CrawlDroid: Effective Model-based GUI Testing of Android Apps

Yuzhong Cao
Institute of Software, CAS
Beijing City University
Beijing, China
caoyuzhong14@otcaix.iscas.ac.cnm

Guoquan Wu
State Key Laboratory of Computer Science, ISCAS
University of Chinese Academy of Sciences
Beijing, China
gqwu@otcaix.iscas.ac.cn

Wei Chen
State Key Laboratory of Computer Science, ISCAS
University of Chinese Academy of Sciences
Beijing, China
wchen@otcaix.iscas.ac.cn

Jun Wei
State Key Laboratory of Computer Science, ISCAS
University of Chinese Academy of Sciences
Beijing, China
wj@otcaix.iscas.ac.cn

ABSTRACT

This paper presents an effective model-based GUI testing technique for Android apps. To avoid local and repetitive exploration, our approach groups equivalent widgets in a state and designs a novel feedback-based exploration strategy, which dynamically adjusts the priority of actions based on the execution result of those already triggered ones, and tends to select actions that can reach news states of apps. We implemented our technique in a tool, called CrawlDroid, and conducted empirical experiments. Our results show that the proposed technique is effective, and covers more code within a fixed testing budget.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

mobile application, GUI, model-based testing

ACM Reference Format:
Yuzhong Cao, Guoquan Wu, Wei Chen, and Jun Wei. 2018. CrawlDroid: Effective Model-based GUI Testing of Android Apps. In *The Tenth Asia-Pacific Symposium on Internetware (Internetware '18), September 16, 2018, Beijing, China.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3275219.3275238

1 INTRODUCTION

Mobile applications (or simply, "apps") ecosystems have experienced tremendous growth in the past few years. Android, a mobile OS from Google, currently dominates the market share in the global smartphone market. There are over 3.0 million apps in Google Play [28] and we use apps daily to perform lots of activities.

Internetware '18, September 16, 2018, Beijing, China

ACM ISBN 978-1-4503-6590-1/18/09…$15.00
https://doi.org/10.1145/3275219.3275238

Apps, like other software, must be adequately tested to eliminate the potential bugs and improve the quality. According to a recent study on mobile app development [19][20], mobile app testing still relies heavily on manual testing in practice. Lots of android testing frameworks, such as Android Espresso [15], Robotium [31], UIAutomator [33] and Appium [8], have been proposed, which support to run test cases across different devices. However, these test cases are written manually, which is a tedious process and only covers a small part of the state space of apps under test (AUTs).

On the other hand, many automated testing techniques for apps [13] have been proposed recently. The simplest methodology is fuzzy testing, which generates random events to explore the behaviors of AUTs and detect failures. However, one major limitation of fuzzy testing is that lots of redundant events are generated due to the nature of randomness. Besies this, tracing and debugging suspicious path to a detected failure is more difficult than systematic approaches [10].

This paper focuses on model-based GUI testing of Android apps as it facilitates systematic exploration of AUTs and effective debugging [3][18] by building a GUI model to represent the behavior space of AUTs and generating test inputs based on the model. However, according to a recent empirical study [13], existing model-based testing techniques [9][3][12] achieve lower coverage on average (<30%) than fuzzy testing (e.g., Monkey[6], Dynodroid[22]) under their study. New exploration strategy is required for model-based testing to effectively test apps.

Different from fuzzy testing (which generate random events as test inputs), model-based testing will perform an action on each identified *executable* widget according to predefined traversal policy (e.g. *Depth-first search*, *Breadth-first search*). It will be very likely that the testing tool spends lots of testing resources to explore only small part of a state without improving any code coverage. For example, performing a *click* action on all list items sequentially may reach the same states which only have difference in text content.

To address this challenge, this paper proposes a novel technique that aim to improve the effectiveness of model-based testing. Our approach designs a novel feedback-based exploration strategy, which groups widgets in a state and assigns a priority value for each supported action that the group supports. Based on the execution result of an action, our exploration strategy will adjust the priority of the action (belonged to a group). By grouping widgets and dynamically

adjusting priority of actions, the proposed technique can avoid falling into local exploration, and has more chances to reach new states of AUTs.

