Benchmarking Automated GUI Testing for Android against Real-World Bugs

Ting Su
Shanghai Key Laboratory of Trustworthy Computing, East China Normal University
China
tsu@sei.ecnu.edu.cn

Jue Wang
State Key Lab for Novel Software Tech. and Dept. of Computer Sci. and Tech., Nanjing University
China
juewang591@gmail.com

Zhendong Su
Department of Computer Science, ETH Zurich
Switzerland
zhendong.su@inf.ethz.ch

ABSTRACT

For ensuring the reliability of Android apps, there has been tremendous, continuous progress on improving automated GUI testing in the past decade. Specifically, dozens of testing techniques and tools have been developed and demonstrated to be effective in detecting crash bugs and outperform their respective prior work in the number of detected crashes. However, an overarching question "*How effectively and thoroughly can these tools find crash bugs in practice?*" has not been well-explored, which requires a ground-truth benchmark with real-world bugs. Since prior studies focus on tool comparisons *w.r.t.* some selected apps, they cannot provide direct, in-depth answers to this question.

To complement existing work and tackle the above question, this paper offers the first ground-truth empirical evaluation of automated GUI testing for Android. To this end, we devote substantial manual effort to set up the Themis benchmark set, including (1) a carefully constructed dataset with 52 real, reproducible crash bugs (taking two person-months for its collection and validation), and (2) a unified, extensible infrastructure with six recent state-of-the-art testing tools. The whole evaluation has taken over 10,920 CPU hours. We find a considerable gap in these tools finding the collected real bugs — 18 bugs cannot be detected by any tool. Our systematic analysis further identifies five major common challenges that these tools face, and reveals additional findings such as factors affecting these tools in bug finding and opportunities for tool improvements. Overall, this work offers new concrete insights, most of which are previously unknown/unstated and difficult to obtain. Our study presents a new, complementary perspective from prior studies to understand and analyze the effectiveness of existing testing tools, as well as a benchmark for future research on this topic. The Themis benchmark is publicly available at *https://github.com/the-themis-benchmarks/home*.

CCS CONCEPTS

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

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08…$15.00
https://doi.org/10.1145/3468264.3468620

KEYWORDS

GUI testing, Android apps, Crash bugs, Benchmarking.

ACM Reference Format:
Ting Su, Jue Wang, and Zhendong Su. 2021. Benchmarking Automated GUI Testing for Android against Real-World Bugs. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3468264.3468620

1 INTRODUCTION

Android apps typically run in complex end-user environments post-deployment. Ensuring their reliability and correctness — avoiding *fatal crashes* in particular — is thus a top priority of any app development team. Since the first effort by Hu and Neamtiu [13] in 2011, tremendous and continuous efforts have been made to improve automated GUI testing for Android [19, 47, 49], which complement the commonly-adopted manual testing in this field [18, 48]. Specifically, dozens of automated GUI testing or fuzzing tools (*e.g.*, [5, 9, 12, 15, 24, 25, 29, 31, 37]) have been developed and demonstrated to be effective in detecting crash bugs and outperform their respective prior work in the number of detected crashes.

Thus, an overarching question is "*How effectively and thoroughly can these tools find crash bugs in practice?*". To answer this question, an ideal approach is to directly assess these tools against a ground-truth benchmark with real-world bugs, and check how many bugs are found or missed by a given tool. Indeed, such a benchmarking approach is well-justified and widely-adopted in practice for evaluating software testing or analysis tools [17], *e.g.*, LAVA [8], Defects4J [16] and DeCapo [4]. It has two *key benefits*: (1) *enabling many direct, in-depth analyses* (*e.g.*, analyzing the false negatives and common weaknesses of tools), and (2) *consolidating the evaluation validity* (*e.g.*, avoiding such false positives as bug overcounting due to the imprecision of bug de-duplication strategies). In contrast, evaluating testing tools against *only* apps (without known bugs) is difficult to obtain such benefits if not impossible.

On the other hand, no effort exists in the literature yet to answer the aforementioned question against ground-truth. For example, by investigating the recent literature reviews [19, 47, 49] and relevant publications in this field, we identified 32 research papers that propose automated testing techniques for detecting crash bugs in Android apps. However, none evaluated the proposed techniques against real-world bugs, and all compare tools *w.r.t.* some selected apps alone. Similarly, all prior relevant empirical studies [3, 7, 32, 50, 52] in this field also use only apps (without known bugs) to

Table 1: Key differences between the prior relevant studies and ours in evaluating automated testing techniques for Android ("✓", "✗" and "?" denote that the study can, cannot or can only partially give answers, respectively, and "N/A" is not applicable).

