Event Trace Reduction for Effective Bug Replay of Android Apps via Differential GUI State Analysis

Yulei Sui
University of Technology Sydney
Australia

Yifei Zhang
Alibaba Group
China

Wei Zheng
Northwestern Polytechnical
University, China

Manqing Zhang
Northwestern Polytechnical
University, China

Jingling Xue
University of New South Wales
Australia

ABSTRACT

Existing Android testing tools, such as Monkey, generate a large quantity and a wide variety of user events to expose latent GUI bugs in Android apps. However, even if a bug is found, a majority of the events thus generated are often redundant and bug-irrelevant. In addition, it is also time-consuming for developers to localize and replay the bug given a long and tedious event sequence (trace).

This paper presents Echo, an event trace reduction tool for effective bug replay by using a new differential GUI state analysis. Given a sequence of events (trace), Echo aims at removing bug-irrelevant events by exploiting the differential behavior between the GUI states collected when their corresponding events are triggered. During dynamic testing, Echo injects at most one lightweight inspection event after every event to collect its corresponding GUI state. A new adaptive model is proposed to selectively inject inspection events based on sliding windows to differentiate the GUI states on-the-fly in a single testing process. The experimental results show that Echo improves the effectiveness of bug replay by removing 85.11% redundant events on average while also revealing the same bugs as those detected when full event sequences are used. Our tool is publicly available at https://github.com/zmqgeek/Echo and its demo video is available at https://youtu.be/0UCVIIEigEI.

CCS CONCEPTS

• **Theory of computation → Program analysis**.

KEYWORDS

Android testing, program analysis, bug replay

ACM Reference Format:
Yulei Sui, Yifei Zhang, Wei Zheng, Manqing Zhang, and Jingling Xue. 2019. Event Trace Reduction for Effective Bug Replay of Android Apps via Differential GUI State Analysis. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3338906.3341183

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5572-8/19/08...$15.00
https://doi.org/10.1145/3338906.3341183

1 INTRODUCTION

GUI testing has become an essential part of the Android development cycle to improve the overall quality of Android apps. Due to non-deterministic user events in Android apps [23, 27], traditional human testing is unable to test an app thoroughly. Many Android testing techniques have been proposed to find latent GUI bugs via automatic event generation [4–8, 15–20]. However, replaying the bugs found by testing tools to help developers localize them still remains a time-consuming task. Tedious event traces pose a major challenge to effective bug replay, even if a bug is found.

Background and Insights. Existing testing tools generate GUI events (e.g., tapping and dragging) to explore all possible program executions for the purposes of finding GUI bugs. A testing process for an app can be described as a series of state transitions. It starts with a (unique) initial state. Then, events (e.g., a tapping on the screen) generated by a testing tools are injected into the Android system to exercise the app. An event can cause some side-effects (e.g., a click on a button in an activity), which may cause a state transition (e.g., transiting to another activity) or stay at the same state if an event is side-effect-free (e.g., a tapping on the blank area of the screen). Once a bug (e.g., a crash) is found, the testing process will terminate at a (unique) error state. To reproduce this bug, the triggering event sequence are re-injected into the Android system to replay the bug finding process. In order to increase coverage, GUI testing tools generate a large quantity and a wide variety of user events. However, in reality, bugs only reside in some parts of an app. Frequently, testing tools end up exercising many bug-unrelated code regions through redundant events, which makes developers difficult to identify critical and meaningful events for effective bug replay.

Challenges. Developing a practical replay technique for Android apps is challenging. First, existing tools often generate a large number of various events in the presence of dynamically changed GUIs. The side-effect of an event can only be known when the event interacts with a particular GUI. This needs to be observed dynamically during testing. Second, recording and inspecting a large amount of events adds the overall overhead of the testing process. It is nontrivial to develop a cost-effective strategy to record useful information where necessary to reflect the testing process in order to correctly replay bugs. Third, the side-effect of an event (e.g., a click on the screen) generated by a testing tool may differ. Good criteria need to be developed to differentiate the side-effects of events to remove redundant events driven by the triggered bugs.