We implemented our technique in a tool, called CrawlDroid, and apply it to 45 open-source apps and 12 commercial apps. The experimental results show that CrawlDroid is effective, improves the state-of-the-art of model-based testing. In summary, the main contribution of our work are:

- We present a novel feedback-based exploration strategy, which can avoid local and repetitive exploration and have more chances to trigger actions that will expose new states of AUTs.
- We implement the idea into a practical tool called CrawlDroid, and provide empirical evidence that it improves code coverage, saves testing time, detects more failures compared with existing model-based testing techniques;

2 EXAMPLE AND OVERVIEW

In this section, we introduce a real mobile app Smartisan Notes (com.smartisan.notes) to illustrate some challenges that are not well addressed by existing model-based testing techniques. Fig.1(b) shows the main activity after this app is launched, which contains a note list. User can make new notes, modify or delete an existing note. Such kind of apps are very common and pre-installed on almost all devices.

To explore the state after app starts (see Fig.1(b)), existing model-based testing tools will click all the items in the ListView widget to improve code coverage. However, performing *click* action on these items will not test more new functionality. In fact, as shown in Fig.1(a) and Fig.1(c), the states after clicking different items are the same as only text contents are different.

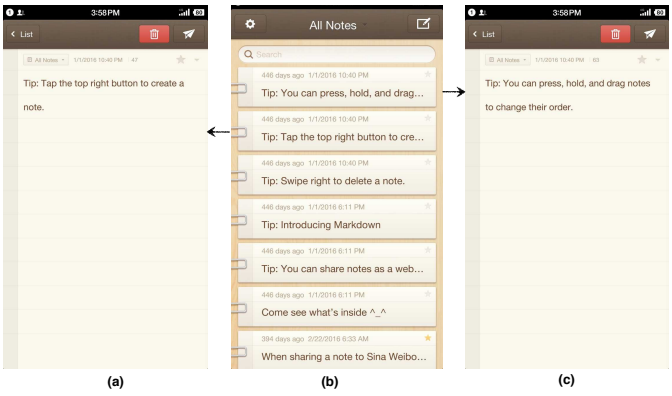

Figure 1: Click List Item

To address this limitation, we propose a novel feedback-based exploration strategy, which group widgets according to their position in a state, and assign an initial priority for each action that the group supports (also called *action group*). After performing an action on a widget, if a new state is exposed, our approach will increase the priority of corresponding *action group*, making this *action group* have more opportunity to be triggered. Otherwise, it decreases the priority of this *action group*. Our approach also designs a biased random algorithm, which tends to select the actions with high priority, while at the same time not starving the actions with low priority.

3 OUR APPROACH

One main limitation of existing model-based testing techniques is that they spend lots of time exploring some local parts of AUTs without gaining coverage improvement. There exists many equivalent widgets in the app, which will reach the same state after being fired. To avoid repeatedly performing actions without contributing to new code coverage, our approach groups the widgets in a GUI state, and designs a novel feedback-based exploration strategy, which intends to trigger actions that can improve code coverage. In the following, we will describe our exploration technique in detail.

Definition 1 (Equivalent Widgets). *Two widgets in a GUI state are considered to be equivalent, if they are both executable (e.g.,clickable or long-clickable), and their paths from root to the node (that represents the widget) are the same when ignoring their position in sibling nodes.*

Based on *equivalent widgets*, CrawlDroid groups widgets in a state, and each group is represented by a common path (called *group path*) that equivalent widgets have (ignoring their position in sibling nodes). The widgets in the same group also support the common action. Therefore, if one widget supports a domain-specific action (learned from manual tests), our approach infers that all other widgets in the same group can support this action. As each group can support different actions (e.g., click, longClick), we define *action group* to distinguish different actions that each group supports.

Definition 2 (Action Group). *action group is defined as a triple ⟨group path, action type, priority⟩, where group path represents all the equivalent widgets, action type denotes the supported action of this group, and priority is an integer value, and a high priority means this action will have more chance to improve code coverage.*

Based on the above definition, the *action group* of a compound action can be specified as ⟨(*group path, action type*),· · · ,(*group path, action type*),*priority*⟩.

Our feedback based exploration technique will assign an initial priority for each supported action. Table I shows the supported UI actions by default and their initial priority. CrawlDroid distinguishes between *app action* and *device action*. The *app action* includes *click*, *longClick*, *scroll* and *inputText*. The *device action* includes *pressBack*, *rotate* and *swipe*. For the action learned from manual tests, their initial priority is also set as 50.