Studies	Venue	Evaluation Basis			Analysis Basis				New Study Insights		
		#Tools	Basis	Has the ground-truth?	Basis	Are crashes known and reproducible?	Examine tool implementations?	Discuss/confirm with tool authors?	Quantifying bug finding abilities (RQ1)	Common challenges (RQ2)	Factors and Opportunities (RQ3)
Choudhary *et al.* [7]	ASE'15	multiple	apps	✗	#crashes, coverage	✗	✗	✗	✗	?	✗
Wang *et al.* [50]	ASE'18	multiple	apps	✗	#crashes, coverage	✗	✗	✗	✗	✗	✗
Zheng *et al.* [52]	ICSE'17-SEIP	one	one app	✗	coverage	N/A	✗	✗	✗	?	✗
Behrang *et al.* [3]	ASE'20	one	apps	✗	coverage	N/A	✗	✗	✗	?	✗
Our study	ESEC/FSE'21	multiple	real bugs	✓	real-world bugs	✓	✓	✓	✓	✓	✓

evaluate testing tools. Therefore, we still do not have direct, in-depth answers to the aforementioned question.

Our work aims to fill this gap by providing the first ground-truth evaluation of existing testing techniques for Android. *To clearly present the necessity and novelty of our study, Table 1 summarizes the key differences between the prior relevant studies [3, 7, 50, 52] and ours.* These differences show that our study presents a new, complementary perspective from prior studies.

Specifically, the studies by Choudhary *et al.* [7] and Wang *et al.* [50] compare one tool to another based on some selected apps in terms of *the numbers of found unique crashes* and achieved code coverage. However, due to the lack of ground-truth, we cannot analyze the false negatives of these tools on a common basis. As a result, quantifying the degree of effectiveness of these tools becomes difficult. In practice, these two studies face two additional challenges. First, existing testing tools for Android cannot reliably provide reproducible tests for found crashes (see Section 5 in [7] and Section 3.5 in [36]) due to the open technical challenges like GUI flakiness [27, 46] and lengthy tests [6]. As a result, it is difficult to analyze the bug features, thus unable to offer fine-grained analyses on the tools' bug finding abilities. Second, existing tools *heuristically* de-duplicate crashes by hashing stack traces, which is difficult to make reliable, thus likely incurring bug-overcounting [17].

The studies by Zheng *et al.* [52] and Behrang *et al.* [3] investigate the tool limitations by analyzing the uncovered code of one or more apps, respectively. But these two studies cannot give direct answers to the raised question, because they only focus on code coverage, which is a proxy indicator of bug finding abilities and the correlation could be weak [14] (our study also observes this in Section 4.2). Moreover, they only evaluate one tool, Monkey [29]. Thus, the generability of the identified tool limitations is unclear.

To achieve our study, one important step is to setup a ground-truth benchmark with real-world bugs based on an agreed-upon criterion. To this end, we resort to the industrial practitioners for gaining insights. Specifically, we contact 8 senior app testing managers and engineers (with 3∼10 years' working experience) from five well-known companies, *i.e.*, Google, Facebook, Tencent, ByteDance and Testin (a major mobile app testing service provider in China) within our networks. Their teams are responsible for testing their own apps (like Google Pay, Messenger, WeChat and TikTok which have billions of monthly active users worldwide) or the apps from different vendors. We conduct independent on-line interviews with them per company, and ask them 5 prepared and some follow-up questions to fully understand their testing practice.

Finally, all the interviewees respond that in practice they assign priority labels to the bugs reported by in-house testing or app users, and they prioritize *critical bugs* (namely *important bugs*) — the bugs that break the major app functionalities and affect the larger percentage of app users (in practice, realtime crash reporting platforms are used to track crash issues from end-users). In other words, critical bugs are more likely to affect more users in reality. All the interviewees indicate and agree that the ability of finding critical bugs is an objective metric to measure the effectiveness of testing tools in practice. Thus, we decide to choose critical bugs as the agreed-upon criterion to setting up the benchmark. In fact, such ability has already been strongly and widely advocated for evaluating testing tools in both industry and academia [26, 34].

To this end, we take three steps to approach this study. First, we choose open-source apps as the targets to collect critical bugs because their issue repositories are public. Specifically, we designate the importance of bugs *w.r.t.* their issue labels assigned by app developers themselves. We collect the bugs with critical issue labels like *high-priority*, *blocking-release*, *P1-urgent*. We finally construct a dataset of 52 real bugs from 20 open-source Android apps by crawling the issue repositories of 1,829 Android apps. This process took us substantial manual effort (nearly *two person-months*) that could not be automated. It involves manually reviewing bug reports, locating buggy code versions, building app binaries, and reproducing and validating bugs. Section 3.1 details this step.