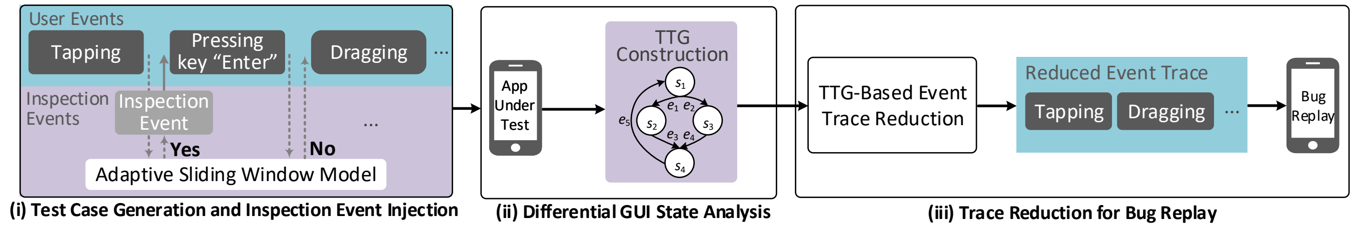

Figure 1: An Overview of ECHO.

Our solution. We present ECHO, an event trace reduction technique for effectively replaying bugs found by existing GUI testing tools, such as Monkey [4]. Due to the event-driven nature in Android apps, GUI bugs are triggered through a sequence of user events to interact with the Android system via dynamically changed GUIs. *GUI state*, which is the GUI information extracted from the dynamically generated Android layout XML file of the current GUI on the screen, is the key to reflecting the GUI changes causing a bug. Given this insight, ECHO aims at removing redundant events by exploiting the differential behavior between the GUI states collected when their corresponding events are exercised. As illustrated in Figure 1, ECHO has the following three components:

(i) Test generation and inspection event injection. Our test generation phase adopts the same event generation strategy as in Monkey, an internal Android GUI testing tool, which is often used to compare with different testing tools in the literature [10, 14, 17, 20, 22]. ECHO injects at most one inspection event after every user event to collect its corresponding *GUI state*. The inspection event serves as a snapshot to record the side-effect of the event if it changes the current GUI state. To efficiently record GUI states under a large amount of events, we propose an adaptive model based on sliding windows to selectively inject inspection events. As illustrated in Figure 1(i), an inspection event is injected between two user events only if the model allows it (i.e., returns 'Yes'). By adjusting the granularity of inspection, the sliding window facilitates trade-offs between efficiency and effectiveness for bug replay.

(ii) Differential GUI state analysis. Due to the highly interactive nature of Android GUIs, the outcomes of injecting an event usually manifest on the screen. The current GUI state of an app can be extracted from the dynamically updated XML file describing the current GUI on the screen in the Android resource folder. Therefore, the side-effect of an event can be obtained by differentiating between the current GUI and the previous one. We abstract the testing process of an app using a Testing Trace Graph (TTG), through which the event trace reduction can be formulated as a path-finding problem. A TTG is constructed on-the-fly during the single testing process. It is a directed graph $\langle S, E \rangle$, where $s \in S$ is a set of nodes with each representing a GUI state, and each edge $(s \xrightarrow{e} s') \in E$ denoting an event e that causes a state transition from s to s'.

(iii) Trace reduction for bug reply. A TTG has an entry and an exit, which represent the initial and error states of the testing process. Once a bug is found during testing, TTG construction is completed. Event trace reduction is to find a shortest path from the initial to the error state. The reduced event trace is collected from the edges on that path. The bug can be replayed by re-injecting the reduced trace. Hence, the bug-irrelevant events are removed to improve the overall quality of bug replay.

2 A MOTIVATING EXAMPLE

This section illustrates the idea and workflow of ECHO using a bug in a real-world Android app `Addi` [1] (a scientific computation tool), found and replayed by ECHO as illustrated in Figure 2.

2.1 Bug Scenario

`Addi` provides a command-line interface (CDI) for users to perform scientific computation. It can also load and execute a script via a file manager from the device. If the file manager is not installed, `Addi` sends an implicit intent for downloading it by navigating users to an app store (e.g., Google Play). However, the app crashes if the app store is not installed on the current device since no available components can handle this intent. The following subsections describe ECHO's event trace reduction process, which reduces a sequence of 162 events that triggered this bug to only four (i.e., e_0, e_1, e_{160} and e_{161}) with five bug-relevant GUI states (i.e., s_0, s_1, s_2, s_{35} and s_{36}) for a successful bug replay as shown in Figure 2.

