	10,000 - 50,000	9,129	3,159
A Time Tracker v0.23	Productivity	1000 - 5000	8,351	1,980
Repay v1.6	Finance	1000 - 5000	7,124	2,059
SimpleDo v1.2.0	Productivity	100 - 500	5,355	1,259
Moneybalance v1.0	Finance	-	4,959	1,460
WhoHasMyStuff v1.0.25	Productivity	1,000 - 5000	3,597	1,026

app repository[1] and/or Google Play Store[2]. We selected each app using the following criteria: (i) the app must be GUI-based i.e. no system services (ii) the app's bytecode can be automatically instrumented to collect code coverage metrics without manually modifying its source code. Many Android apps are designed to prevent bytecode instrumentation. We used techniques described by Zhauniarovich et al. [17] to automatically instrument the bytecode of each app. Table I shows characteristics of the selected apps. The apps range from 1,026 to 5,736 source lines of code (SLOC), 3,597 to 22,169 blocks of bytecode and 1,000 to over 50,000 downloads from Google Play Store. Download metrics are only available for apps in the Google Play Store.

C. Implementation

We implemented each event selection strategy as part of a prototype tool called *Autodroid*. The tool takes packaged Android apps (APK files) as input and automatically generates test suites with event sequences. *Autodroid* collects log files for each test suite it generates. We identify faults by analyzing the log files for unhandled exceptions. The test suites, log files and coverage metadata used in this paper are publicly available online[3].

D. Experimental Setup and Design

For each subject app, we use each event selection strategy to generate 10 test suites on an Android 4.4 emulator. Each test suite contains 200 event sequences. The length of an event sequence is determined by the probability of selecting an event that terminates the AUT. We set the probability of the "back" and "home" events in each GUI state to 5% each. Existing test suites for most apps in the F-droid repository have less than 40% block coverage [9]. Our preliminary experiments with each event selection strategy show that the 5% probability value constructs test suites that achieve greater than 40% block coverage for each subject app.

The independent variable in our evaluation is the choice of event selection strategy. We evaluate the following event selection strategies: (i) *Rand* – uniform random event selection (ii) *FreqWeighted* – frequency weighted event selection and (iii) *MinFrequency* – minimum frequency event selection. To answer our research questions, we use two measures of test suite effectiveness as dependent variables: *block coverage* and *number of unique exceptions found*. A *(basic) block* is a sequence of code statements that always executes as a single unit. Block coverage is a measure of the proportion of code blocks executed by a test suite. Exceptions typically indicate faults in an Android app.

E. Results

Code Coverage: Table II shows the mean and median block coverage across ten test suites for each app and each event selection strategy. The *FreqWeighted* test suites achieve higher mean and median block coverage than uniform random selection in 6 out of 9 apps. *MinFrequency* generates test suites that achieve higher mean block coverage compared to uniform random event selection in 6 out of 9 apps. The *MinFrequency* test suites also achieve higher median block coverage in 8 out of 9 apps compared to uniform random event selection.

TABLE II
SUMMARY BLOCK COVERAGE STATISTICS

Measure	Rand	FreqWeighted	MinFrequency
Tomdroid			
Mean (%)	50.10	**51.68**	**52.11**
Median (%)	48.38	**51.25**	**48.39**
Droidshows			
Mean (%)	55.78	**56.19**	55.38
Median (%)	54.15	**54.30**	**55.80**
Loaned			
Mean (%)	60.94	58.58	**64.21**
Median (%)	58.79	57.30	**63.71**
Budget			
Mean (%)	73.42	**75.08**	**76.28**
Median (%)	72.99	**74.68**	**76.73**
A Time Tracker			
Mean (%)	67.55	**71.50**	**73.58**
Median (%)	68.35	**71.93**	**74.08**
Repay			
Mean (%)	**48.71**	47.79	48.35
Median (%)	46.16	45.98	**48.90**
SimpleDo			
Mean (%)	49.68	**51.30**	**50.67**
Median (%)	50.00	**51.56**	**52.38**
Moneybalance			
Mean (%)	**91.24**	87.59	85.43
Median (%)	**91.34**	86.88	86.14
WhoHasMyStuff			
Mean (%)	82.44	**83.39**	**83.90**
Median (%)	83.32	**83.61**	**83.95**