Second, we rigorously setup a unified, extensible experimental infrastructure, and integrate Monkey [29], the state-of-the-practice testing tool, and five most recent state-of-the-art ones for thorough evaluation, namely Ape [12], Humanoid [21], ComboDroid [15], TimeMachine [9] and Q-testing [31]. Specifically, we run these tools on the collected bugs, and profile different metrics: the number of bugs they can find (*i.e.*, *effectiveness*), how many times they can trigger a bug given a number of runs (*i.e.*, *stability*), and how long they take to trigger a bug (*i.e.*, *efficiency*). Section 3.2 details this step. We name our dataset and infrastructure as the Themis benchmark, which aims for an objective evaluation *w.r.t.* ground-truth.

Finally, we give the detailed quantitative and qualitative analysis on the testing results of these tools by reviewing the bug features, examining these tools' implementations, and discussing/confirming with the tool authors. We identify the common challenges that existing tools face and the factors that affect bug finding, which have not been well-identified by the prior studies. In particular, we investigated the following research questions (answered in Section 4):

- **RQ1**: *How effectively and thoroughly can these testing tools find the collected real-world bugs?*
- **RQ2**: *Are there any common challenges that all these tools face in finding these bugs (by analyzing the common false negatives)?*
- **RQ3**: *Are there any factors affect these tools in finding these bugs (by pair-wisely analyzing the testing results of these tools)? What are the opportunities for improving the state-of-the-arts?*

Summary of main findings. Out of 52 bugs, 18 (≈34.6%) bugs *cannot* be detected by any testing tool, which indicates that a considerable gap exists between the existing tools and the collected

Table 2: Summary of the selected automated GUI testing tools for Android in our study.

Tool Name	Venue	Open Source	Main Technique	Need App Source Code?	Need App Instrumentation?	Supported SDKs	Tool Version	Implementation Basis
Monkey [29]	-	✓	Random testing	✗	✗	Any	default	-
Ape [12]	ICSE'19	✓	Model-based	✗	✗	6.0 / 7.1	a53e98c	Monkey-based
Humanoid [21]	ASE'19	✓	Deep learning-based	✗	✗	Any	c494c7d	DroidBot-based
ComboDroid [15]	ICSE'20	✓	Model-based	✗	✓	6.0 / 7.1	567b3f6	Monkey-based
TimeMachine [9]	ICSE'20	✓	State-based	✗	✓	4.4 / 7.1	e79beb5	Monkey-based
Q-testing [31]	ISSTA'20	✗	Reinforcement learning-based	✗	✗	4.4 / 7.1 / 9.0	045825a	-

real-world bugs. Specifically, these 18 bugs impose five common major challenges blocking any tool, *e.g.*, *deep use case scenarios*, *changes of system/app settings*, and *specific user interaction patterns*. It indicates that continuous, long-term research effort is needed to tackle these challenges (Section 4.2). On the other hand, the gap is larger when these tools are applied individually — they miss a large portion (53.8~71.2%) of bugs, although we indeed observe their unique advantages in finding specific bugs (Section 4.1). Also, we find these tools have obvious randomness in triggering bugs, and no one can absolutely outperform the others in bug finding. By pairwise comparisons, we find that their testing results are largely affected by the *GUI exploration strategies*, *state abstraction criteria*, and *small heuristics*, which are the opportunities for tool improvement in the short-term (Section 4.3). Table 6 in Section 4.4 summarizes the concrete new insights we obtained from this study, most of which are unknown/unstated and difficult to obtain.

Contributions. Our study makes several contributions:

- It takes the *first* step to conduct an empirical study against real-world bugs to evaluate GUI testing tools for Android, which presents a new, complementary perspective from prior studies.
- It carefully setups the Themis benchmark, including the *first* ground-truth dataset of 52 real, reproducible crash bugs and a unified, extensible infrastructure, to achieve this study.
- It gives in-depth quantitative and qualitative analysis on the testing results. It obtains new concrete findings, most of which were unknown/stated before. It also motivates the future research on this topic with a benchmark (discussed in Section 4.4).

2 TESTING TOOLS FOR OUR STUDY

Table 2 lists the selected tools for our study. Note that we use the latest versions of these tools at the time of our study.