2.2 Test Generation and Inspection Events

Monkey-style testing, which relies on a random event generation strategy, requires a seed for a pseudo-random number generator to generate each test case. To fairly generate event traces, we randomly select five seeds to perform test case generation when testing every app including `Addi`. The shortest event sequence that triggered the above mentioned bug consists of 162 events. ECHO injects inspection events on-the-fly during dynamic testing to record the current GUI state, i.e., the structure and contents of the XML layout file of the current GUI on the screen. At most one inspection event is injected after each generated user event based on a sliding window model. The window size ws is the inspection interval, requiring one inspection every ws events.

2.3 Differential GUI State Analysis

By using inspection events, differential GUI state analysis constructs a TTG on-the-fly during testing, aiming at exploiting the differential behavior before and after a user event. Figure 2 gives the constructed TTG with 37 nodes (screenshots) representing the corresponding 37 unique GUI states. The bug-relevant states, s_0, s_1, s_2, s_{35} and s_{36}, which form the shortest path to the error state on TTG, are highlighted inside blue rectangles. s_{19}, a bug-irrelevant state (after a keyboard input) that is not on the shortest path, is highlighted in a red rectangle. All other redundant states, i.e., s_3 to s_{18} and s_{20} to s_{34}, are omitted for brevity. The transitions between the GUI states on the shortest path are highlighted in blue arrows. On the contrary, the red dotted arrows (e.g., e_2, e_3, e_4 and e_{159}) represent redundant events reduced for bug replay.

Let us take a look at the transition between s_0 and s_1 on the TTG using our differential GUI state analysis. The event e_0 sends

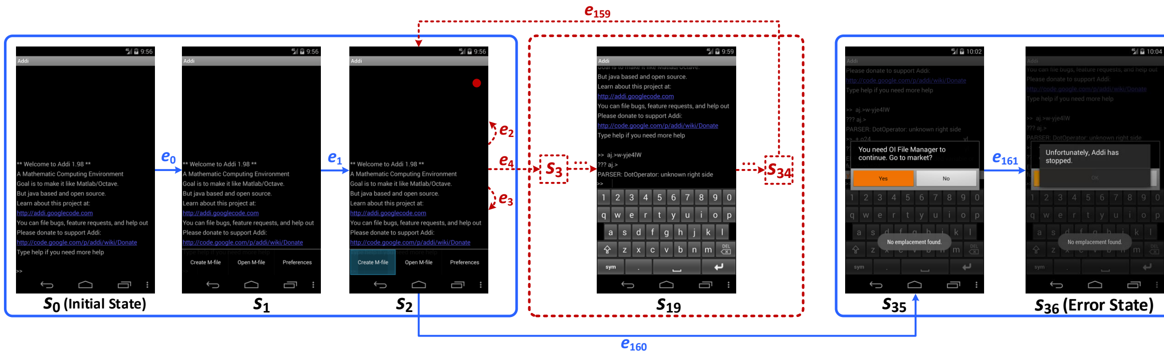

Figure 2: TTG constructed on-the-fly during the testing process Addi. s_n denotes a GUI state, corresponding to the n-th node on the TTG. e_i, which represents an state transition edge between two states, denotes the i-th user event that causes a state transition. ⟶ represents the shortest path (reduced event trace) to the error state on TTG. - - → represents the redundant state transitions removed by Echo. ● denotes the position of the tapping event of e_3 on the screen.

KEYCODE_MENU to press the menu key (the three dots button on the bottom right corner of s_0). The side-effect of this event is to pop out the menu (s_1 in Figure 2), which can be detected via the inspection event by comparing the current GUI state with the previous one. The difference between the two GUI states allows Echo to add a new state s_1 into the TTG and connect s_0 to s_1 with e_0 on its edge, indicating that the transition is caused by this event.

Some events do not introduce any side-effects. For example, the event e_3 taps the top-right corner on the screen, (highlighted using ● in s_2 of Figure 2). Therefore, e_3 is side-effect-free since there is no GUI element to interact with the event on that blank area. A self-loop is introduced with e_3 on the edge to represent this side-effect-free transition on the TTG.