Similar to *app action*, *device action* also has its *action group* (where *group path* is *null*). Our approach assigns higher initial priority for *app action* than *device action* as it tends to perform *app action* firstly when reaching a new state. However, their priority can be adjusted dynamically according to whether new states can be exposed. For *inputText* action, the initial priority is -1, meaning that it has no opportunity to trigger independently. However, this action can be combined with other actions (e.g., click a button) to try to change state.

Our feedback-based exploration technique assumes that the same action performed on equivalent widgets will have similar effects. They either expose new GUI state or reach the same GUI state. If the actions performed on equivalent widgets reach the same GUI state, the priority of corresponding *action group* will be decreased to

Table 1: UI Action and Initial Priority

Action type	Init. score	Action type	init. score
click	50	longClick	50
scroll	40	inputText	-1
swipe	10	pressBack	10
rotate	10		

reduce the chances that this action group is selected later. Otherwise, the priority will be increased to give those un-triggered equivalent widgets (in the same *action group*) more chances to be selected.

Algorithm 1 shows the proposed exploration strategy. It first starts the AUT (*startApp* function). Function *TransToHighPriorityState* will summarise the priority of untriggered actions for each state, and then transfers to state S with the highest priority based on *SFG*.

Function *biasSelectGroup* (line 18) implements a biased random algorithm which tends to select an *action group* with high priority. For a new state S, it invokes *computeGroupAction* (line 20) to compute *action groups*, which will first group *equivalent widgets* in S, and then construct *action group* for each action that widget groups support. For each action ac in $domainActs$, according to widget group that ac's target belongs to, an *action group* is constructed for ac. If no such widget group exists in S, ac will be skipped. This function also builds a *action group* for each *device action* in S. After that, *getActionGroups* (line 20) is invoked to return all the *action groups* (whose priority is > 0) in S. For each *gp* group, *sumPriority* summarises the priority of all *action groups*, and each element of $arrayPriority$ saves the accumulated priority of those visited *gp*s. For a generated random number, *mapping* function first determines the range where this number locates, and then returns corresponding *action group* (saved in *group*).

Function *randSelAction* (line 6) will select an action ac from gr, and then is triggered by *triggerAction*. For *click* action, this function will check whether input fields exist and generates some inputs before firing this action. Function *compare* (line 8) will match nodes in state S and N. If state does not change, function *decGroupPriority* (line 10) will decrease the priority of gr. If $N.priority$ is less than a threshold value, function $transitState$ will transfer to a state that has the high priority. If state changes, *SFG* is updated by comparing N with existing states in *SFG*. If N is a new state, *incGroupPriority* (line 16) is invoked to increase the priority of gr, and function *computeActionGroup* (line 18) will construct action groups for N.

Our exploration strategy assigns a *select* attribute for each action in an *action group* to assure this action has not been selected when exploring a new state. However, during the phase of executing an compound action or transferring states in *SFG*, these triggered actions can be selected and fired again. Currently, if a new state is exposed after an action is triggered, the priority of corresponding *action group* will be increased 5. If no hidden state is exposed, the priority of action group will be decreased 10 to reduce the chances that this *action group* is selected later. By doing this, our biased random selection algorithm tends to select the actions that can reach more new states and test more functionality within a limited testing time, while not starving those actions with low priority.

Algorithm 1: Overall Exploration Algorithm

```
1  begin
2      startApp(); SFG ← SFG_init;
3      S ← TransToHighPrioriyState(SFG);
4      while constraintSatisfied(CC) do
5          gr ← biasSelectGroup(S);
6          ac ← randSelAction(gr); N ← triggerAction(S, ac);
7          ac.select ← 1 for app action;
8          compare(N, S);
9          if state does not change then
10             decGroupPriority(S, gr);
11             if N.priority < threshold then
12                 S ← transitState(SFG, N);
13         else
14             SFG.update(N, ac);
15             if N is a new state then
16                 incGroupPriority(S, gr);
17                 computeActionGroup(N); S ← N;
18 Procedure biasSelectGroup(S)
19     sumPriority ← 0; arrayPriority ← []; groups ← [] ;
20     computeGroupAction(S); groups ← getActionGroups(S);
21     foreach gp ← groups do
22         sumPriority ← sumPriority + gp.priority;
23         arrayPriority.push(sumPriority);
24     random ← Random.nextInt(sumPriority);
25     group ← maping(arrayPriority, random, groups);
26     return group;
```

4 IMPLEMENTATION

We implement our approach in a tool called CrawlDroid, which is publicly available. Generally, it consists of two parts: client and server. The communication between client and server is based on *Android Debug Bridge* (*ADB*).