Monkey. Monkey [29] is a pure random testing tool. In principle, Monkey emits pseudo-random streams of UI events (*e.g.*, touch, gestures, random texts) and some system events (*e.g.*, volume controls, navigation). Monkey is widely-used in industry for stress-testing because it is easy-to-use and compatible with any Android version. It is a popular baseline to evaluate new testing techniques.

Ape. Ape [12] is a novel model-based GUI testing tool. Different from prior model-based testing tools like Stoat which use static GUI abstraction criteria, Ape uses the runtime information to dynamically evolve its abstraction criterion via a decision tree, which can effectively balance the size and precision of the model. Specifically, with this dynamically refined model, Ape generates UI events via a random and greedy depth-first state exploration strategy. Moreover, Ape also internally utilizes Monkey to occasionally emits random UI events and system events to avoid stucking at local states.

Humanoid. Humanoid [21] is the first deep learning-based testing tool. The core is a deep neural network model that predicts which UI elements on the current GUI page are more likely to be interacted with by users and how to interact with it. The model was trained upon a large-scale crowd-sourced human interactions dataset. Humanoid is expected to drive the GUI exploration towards important states faster as it prioritizes UI elements according to their importance and meaningfulness like a human. Humanoid is built on DroidBot [20], a lightweight, model-based GUI testing tool, which received 500+ stars on GitHub at the time of our study.

ComboDroid. ComboDroid [15] is a novel model-based testing tool. Its core idea is to generate long and meaningful event sequences by combining a number of short, independent *use cases*, to explore deep app states. ComboDroid obtains such use cases either from humans or automatically generates from a GUI model constructed by GUI exploration. It then analyzes the data-flow and GUI-transition relations between obtained use cases, and combines them (*i.e.*, concatenating use cases in specific orders) to generate final tests. Moreover, it works in a feedback loop, *i.e.*, generating additional use cases when prior tests reached new app states.

TimeMachine. TimeMachine [9] is a novel state-based testing tool. Different from prior tools like Sapienz [25] and Stoat [37] that evolve event sequences to maximize code coverage, TimeMachine instead evolves a population of states which can be captured upon discovery and resumed when needed for finding deep errors. During test execution, its core is to take a snapshot of every *interesting state* and add into the state corpus, and travel back to a most *progressive state* and execute next test when the current exploration cannot reach *new* interesting states. Its uniqueness is the ability to snapshot and resume specific app state for further testing via the underlying Android-based virtual machine.

Q-testing. Q-testing [31] is a reinforcement learning-based testing tool. It uses a trained neural network to compare GUI pages. If a page is similar to any of prior explored GUI pages, the comparator will give a small reward. Otherwise, the comparator will give a large reward. These rewards are used and iteratively updated to guide the testing to cover more functionalities of apps.

Sapienz and Stoat. We also evaluated Sapienz and Stoat, although the tools in Table 2 outperform them. Sapienz uses genetic algorithms, while Stoat uses the stochastic model learned from an app to optimize test suite generation. Despite Sapienz is closed-source and only compatible with Android 4.4, we still include it because it is well-known and its technique is unique.

3 EXPERIMENTAL SETUP

3.1 Themis's Dataset

Collect open-source apps. We chose the open-source Android apps on GitHub as the main source of collecting real-world bugs. To include as many candidate apps as possible, we use two strategies: (1) We crawled all the apps from F-Droid [42], the largest open-source app market, because most of these apps are maintained on

GitHub. (2) We used the keyword "Android" and AndroidManifest.xml, the unique file of any Android project, to collect missing Android apps that are only maintained on GitHub but not released on F-Droid. We finally got 1,829 unique Android apps on GitHub.

Filter apps with critical issues. We designate the importance of bugs *w.r.t.* the issue labels assigned by developers. To collect as many critical issues as possible, we built a GitHub API [43] based crawler to collect all the issue labels from 1,829 candidate apps, and manually identified 111 different labels denoting critical issues. Then, we extracted 12 shorten forms of keywords from these 111 labels for matching concrete issue labels. For example, we use "*block*" to match "blocking-release", "blocked"; "*sever*" to match "severity-high", "severity: crash"; "*pri*" to match "high priority", "Priority-Critical", "Major priority"; "*urgen*" to match "urgency: HIGH", "p1-urgent"; "*importan*" to match "important!", "P2: Very Important", "BUG: High Importance" *etc*. We find these shorten forms of keywords can effectively reduce the false negatives of critical issues. By filtering those apps whose issue labels contain one of these 12 shorten forms of keywords, 200 valid apps remained.