2.4 Event Trace Reduction and Bug Replay

A TTG captures all GUI state transitions for a particular bug. TTG construction is on-the-fly during the testing process until the bug is found. A TTG can have cycles (e.g., s_2, s_3, ... s_{19}, ... s_{34}, s_2) since the current GUI state can be transited to any of the old ones on the TTG via different events. For example, tapping the return button on the screen can cause a transition from the current GUI to the old one. Due to cycles on the TTG, we may have multiple paths from the initial to the error state.

Echo formulates the event trace reduction problem as a path-finding problem on the TTG. The longest path is the one containing the full event sequence (i.e., e_0, e_1, e_2, ... e_{161}), which is obviously unnecessary costly for bug replay, since many of the events residing in cycles are redundant. We use Dijkstra's algorithm to find a shortest path from the initial to the error state. The reduced trace is obtained on the edges along the shortest path. The shortest path from s_0 to s_{36} are highlighted using blue arrows via events e_0, e_1, e_{160} and e_{161}. 158 events are reduced including 85 side-effect-free (e.g., e_2 and e_3) and 73 events with side-effects, i.e., the ones can differentiate GUI states, but identified as redundant e.g., e_{159}. Finally, the reduced trace consisting of only four events (e_0, e_1, e_{160} and e_{161}) are re-injected into the system to replay this bug.

3 EVALUATION

3.1 Tool Implementation

We have implemented Echo in Appium [2], an open source test automation infrastructure, which uses vendor-provided automation frameworks under the hood so that apps can be tested without any modification. Echo's test generation component, based on Appium, provides the same configuration options as the original Monkey. It can generate the following types of GUI events: tapping and dragging on the screen, snapshot, volume adjustment, pressing keys on the device and multi-touch gestures such as pinching and zooming. In addition, a throttle event can create a fixed delay between other types of user events to allow the devices to process and react to the events just injected. The percentage of each type of events in Echo can be configured via command line options.

Echo injects at most one inspection event after each user event generated by a GUI testing tool. To effectively and efficiently record the testing process for building TTG, the injection of an inspection event is determined by a sliding window model. The window size ws represents the inspection interval, i.e., suggesting one inspection after every ws user events. f denotes the adjusting frequency, which allows Echo to periodically check the status of the current TTG once every f inspections. Δ denotes the delta value to increase or decrease the current window size.

The sliding window model leverages the historical information to predict the future window size for inspection event injection by adjusting its granularity. For each periodical check, we will increase the window size ($ws \quad ws + \Delta$) if there are no new GUI states added to TTG. Otherwise, the window size is decreased as $ws \quad {}^{11}ws \quad \Delta^{\circ} > 1^{\circ} ? {}^{1}ws \quad \Delta^{\circ} : 1^{\circ}$ since the minimum window size is 1. We illustrate our model using an example in Figure 3, where $f = 2$, $\Delta = 1$ and ws is dynamically adjusted. Our model starts with a fine-grained inspection with its window size $ws = 1$. The model injects inspection events ie_0 and ie_1 after user events e_0 and e_1, respectively. Since $f = 2$, Echo periodically checks the status of the TTG once every two inspections. Through the second inspection ie_1, we found no changes on the TTG. The window size is then adjusted to 2 to reduce testing overhead by avoiding ineffective inspections that repeatedly record user events that all likely stay on the same GUI (e.g., e_2 and e_3 in Figure 2). Consequently, the inspection is then performed once after every two user events. Through the inspection event ie_3, the window size is decreased to 1 because there is a new GUI state added to the TTG. Shrinking the sliding window allows us to pay closer attention to the new GUI states on the changed TTG. Note that a larger sliding window

Table 1: ECHO's results

App Name	Sliding Widow Size Fixed to One					Adaptive Sliding Window Model Applied					Bug
	#TTG Nodes / #States	#TTG Edges / #User Events	#Inspection Events	#Event After Reduction	#Reduction Percentage	#TTG Nodes / #States	#TTG Edges / #User Events	#Inspection Events	#Events After Reduction	#Reduction Percentage	
Aagtl	3	5	5	2	60.00%	3	4	**4**	**2**	60.00%	FileNotFoundException
Addi	38	162	162	4	97.53%	36	106	**106**	**4**	97.53%	ActivityNotFoundException
Amazed	4	100	100	1	99.00%	4	57	**57**	**1**	99.00%	NullPointerException
Photostream	8	21	21	4	80.95%	6	11	**11**	**3**	85.71%	NullPointerException
ADSdroid	8	30	30	5	83.33%	8	18	**18**	**5**	83.33%	IndexOutOfBoundsException