The client part is implemented as *InstrumentationTestRunner* [17] and is started along with AUT using *adb instrument* command. After client starts, it will receive the command from the server through *broadcast receiver*, perform the action and extract the view hierarchy information of running activity using APIs that UIAutomator provides. Server pulls the layout information from the client, and invokes the exploration strategy described in section III to update GUI model, select an action based on execution result, and then send it to the client through *adb broadcast* command.

5 EVALUATION

The goal of our experiments is to observe the effects of the proposed techniques on improving model-based testing by conducting empirical studies on both open-source and commercial Android apps. In our evaluation, we investigate the following research questions:

- **RQ1:** How does the coverage achieved by the proposed feedback based exploration technique compared to existing model-based testing technique?
- **RQ2:** Can CrawlDroid find any real bugs on real-world apps?

In the rest of this section, we present the subject apps, the experimental protocol and our results.

5.1 Experimental Protocol

We randomly chose 28 apps from 68 benchmark apps studied by Choudhary et al. [13], 17 open-source apps from F-Droid Repository [30], and 12 commercial apps from the largest Android Market

Table 2: GUI Graph and Code Coverage based on DFS and FeedBack

Subject	BFS				Feedback			
	States	EG	AC	MT	States	EG	AC	MT
WordPress	13	72	9	6	28	60	9	15
MyExpenses	44	151	23	7	54	200	67	10
Sanity	1	5	3	6	32	69	86	46
OSChina	58	189	19	25	88	223	45	40
AnyMemo	30	115	15	10	72	169	66	15
DalvikExplorer	36	235	43	51	22	68	69	80
NRPNews	49	254	30	5	64	38	77	6
BookCatalogue	61	190	34	18	78	74	54	25
Tomdroid	34	177	50	31	36	79	63	34
Tipitaka	17	115	36	78	66	158	73	90
ShoppingList	1	6	33	4	3	4	66	13
Blokish	13	48	33	41	12	7	100	55
Mileage	39	131	14	28	42	242	42	33
LogicalDefence	11	94	100	13	11	41	100	14
PasswordMaker	5	23	33	41	9	18	66	54
Whohasmystuff	11	135	100	58	18	83	100	91
WorldClock	14	134	50	79	23	194	50	80
OpenManga	46	206	35	14	86	56	57	35
FileExplorer	23	142	50	30	42	230	50	42
Ultramegatech	13	197	50	60	15	24	50	63
Omnomagon	17	126	100	14	30	73	75	34
ALogcat	13	153	100	73	16	126	100	74
Feeder	34	193	45	31	47	284	55	64
BatteryManager	4	19	100	71	7	4	100	71
Yahtzee	3	35	100	43	5	5	100	47
AGrep	25	137	66	49	18	150	83	58
Mirrored	14	138	75	76	28	155	100	82
BatteryDog	3	19	100	87	5	17	100	89
Addi	8	27	50	9	13	9	50	11
CrimeTalk	28	162	100	41	31	159	100	41
A2DP	23	175	50	60	27	73	100	63
Democracydroid	29	200	80	29	22	123	100	37
Autoanswer	3	24	100	15	3	14	100	15
Zooborns	5	20	50	23	5	11	50	23
LearnMusicNotes	15	106	100	57	14	176	100	62
DdalyHeart	8	37	75	9	11	49	75	12
Chronosnap	7	53	100	31	9	35	100	45
ImportContacts	19	100	100	37	20	65	100	39
NetCounter	13	56	66	55	3	14	66	57
MiniNoteViewer	6	6	12	2	57	320	12	57
AnyCut	14	100	100	72	15	67	100	77
CountDownTimer	1	4	100	26	3	18	100	70
Multismssender	15	102	66	52	17	22	33	68
Alarmclock	25	180	60	5	28	123	60	64
Dotools Clock	16	145	17	16	31	121	40	29
AudioClass	10	53	25	4	10	53	25	4
WeatherBug	39	146	11	30	46	134	17	30
Cdxc	7	24	11	2	16	24	28	9
TED	18	152	8	33	28	153	28	35
CNN mobile	14	130	9	13	34	120	36	28
Ergedd	29	223	17	24	43	221	25	26
Smartisan Notes	15	164	16	14	26	175	30	17
Jams Music	30	103	25	5	71	108	56	29
Qukan	41	217	8	18	55	216	17	20
Flixster	22	190	8	31	62	196	31	44
Qiyouyuedu	26	178	25	19	46	194	33	36

AnZhi [7] in China. The first column of Table II lists the selected subjects.