[1]https://f-droid.org/

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

[3]https://github.com/davidadamojr/random_strategy_suites

TABLE III
COMPARISON OF BLOCK COVERAGE MEASUREMENTS FOR RANDOM AND FREQUENCY-BASED EVENT SELECTION

Null Hypothesis	Alternate Hypothesis	p-value
$Cov(FreqWeighted) \leq Cov(Rand)$	$Cov(FreqWeighted) > Cov(Rand)$	0.18
$Cov(MinFrequency) \leq Cov(Rand)$	$Cov(MinFrequency) > Cov(Rand)$	0.03

TABLE IV
COMPARISON OF NUMBER OF UNIQUE EXCEPTIONS FOUND WITH RANDOM AND FREQUENCY-BASED EVENT SELECTION

Null Hypothesis	Alternate Hypothesis	p-value
$Faults(FreqWeighted) \leq Faults(Rand)$	$Faults(FreqWeighted) > Faults(Rand)$	0.02
$Faults(MinFrequency) \leq Faults(Rand)$	$Faults(MinFrequency) > Faults(Rand)$	0.12

To facilitate comparisons across apps, we used min-max scaling [14] to normalize the block coverage measurements for each app. We combined the rescaled data from all apps and performed a Mann-Whitney U-test [12]. The p-values in Table III indicate that *MinFrequency* shows statistically significant improvement in code coverage over uniform random event selection at the $p < 0.05$ level.

TABLE V
AVERAGE NUMBER OF UNIQUE EXCEPTIONS

Application	Rand	FreqWeighted	MinFrequency
Tomdroid	11.4	**11.5**	**12.6**
Droidshows	2.5	**3.1**	**2.7**
Loaned	0	0	0
Budget	0.3	**0.6**	**0.7**
A Time Tracker	0.1	**0.2**	0.1
Repay	0.4	**0.5**	**0.8**
SimpleDo	0.2	**0.5**	0.2
Moneybalance	**1**	0.9	0.6
WhoHasMyStuff	0.4	**0.8**	**0.6**

Fault Detection: Table V shows the average number of unique exceptions found by each event selection strategy for each of the apps. The *FreqWeighted* strategy found a higher average number of faults than uniform random selection in 7 out of 9 apps. The *MinFrequency* strategy found a higher average number of faults than uniform random event selection in 5 out of 9 apps. None of the event selection strategies generated test suites that found any faults in the *Loaned* app.

We rescaled and combined the experiment data from all apps and performed a Mann-Whitney U-test [12]. The p-values in Table IV indicate that *FreqWeighted* shows statistically significant improvement in fault detection over uniform random selection at the $p < 0.05$ level.

VI. DISCUSSION AND IMPLICATIONS

A. Potential correlations and factors affecting effectiveness

To gain further insight into the results of our experiments, we examined the test suites. Uniform random selection generates test suites that contain a lower number of unique events than those generated with the frequency-based strategies. This may be a factor in the diminished effectiveness of uniform random selection compared to the frequency-based strategies in terms of code coverage and fault detection.

One or both frequency-based strategies generated test suites that achieved better mean or median code coverage in all subject apps except *Moneybalance*. This suggests that the effectiveness of each event selection strategy may be affected by the nature of the AUT's GUI. Most of Moneybalance's functionality can only be accessed by successfully filling multiple validated text input fields. Since the frequency-based strategies take a more systematic approach to event selection, uniform random selection is more likely to repeat valid text entry sequences and may be more suitable for apps where most of the functionality can only be accessed by repeating particular events.

Frequency weighted selection generated test suites that achieved better code coverage and showed the most significant improvement in fault detection over uniform random selection. This may be attributed to a number of factors. A significant body of work in adaptive random testing (ART) shows that improvements in test case diversity lead to improvements in fault detection ability of test suites [5], [10]. To estimate test case diversity, we calculated the sum of the minimum Hamming distances [7] from each event sequence in a test suite to every other event sequence in the test suite. Frequency weighted event selection generated test suites with equal or greater test case diversity compared to test suites generated with uniform random selection. The frequency weighted test suites also had a tendency to cover a higher number of unique events compared to uniform random event selection. Frequency weighted event selection biases test suite construction toward events that have been previously selected fewer times relative to other events. The increase in test case diversity and bias toward unexplored events may be a factor in the significant improvement in fault detection observed with frequency weighted selection.