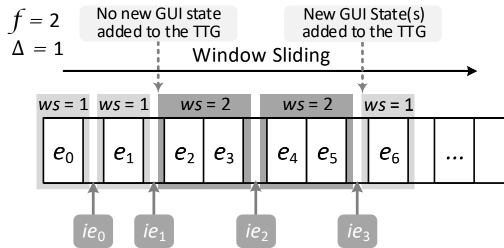

Figure 3: Inspection events based on sliding windows.

is more efficient but at the cost of ineffective bug replay (due to missing bug-relevant events in replay).

3.2 Results and Analysis

We have evaluated ECHO using five real-world Android apps (Table 1) from the F-Droid repository [3]. `Aagtl` is a GPS-based geocaching aide. `Addi` is for scientific computation. `Amazed` is a game app. `Photostream` for photo sharing. `ADSdroid` is a datasheet searching app. For each of the five apps, ECHO has successfully detected and replayed the bugs as listed in the last column in Table 1. The `FileNotFoundException` error in `Aagtl` is triggered when a user event tries to open a non-existent file on the SD card. `Addi` triggers an `ActivityNotFoundException` when sending an implicit intent that does not have a receiver. The `NullPointerException` error in `Amazed` occurs when calling a method through an null pointer, which is correctly initialized but is set to null before the call. The `NullPointerException` error in `Photostream` is due to a null value of a parameter passed from a caller. The `IndexOutOfBoundsException` error in `ADSdroid` is detected when retrieving an element from an empty ArrayList.

Table 1 gives the ECHO's results under two settings. Columns 2-6 show the results when the sliding window size is fixed to one. Columns 7-11 give the results when the adaptive model is applied. In both settings, the original user events are significantly reduced by ECHO for effective bug replay. When the window size is fixed to one, we have the same number of inspection events (Column 4) as user events (Column 3). ECHO's differential GUI state analysis has successfully eliminated 84.16% (on average) user events (Column 6), while replaying the same bugs when full event traces are used.

When the adaptive model is applied, Column 9 shows that a substantial number of inspection events are reduced (5 to 4 in `Aagtl`, 162 to 106 in `Addi`, 100 to 57 in `Amazed`, 21 to 11 in `Photostream`, and 30 to 18 in `ADSdroid`). The events after reduction (Column 10) by this setting are almost the same as those (Column 5) under the setting when the window size is fixed to one (except `Photostream`). It demonstrates that our adaptive model is effective in identifying critical and meaningful events while reducing unnecessary inspections for effective bug replay.

4 RELATED WORK

GUI testing. Recently, there are several research efforts on developing effective techniques for finding GUI bugs. Monkey [4] is a representative random testing approach by generating a large quantity and a wide variety of events to achieve good code coverage. Choi et al. present SWIFTHAND [8], a testing technique that uses machine learning to learn a model of an app and uses it to generate test inputs. Azim et al. propose A^3E [6], a tool that combines static and dynamic analysis to automatically generate test cases to exercise various Android activities. Mao et al. present SAPIENZ [17], a multi-objective search-based testing technique to increase coverage. Song et al. present EHBDROID [19], an approach that does not generate GUI events but directly invokes callbacks of event handlers through instrumenting the app. TRIMDROID [18] employs static analysis to improve the quality of GUI testing by extracting GUI dependencies. Su et al. present STOAT [20], a model driven approach to detecting latent bugs by increasing GUI testing coverage via a stochastic model. Collider [13] and SynthesiSE [12] present symbolic execution based approaches to Android testing. Unlike the prior art, ECHO, as an effective replay approach, accepts the event sequence that triggers a bug from any testing tool. Thus, ECHO can be an important complementary tool to existing GUI testing tools to improve their bug replay processes.