We built a testing environment on both a PC and android devices (including emulator and real physical device). The server part of CrawlDroid runs on a 64-bit Linux machine with 4GB memory, 3.40GHz Intel Core i7 processor. The android emulator is configured with 200MB memory and the physical device is JianGuo YQ604 with 2GB memory. Both devices install Android KitKat version (Android 4.4 - API level 19).

In our studies, open-source apps runs on the emulator. Commercial apps runs on the real device, as some functionalities of them cannot work on the emulator. For the emulator, each app was given a freshly created environment. After every run, we destroyed the emulator to prevent it from affecting other runs. For the physical device, only default system services are allowed to run.

To compute code coverage, CrawlDroid uses ELLA [29] to achieve method coverage, and calculates activity coverage by calling Android's *ActivityManger* for extracting activity information. This paper focuses on activity/method coverage, as one advantage of model-based testing is to guide test cases generation [18][9][11]. If CrawlDroid covers more activities and methods, it can guide target-based test cases generation more effectively. Note that,we also use EMMA [1] to achieve statement coverage, and details are omitted for space limitation.

To investigate RQ1, we implemented a *BFS* (Breadth-first search) based exploration strategy in CrawlDroid, which will trigger default actions on all executable widgets, and *device actions* for each explored state. This strategy is also used as the baseline and compared to the proposed feedback based exploration strategy. Each app is tested for 60 minutes, and every 1 minute we calculate the achieved coverage of two model-based testing techniques.

One important goal of the automated GUI testing is to detect errors while exploring the behavior space of AUTs. Therefore, beside code coverage, we checked how many failures of two model-based testing techniques can reveal with a time budget of one hour per app for investigating RQ2.

5.2 Results

To answer RQ1, Table II shows the covered states (States), transition edges between states (EG), activity coverage (AC%) and method coverage (MC%) of selected apps using BFS-based and feedback-based exploration strategy (called BFS and Feedback for simplicity later), respectively. It can be seen that for all of selected apps, Feedback technique covers more states, achieves higher activity/method coverage than BFS. Although BFS has more transition edges for some apps (e.g.,BookCatalogue), it exposes less states and archives lower code coverage as it spent lots of time to explore states without exposing new states.

Table 3: Statistics on found crashes

Strategy	App Crashes	Unique Crashes	Tests length		
			min	max	ave
Feedback	20	28	3	43	6.6
BFS	10	10	4	32	8.7

In answering RQ2, Table III shows the detected reproducible runtime failures in benchmark apps for Feedback and BFS, respectively. It can be seen that Feedback approach reports 28 unique failures in 20 apps, and BFS approach reports 10 unique failures in 10 apps. We manually checked these failures and found that they have only 3 common failures. The reason for the less common failures is that Feedback and BFS have different exploration strategy, and they may explore different states of AUTs where the error occurs. For the test sequence to reproduce the failures, two approaches have similar length on average. For Feedback, the mean length is 6.6, and for BFS, the mean length is 8.7.

6 THREATS TO VALIDITY

Like any empirical study, there are potential threats to validity of our experimental results. Firstly, to avoid local and repetitive exploration, currently our approach groups equivalent widgets based on path information of the widget in the view hierarchy. Although achieving good results, it may group widgets incorrectly (e.g., for the same actions on equivalent widgets, some reach the same state while some reach different states). In the future work, we will investigate to introduce more conditions (e.g., layout of widget in the screen, resource id) to group widgets correctly.

Secondly, our experimental results may not be generated for other types of apps. To mitigate this threat, we randomly select 28 apps from 68 benchmark subjects used by Choudhary et al. [13], select 17 apps from F-Droid, and 12 commercial apps from AnZhi app store. These selected apps covers different categories (e.g.. tool, media, news, communication, education). In the future work, we plan to test more different types of commercial apps to evaluate the effectiveness of our approach.