From the above results, we find many apps do not have critical issue labels. To further avoid missing critical issues, we continued to scan the remaining 1,629 apps by checking whether any issue whose title, body or comments contain the keywords such as "block", "severe", "critical", "major", "urgent", "important", "heavy" (derived from the 12 shorten forms of keywords). We got 209 valid apps with such issues. Thus, we obtained 409 (=200+209) valid apps in total.

Collect raw data of critical issues. Based on the previous data, we manually inspected each issue of the 200 apps with explicit critical issue labels, and the issues of the 209 apps which have matched keywords. Specifically, a candidate issue for our study should satisfy these criteria: (1) was a crash bug with the keywords "crash" or "exception"; (2) have explicit reproducing steps; (3) was submitted after 1st Jan, 2017 to avoid apps that could have outdated dependencies. We finally got 228 critical issues from 51 apps. *Many issues were excluded because the bug reports do not have clear reproducing steps and the corresponding app was outdated for a long time.*

Validate and archive critical issues. We manually checked and validated each of these 228 critical issues. The typical process is: (1) reviewing and understanding the bug report, (2) locating the buggy code version, (3) building and instrumenting the buggy app version, (4) reproducing the bug, and (5) archiving the bug data. Note that in practice we often have to iterate between step (2) and (4). Because many bug reports are not well-formatted (*e.g.*, missing buggy code version or code fixing commits), we have to manually locate the right version by trial and error until we can reproduce the described bug. Moreover, building apps is very time-consuming because we usually have to resolve outdated or missing dependencies and set up necessary building environments (*e.g.*, local servers). Reproducing bugs also takes time because we have to link the steps to reproduce in text with the app functionalities in GUIs. Many bug reports are not well-written; and many apps do not have clear documentation.

During this process, an issue would be excluded if (1) we cannot fully understand the bug report; (2) the buggy app version cannot be located; (3) the buggy app version cannot be built into an executable APK; (4) the issue cannot be faithfully reproduced on Android 7.1 (the version supported by the selected tools), *e.g.*, the backend server was obsoleted, the bugs were concurrency or compatibility issues; and (5) the issue is deadly simple (*e.g.*, start-up crashes). In addition, we excluded an issue if its corresponding app is not "self-contained", *i.e.*, testing such an app requires the non-trivial collaborations with humans or other devices. For example, a GitHub client app was excluded because none of existing GUI testing tools can automatically test it without any appropriate, complicated app data preparation (*e.g.*, manually creating a sample project repository with proper code commits, issues, branches and other info).

In our experience, it usually took 1~4 hours to validate one issue without the guarantee of success. We spent nearly two person-months on validating the 228 issues, and obtained 52 valid critical issues from 20 apps. For each successfully validated issue, we archived its corresponding bug data, which includes (1) an executable APK file (Jacoco-instrumented), (2) a bug-reproducing video, (3) the exception stack trace, and (4) other necessary information (*e.g.*, login script). Table 3 lists these 52 critical crash bugs. It gives the app name, issue id, app feature, code version, number of stars on GitHub, lines of code (LOC), number of steps to reproduce (#STR) and other bug information (*e.g.*, whether it needs network access, account login or system setting changes for reproducing). Note that #STR denotes the number of *shortest* steps observed by us, and does not include the steps to login or change external system settings.

Discussion. Note that the 20 apps in Table 3 have diverse features and many of them are highly-starred. Thus, these apps could serve a good basis for evaluation. On the other hand, all these 52 bugs can be deterministically reproduced by a GUI test in our evaluation setting, *i.e.*, an ideal testing tool could find each of them. Thus, these bugs provide a fair basis for all testing tools. We note that some prior work [11, 36, 39, 51] provides crash bug dataset. But we did not reuse those datasets. Because those bugs are selected only based on whether the bug reports describe bug-reproducing steps rather than the agreed-upon criterion of critical bugs in our study.

3.2 Themis's Infrastructure

We built a unified, extensible infrastructure for our study. Any testing tool can be integrated into this infrastructure and deployed on a given machine with one line of command:

```
Themis: themis --avd avd_name -n dev_cnt --apk apk_name
     -o output_dir --time testing_time --repeat run_cnt
     --tool tool_name [--login login_script] [--gui]
     [--check_crash] [--coverage]
```

One can specify the target device (avd_name), size of device pool (dev_cnt), target app (apk_name), testing time (testing_time), number of runs (run_cnt), the target testing tool (tool_name), automatic login (via UiAutomator-based scripts [45]), showing GUI screens, checking crashes and dumping coverage at runtime.