Test Case Reduction. Delta debugging [25, 26] is an automated debugging method to narrow down causes of program failures. Recently, it has been used for test case reduction in Android testing. Clapp et al. [11] develop a variant of delta debugging to minimize long event traces reaching a particular Android activity. DetReduce [9] minimizes the test suites for regression testing by eliminating redundant loops and trace fragments. SIMPLYDROID is a trace simplification technique [14], which uses Activity IDs to form a trace representation and applies delta debugging to find a minimal subtrace that triggers a bug. ECHO differs from SIMPLYDROID in two aspects. First, instead of repeatedly testing an app until a minimal subtrace is found, ECHO performs a single testing process to produce effective traces by injecting lightweight inspections based on an adaptive sliding window. Second, ECHO abstracts the testing process using an on-the-fly built TTG, on which reduction is formulated as a path-finding problem. Our differential GUI state analysis can also be applied to delta debugging to achieve good trade-offs between efficiency and precision to enable more powerful bug replay.

5 CONCLUSION

This paper presents ECHO, an event trace reduction tool for effective bug replay of Android apps using differential GUI state analysis. In the future, we plan to enhance ECHO by integrating more sophisticated static analysis techniques, e.g., program dependence analysis [21, 24] and static happens-before analysis [23] to facilitate dynamic analysis to detect and replay more complicated concurrency bugs (e.g., order violations) in Android apps.

REFERENCES

[1] [n.d.]. Addi, a math calculation environment. https://f-droid.org/en/packages/com.addi/
[2] [n.d.]. Appium: Mobile App Automation Made Awesome. http://appium.io/
[3] [n.d.]. F-Droid: Open-Source Android Apps Repository. https://f-droid.org/
[4] [n.d.]. Google Monkey. https://developer.android.com/studio/test/monkey.html
[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *FSE '12*. 59.
[6] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *OOPSLA '13*. 641–660.
[7] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria. In *ASE '16*. 238–249.
[8] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning. In *OOPSLA '13*. 623–640.
[9] Wontae Choi, Koushik Sen, George Necula, and Wenyu Wang. 2018. DetReduce: minimizing Android GUI test suites for regression testing. In *ICSE '18*. ACM, 445–455.
[10] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *ASE '15*. 429–440.
[11] Lazaro Clapp, Osbert Bastani, Saswat Anand, and Alex Aiken. 2016. Minimizing GUI Event Traces. In *FSE '16*. 422–434.
[12] Xiang Gao, Shin Hwei Tan, Zhen Dong, and Abhik Roychoudhury. 2018. Android testing via synthetic symbolic execution. In *ASE '18*. ACM, 419–429.
[13] Casper S Jensen, Mukul R Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *ISSTA '13*. ACM, 67–77.
[14] Bo Jiang, Yuxuan Wu, Teng Li, and W. K. Chan. 2017. SimplyDroid: Efficient Event Sequence Simplification for Android Application. In *ASE '17*. 297–307.
[15] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An Input Generation System for Android Apps. In *FSE '13*. 224–234.
[16] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *FSE '14*. 599–609.
[17] Ke Mao, Mark Harman, and Yue Jia. [n.d.]. Sapienz: Multi-objective Automated Testing for Android Applications. In *ISSTA '16*. 94–105.
[18] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing Combinatorics in GUI Testing of Android Applications. In *ICSE '16*. 559–570.
[19] Wei Song, Xiangxing Qian, and Jeff Huang. 2017. EHBDroid: Beyond GUI Testing for Android Applications. In *ASE '17*. 27–37.
[20] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. [n.d.]. Guided, Stochastic Model-based GUI Testing of Android Apps. In *FSE '17*. 245–256.
[21] Yulei Sui and Jingling Xue. 2016. SVF: interprocedural static value-flow analysis in LLVM. In *CC '16*. ACM, 265–266.
[22] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of android test generation tools in industrial cases. In *ASE '18*. ACM, 738–748.
[23] Diyu Wu, Jie Liu, Yulei Sui, Shiping Chen, and Jingling Xue. 2019. Precise Static Happens-Before Analysis for Detecting UAF Order Violations in Android. In *ICST '19*. IEEE, 276–287.
[24] Xuezheng Xu, Yulei Sui, Hua Yan, and Jingling Xue. 2019. VFix: value-flow-guided precise program repair for null pointer dereferences. In *ICSE '19*. IEEE Press, 512–523.
[25] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *FSE '99*. 253–267.
[26] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.
[27] Yifei Zhang, Yulei Sui, and Jingling Xue. 2018. Launch-mode-aware context-sensitive activity transition analysis. In *ICSE '18*. IEEE, 598–608.