Finally, we only evaluated our approach on a single version of the Android platform. Considering the rapid evolution of Android system, the performance of three evaluated techniques may vary as the subsequent versions become available.

7 RELATED WORK

Recently, many techniques and tools have been proposed, which aim to test Android apps automatically to achieve high code coverage and reveal faults.

The simplest approach is fuzzy test input generation, which generates random events as test inputs to explore behaviors of AUTs and checks runtime failures. Android Monkey [6] is the best-known random testing tool, and Dynodroid [22] is an advanced random testing tool that not only generates *UI events* but also *system events*.

One main problem of random testing is that it generates lots of redundant events due to the nature of randomness. Zeng et al. [37] addresses the limitation of random testing by incorporating *widget awareness* and *state awareness*, and selects the widgets according to predefined priority rules. In [39], they further improves the exploration technique, which allows developers to specify what widgets should not be *clicked* to avoid redundant exploration, and use adb command to improve the efficiency of firing an action.

Model-based approaches are popular for testing Android apps, which builds a model that represents a behavior space of an AUT and generates test inputs based on the model. *AndroidRipper* [3][4] builds a model using a depth-first search over the user interface. A^3E [9] is an automated GUI exploration tool for android applications, which can explore the app running on the actual devices. *Swifthand* [12] uses machining learning to learn a model of the app, uses the learned model to generate user actions that visit unexplored states of the app, and uses the execution of app to refine the model. *PUMA* [16] is a framework that helps testers incorporate different exploration strategy into dynamic analysis of Android apps. It also provides a FSM representation of AUT. Baek et al. [10] proposed a set of multi-level GUI comparision criteria that provides the selection of multiple abstraction levels for GUI model generation. The state comparision criteria used in CrawlDroid is similar to C-Lv4 in [10], which mainly considers layout widgets and executable widgets.

Some use static analysis to infer possible user actions for each widget. For example, *ORBIT* [36] statically analyses the source code of the app to understand which UI events are supported by a specific activity and builds a model of the app by crawling it from a starting state. The state space model of AUT can also be built statically. A^3E [9] statically builds Activity Transition Graph of the app by means of taint analysis, and proposes a targeted exploration strategy, which supports fast, direct exploration activities. Yang et.al [35][34] build a GUI model of apps using static window transition graph, which represents the possible GUI window sequence and their associated events and callbacks. Guided by constructed model, test cases can be generated to detect some potential bugs (e.g., resource leaks [38]).

As some app behavior can only be revealed upon providing specific inputs, some testing tools use more sophisticated techniques such as symbolic execution [27][5][26] and evolutionary algorithm to guide the exploration [26]. *ACTEve* [5] is based on symbolic execution and concolic testing technique, which supports the generation of both UI and system events. *EvoDroid* [23] adopts evolutionary algorithm to generate inputs, which represents individuals as sequences of test inputs and implements the fitness function so as to maximise coverage. *SAPIENZ* [24] combines random fuzzing and search-based exploration techniques, which can optimise test sequences, minimising length while maximising coverage and fault revelation. The exploration strategy adopted by *CrawlDroid* can be seen as a combination of model-based testing and fuzzy testing.

There are also some work [21][25][14][2], which leverage the domain knowledge in the test cases to improve the effectiveness of automatic exploration technique. *Testilizer* [25] infers a finite state model of the web application from existing test suites and explores more states based on this model, while generating assertions about the DOM state. Ermuth et al. [14] proposed to mine the some macro events (a sequence of low-level events) from the execution traces, which are combined with random testing to cover more pages and improve code coverage. Thor [2] makes use of existing test suites in mobile apps, seeking to expose them to adverse conditions.

8 CONCLUSION

This paper presents an effective test input generator for Android apps, which aims to improve the state-of-the-art of model-based testing of android apps. To address the limitation of existing model-based testing techniques, CrawlDroid designs a novel feedback-based exploration strategy which tends to select an action that will have more chances to expose new states of AUTs. Our empirical study shows that CrawlDroid is effective. In the future, we plan to investigate more techniques to improve model-based testing, improve the efficiency of our tool. We will also compare CrawlDroid with fuzzy testing tool Monkey [6], and some advanced two phase testing tools Stoat [32] and Sapienz [24] to further evaluate the effectiveness of our tool.