Minimum frequency event selection generated test suites that achieved the most significant improvement in block coverage over uniform random selection. However, it was less effective than frequency weighted event selection at finding faults despite achieving better code coverage. This may be due to a number of factors. Unlike uniform random and frequency weighted selection, minimum frequency selection only considers the subset of events that have been selected least frequently in each GUI state. We observed that minimum frequency event selection tends to generate test suites that cover a higher number of unique events compared to uniform random selection and frequency weighted selection. However,

the test suites also tend to have lower test case diversity than those generated with uniform random and frequency weighted selection. The consideration of only a subset of events in each GUI state may play a role in the relatively low test case diversity of test suites constructed with minimum frequency selection. The increase in the number of unique events covered with minimum frequency selection may be a factor in its tendency to achieve better code coverage than uniform random and frequency weighted selection. The decrease in test case diversity may contribute to its lower tendency to find faults compared to frequency weighted event selection.

B. Practical implications for testers

Testers need to consider the characteristics of the AUT when choosing an event selection strategy to automatically construct test suites. Uniform random selection is a simple and effective choice for small apps with GUIs that consist primarily of validated text input fields. Test suites generated with frequency weighted event selection are effective at finding faults while those generated with minimum frequency selection tend to maximize code coverage. In future work, we will study the effectiveness of our techniques on additional Android apps.

VII. Threats to Validity

The principal threat to validity of this study is the generalizability of the results as we use a limited number of subject applications. The size and complexity of the AUT may affect the results obtained with our techniques. We minimized this threat by selecting apps of different sizes and from multiple domains. The randomized nature of the event selection algorithms is also a threat to validity. To minimize this threat, we ran the algorithms 10 times for each app. Our assessment of fault finding effectiveness is limited to faults that are exposed as unhandled exceptions. The techniques presented in this work may lead to different results for other types of faults. This work is limited to GUI events and does not consider system events (e.g., changes in network connectivity) that may affect app behavior.

VIII. Conclusions and Future Work

Dynamic event extraction-based techniques can automatically generate event sequence test cases for Android apps. During test case generation, events are typically selected from the AUT's GUI with a uniform probability distribution. In this work, we develop a randomized test suite construction algorithm and evaluate two frequency-based event selection strategies as alternatives to uniform random selection. The first algorithm assigns weights to each event based on its selection frequency and then makes a frequency weighted selection. The second algorithm always chooses an event that has been selected the least number of times in a GUI state. Both algorithms dynamically alter event selection probabilities based on the prior selection frequency of events. We compared the frequency-based strategies to uniform random selection across nine Android apps. The results show that the frequency-based strategies tend to generate more effective test suites compared to uniform random selection. The **frequency weighted** strategy achieves the most significant improvement in fault detection while the **minimum frequency** strategy achieves the most significant improvement in code coverage. In future work, we will evaluate the cost of the frequency-based strategies in terms of time. We will develop more sophisticated test suite construction algorithms that consider system events such as changes in network connectivity. We will also adapt our algorithms to other mobile app platforms.

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Software*, vol. 32, no. 5, pp. 53–59, 2015.

[2] A. Arcuri and L. Briand, "Adaptive random testing: An illusion of effectiveness?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 265–275.

[3] G. Bae, G. Rothermel, and D.-H. Bae, "Comparing model-based and dynamic event-extraction based GUI testing techniques: An empirical study," *Journal of Systems and Software*, vol. 97, pp. 15–46, 2014.

[4] S. Carino, "Dynamically testing graphical user interfaces," Ph.D. dissertation, The University of Western Ontario, 2016.

[5] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[6] Entrepreneur.com, "By 2017, the app market will be a $77 billion industry (infographic)," 2014. [Online]. Available: https://www.entrepreneur.com/article/236832 (Accessed: 10-25-2016).

[7] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.

[8] IDC Research, "Smartphone OS market share, 2016 q2," 2016. [Online]. Available: http://www.idc.com/prodserv/smartphone-os-market-share.jsp (Accessed: 10-25-2016).

[9] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.

[10] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *2010 2nd International Conference on Computer Engineering and Technology*, vol. 2. IEEE, 2010, pp. 297–301.

[11] 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*. ACM, 2013, pp. 224–234.

[12] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other," *The Annals of Mathematical Statistics*, vol. 18, no. 1, pp. 50–60, 1947.

[13] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing Verification and Reliability*, vol. 17, no. 3, pp. 137–158, 2007.

[14] I. B. Mohamad and D. Usman, "Standardization and its effects on k-means clustering algorithm," *Research Journal of Applied Sciences, Engineering and Technology*, vol. 6, no. 17, pp. 3299–3303, 2013.

[15] TechCrunch.com, "Users have low tolerance for buggy apps only 16% will try a failing app more than twice," 2013. [Online]. Available: https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/ (Accessed: 10-25-2016).

[16] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 183–192.

[17] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards black box testing of android apps," in *2015 10th International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2015, pp. 501–510.