Efforts under the hood. To build this infrastructure, we took considerable time to coordinate with the authors of the selected tools to assure correct and rigorous setup. We tried our best efforts to minimize the bias and ensure that each tool is at "its best state" in bug finding. We detail our efforts on each tool as follows.

Ape. We spent slight efforts to setup Ape, but around two weeks to coordinate with the tool authors to ensure its usability. For example, we observe Ape frequently throws OutOfMemory and No Disk Space errors when given a long running time. To resolve these

Table 3: List of 52 real-world, reproducible crash bugs. "#STR" denotes the number of shortest steps to reproduce."N", "L" and "S" denote whether reproducing the bug requires network access, account login and system setting changes, respectively.

App Name	Issue Id	App Feature	Code Version	#GitHub Stars	LOC	#STR	N	L	S	∅	M	A	H	C	T	Q	Sa	St
ActivityDiary	#118	personal diary	1.1.8	58	2,011	9				✗							-	
ActivityDiary	#285	personal diary	1.4.0	58	4,250	8								★			-	★
AmazeFileManager	#1232	file manager	3.2.1	3.2K	16,969	8				✗							-	
AmazeFileManager	#1558	file manager	3.3.2	3.2K	19,456	6				✗							-	
AmazeFileManager	#1796	file manager	3.3.2	3.2K	19,457	9						★	★	★	★			★
AmazeFileManager	#1837	file manager	3.4.2	3.2K	21,070	3					★	★					-	
and-bible	#261	bible study	3.0.286	259	16,162	17	✓			✗							-	
and-bible	#375	bible study	3.1.309	259	16,611	25	✓				★							
and-bible	#480	bible study	3.2.327	259	17,205	14	✓					★		★				
and-bible	#697	bible study	3.2.369	259	20,225	16	✓				★						-	
and-bible	#703	bible study	3.3.377	259	20,301	10	✓				★						-	
AnkiDroid	#4707	flashcard learning	2.9	3.4K	29,390	5					★	★	★	★	★		-	★
AnkiDroid	#4200	flashcard learning	2.6	3.4K	28,572	10				✗								
AnkiDroid	#5638	flashcard learning	2.9.1	3.4K	29,298	4				✗							-	
AnkiDroid	#4451	flashcard learning	2.7	3.4K	28,673	17					★	★				★		
AnkiDroid	#6145	flashcard learning	2.10	3.4K	29,874	17			✓	✗								
AnkiDroid	#5756	flashcard learning	2.9.4	3.4K	29,657	15					★	★	★	★		★	★	★
AnkiDroid	#4977	flashcard learning	2.9	3.4K	29,775	2					★	★	★	★	★	★		★
APhotoManager	#116	photo manager	0.6.4	148	8,969	2					★	★	★	★	★	★	★	★
collect	#3222	online form	1.23.0	528	25,946	7	✓				★	★	★		★			★
commons	#3244	wiki media	2.11.0	661	20,069	8	✓	✓		✗							-	
commons	#2123	wiki media	2.9.0	661	15,540	5	✓	✓			★	★	★	★	★	★	-	★
commons	#1391	wiki media	2.6.7	661	9,182	10	✓	✓		✗							-	
commons	#1385	wiki media	2.6.7	661	9,221	6	✓	✓		✗							-	
commons	#1581	wiki media	2.7.1	661	9,287	6	✓	✓	✓	✗							-	
FirefoxLite	#4881	web browser	2.1.12	251	13,626	7	✓				★	★		★	★			
FirefoxLite	#5085	web browser	2.1.20	251	12,060	5	✓					★	★	★	★			★
FirefoxLite	#4942	web browser	2.1.16	251	13,661	5	✓						★		★			★
Frost-for-Facebook	#1323	facebook wrapper	2.2.1	377	7,749	5	✓	✓	✓		★						-	
geohashdroid	#73	geohashing app	0.9.4	13	5,275	2	✓				★	★	★	★		★		★
MaterialFBook	#224	facebook client	4.0.2	122	1,740	1	✓	✓				★		★			-	
nextcloud	#5173	file-sharing app	3.10.0	2.3K	36,589	5	✓	✓			★	★	★	★	★	★	-	★
nextcloud	#4026	file-sharing app	3.6.1	2.3K	32,798	3	✓	✓			★	★	★	★	★	★	-	★
nextcloud	#4792	file-sharing app	3.9.2	2.3K	35,302	9	✓	✓		✗								
nextcloud	#1918	file-sharing app	2.0.0	2.3K	28,505	3	✓	✓			★	★	★	★			-	★
Omni-Notes	#745	notebook app	6.1.0	2.1K	8,882	10					★	★	★					★
open-event-attendee	#2198	open event app	0.5	1.5K	6,624	7	✓					★	★	★			-	★
openlauncher	#67	home screen app	0.3.1	256	5,591	2			✓		★	★		★	★		-	
osmeditor4android	#729	map editor	11.0.0.8	196	36,887	11	✓					★						
osmeditor4android	#637	map editor	0.9.10	196	32,594	30	✓			✗							-	
Phonograph	#112	music player	0.15.0	2.4K	10,800	1				✗								
Scarlet-Notes	#114	notebook app	6.9.5	300	2,860	12								★				
sunflower	#239	gallery app	0.1.6	12K	1,687	4					★				★		★	
WordPress	#8659	blog manager	11.3	2.5K	68,171	10	✓	✓		✗							-	
WordPress	#7182	blog manager	9.2	2.4K	59,571	2	✓	✓		✗								
WordPress	#6530	blog manager	8.1	2.4K	54,211	24	✓	✓		✗								
WordPress	#11992	blog manager	14.9	2.4K	67,784	6	✓	✓					★	★			-	
WordPress	#11135	blog manager	13.6	2.4K	64,499	9	✓	✓			★	★	★	★			-	★
WordPress	#10876	blog manager	13.7	2.4K	64,564	5	✓	✓		✗							-	
WordPress	#10547	blog manager	13.3	2.4K	64,795	10	✓	✓							★		-	
WordPress	#10363	blog manager	13.1	2.4K	72,387	3	✓	✓				★	★			★	-	★
WordPress	#10302	blog manager	12.9	2.4K	72,202	2	✓	✓			★	★		★	★	★	-	★
#Total	**52**									**18**	22	24	18	21	15	10	3	19