9 ACKNOWLEDGEMENTS

This work is partially supported by National Basic Research Program (973) of China under Grant No. 2015CB352201 and National Natural Science Foundation of China under Grant No. 61472407.

REFERENCES

[1] EMMA: a free Java code coverage tool. 2017. http://emma.sourceforge.net/.
[2] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. 2015. Systematic execution of android test suites in adverse conditions. In *ISSTA*. ACM, 83–93.
[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. 2012. Using GUI ripping for automated testing of Android applications. In *ASE*. ACM, 258–261.
[4] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M Memon. 2015. MobiGUITAR: Automated model-based testing of mobile apps. *IEEE Software* 32, 5 (2015), 53–59.

[5] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *FSE*. ACM, 59.

[6] The Monkey UI android testing tool. 2017. http://developer.android.com/tools/help/monkey.html.

[7] Anzhi. 2017. AnZhi store. In *http://www.anzhi.com/*.

[8] Appium. 2017. http://appium.io/.

[9] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 641–660.

[10] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *ASE*. IEEE, 238–249.

[11] Ravi Bhoraskar, Seungyeop Han, Jinseong Jeon, Tanzirul Azim, Shuo Chen, Jaeyeon Jung, Suman Nath, Rui Wang, and David Wetherall. 2014. Brahmastra: Driving Apps to Test the Security of Third-Party Components.. In *USENIX Security*. 1021–1036.

[12] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 623–640.

[13] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *ASE*. IEEE, 429–440.

[14] Markus Ermuth and Michael Pradel. 2016. Monkey see, monkey do: effective generation of GUI tests with inferred macro events. In *ISSTA*. ACM, 82–93.

[15] Google Espresso. 2017. https://google.github.io/android-testing-support-library/.

[16] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. 2014. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*. ACM, 204–217.

[17] InstrumentationTestRunner. 2017. https://developer.android.com/reference/android/test/InstrumentationTestRunner.html.

[18] Casper S Jensen, Mukul R Prasad, and Anders Møller. 2013. Automated testing with targeted event sequence generation. In *ISSTA*. ACM, 67–77.

[19] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real challenges in mobile app development. In *ESEM*. IEEE, 15–24.

[20] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *ICST*. IEEE, 1–10.

[21] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining android app usages for generating actionable gui-based execution scenarios. In *MSR*. IEEE Press, 111–122.

[22] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *FSE*. ACM, 224–234.

[23] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. Evodroid: Segmented evolutionary testing of android apps. In *FSE*. ACM, 599–609.

[24] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *ISSTA*. ACM, 94–105.

[25] Amin Milani Fard, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Leveraging existing tests in automated test generation for web applications. In *ASE*. ACM, 67–78.

[26] Nariman Mirzaei, Hamid Bagheri, Riyadh Mahmood, and Sam Malek. 2015. Sigdroid: Automated system input generation for android applications. In *ISSRE*. IEEE, 461–471.

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

[28] Number of Android applications. 2017. In *http://www.appbrain.com/stats/number-of-android-apps*.

[29] Ella:Binary Instrumentation of Android Apps. 2017. https://github.com/saswatanand/ella.

[30] Open Source Android Application Repository. 2017. https://f-droid.org/.

[31] RobotiumTech robotium. 2017. https://github.com/RobotiumTech/robotium.

[32] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *FSE*. ACM, 245–256.

[33] Google UIAutomator. 2017. http://developer.android.com/training/testing/ui-testing/uiautomator-testing.html.

[34] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. 2015. Static control-flow analysis of user-driven callbacks in Android applications. In *ICSE*, Vol. 1. IEEE, 89–99.

[35] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. 2015. Static Window Transition Graphs for Android (T). In *ASE*. IEEE, 658–668.

[36] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *FASE*. Springer, 250–265.

[37] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: are we really there yet in an industrial case?. In *FSE*. ACM, 987–992.

[38] Hailong Zhang, Haowei Wu, and Atanas Rountev. 2016. Automated test generation for detection of leaks in Android applications. In *AST*. IEEE, 64–70.

[39] Haibing Zheng, Dengfeng Li, Xia Zeng, Beihai Liang, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *ICSE SEIP*. ACM.