issues, we discussed with the tool authors, and finally reached the consensus that allocating 2GB RAM, 1 GB internal storage and 1 GB external SDCard storage for the Android devices could greatly mitigate this issue. The reason is that Ape maintains all GUI states in memory and dumps large output files and logs. Thus, we also assigned the similar hardware setup for other tools under study to ensure a fair basis. In addition, during the early stage of our study, Ape frequently crashed on a number of apps in our dataset. Thus, we reported all the encountered issues; and the tool authors fixed all those issues before our deployment.

Humanoid. We spent around three days to setup Humanoid. The main effort goes to setting up the compatible TensorFlow version and resolving outdated library dependencies. Other effort includes fixing some obvious implementation bugs in DroidBot (which Humanoid was built on) that affected the usability.

ComboDroid. We spent around one week to coordinate with the tool authors to adapt ComboDroid into our infrastructure. For example, to meet our requirements, the tool authors modified ComboDroid to (1) support running multiple tool instances in parallel, and (2) provide separate tool modules to support our login scripts. During the early stage of our study, we reported some tool crash issues because ComboDroid may fail to instrument some apps by Soot. They fixed all the issues before our deployment.

TimeMachine. We spent around two weeks to adapt the tool into our infrastructure. We made three major modifications, which were later verified by the tool authors before our deployment. (1) TimeMachine requires code coverage information for running. But its original EMMA based coverage collection module cannot work on recent Android apps (created after 2015) which only support Jacoco [44] based coverage profiling. Thus, we replace EMMA with Jacoco. (2) Because TimeMachine uses VirtualBox to snapshot and resume emulator states, all the time-sensitive information will be lost or imprecise. Thus, we added an additional module to enable profiling time-sensitive information (*e.g.*, the time duration to trigger a crash bug). (3) We enhanced TimeMachine to support parallel

Table 4: Results of bug finding for the selected tools. "#Found Bugs" denotes the total number of bugs found by each tool. "n / 5" (0≤n≤5) denotes the breakdown, i.e., which bugs were found in n runs out of the five independent runs.

	M	A	H	C	T	Q	Sa	St
#Found Bugs	22	24	18	21	15	10	3	19
5 / 5	6	10	7	4	4	2	2	4
4 / 5	3	2	1	4	3	3	0	2
3 / 5	3	4	2	4	0	2	1	3
2 / 5	5	2	4	5	1	1	0	4
1 / 5	5	6	4	4	7	2	0	6
0 / 5	30	28	34	31	37	42	49	33

running on server machines, automatic login scripts, Google service apps (required by some apps in our dataset), and fixed several obvious implementation issues to improve its usability.

Other tools. It is easy to setup Monkey and Q-testing. We spent around one week to setup Sapienz and Stoat by supporting parallel running and resolving some usability issues.

3.3 Experimental Setup

We deployed our experiment on a 64-bit Ubuntu 18.04 machine (64 cores, AMD 2990WX CPU, and 128GB RAM). We evaluated all the selected tools on Google Android 7.1 emulators (API level 25). Each emulator is configured with 2GB RAM, 1GB SDCard, 1GB internal storage, and X86 ABI image. Different types of external files (including PNGs/MP3s/PDFs/TXTs/DOCXs) are stored on the SDCard to facilitate file access from apps. We registered separate accounts for each bug that requires login and wrote the login scripts, and during testing reset the account data before each run to avoid possible interference. Note that since Sapienz is only compatible with Android 4.4, we were unable to run Sapienz on all the 52 bugs but only 19 bugs (verified to be reproducible on Android 4.4). The symbol "-" in column "Sa" in Table 3 denotes that the corresponding bug is not reproducible on Android 4.4. For Stoat, we allocated one hour for model learning and five hours for model mutation.

We allocated one device (i.e., one emulator) for each bug/tool in one run (one run required 6 hours), and repeated 5 independent runs for each bug/tool. This time setting was decided based on the setup of these tools in their original papers (Ape uses 1 hour & 5 runs, Humanoid uses 1 hour & 3 runs, ComboDroid uses 12 hours & 3 runs, TimeMachine uses 6 hours & 5 runs, and Q-testing uses 1 hour & 4 runs) and two prior studies (Choudhary *et al.* [7] use 1 hour and 10 runs; Wang *et al.* [50] use 3 hours and 3 runs). Thus, our time setting is large enough. The whole evaluation took over 52×5×6×7 = 10,920 machine hours (not including Sapienz). Due to Android's limitation, we can only run 16 emulators in parallel on one physical machine. Thus, the evaluation took us around 28 days, in addition to around one week for deployment preparation.

4 EXPERIMENTAL RESULTS AND ANALYSIS

4.1 RQ1: Quantifying Bug Finding Abilities

The ultimate goal of testing tools is to find bugs. We measured the bug finding abilities of the selected tools from three different perspectives: (1) *effectiveness*: how many bugs can be found by these tools? Are there any differences between the bugs found by these tools? (2) *stability*: can these tools stably (deterministically) trigger these bugs across the five runs? (3) *efficiency*: how many resources (*e.g.*, time) are required by these tools to trigger these bugs?

Effectiveness. In Table 3, the last eight columns give the testing results of Monkey ("M"), Ape ("A"), Humanoid ("H"), ComboDroid ("C"), TimeMachine ("T"), Q-testing ("Q"), Sapienz ("Sa"), and Stoat ("St") on each bug, respectively. The symbol ★ denotes that the tool found the corresponding bug. In column "∅", the symbol "✗" denotes none of the tools can detect the corresponding bug. We can see, out of the 52 bugs, 18 (≈34.6%) cannot be detected by any tool. *It indicates a considerable gap exists between all the testing tools and the collected real-world bugs. We will look into the gap in RQ2.*

Table 4 summarizes the bug finding results of individual tools. The row "#Found Bugs" denotes the total number of bugs that were found by individual tools across the five runs. We can see Ape, Monkey, ComboDroid, respectively, found 24, 22, and 21 bugs, while Humanoid, TimeMachine, Q-testing found 18, 15, and 10 bugs, respectively. The former three tools found a few more bugs than the latter three ones. Stoat found 19 bugs, while Sapienz only found 3 bugs out of the 19 bugs which it targets. We find Ape, the most effective one among these tools, only found nearly half of all bugs. Monkey, Ape, Humanoid, ComboDroid, TimeMachine, Q-testing and Stoat missed 30 (≈57.7%), 28 (≈53.8%), 34 (≈65.4%), 31 (≈59.6%), 37 (≈71.2%), 41 (≈78.8%), and 33 (≈63.5%) bugs, respectively. *It indicates the gap becomes larger,* i.e.*, more bugs were missed when these tools were applied individually.*

To take a close look, Fig. 1(a) (the bottom-left section) reports the pairwise comparison between the tools on their found bugs. The comparison reports which bugs were found by both tools (reported in gray), and which bugs were found by *only one* of the two tools. This provides us a closer look at the bug finding abilities of these tools. We can clearly see these tools have obvious differences in the bugs that they found. For example, although Monkey, Ape, and ComboDroid are close in the numbers of found bugs, each of them can still find some bugs that the others cannot. This phenomenon also applies to those tools that have obvious differences in the number of found bugs, *e.g.*, Ape and TimeMachine. *It indicates that no one can absolutely outperform the others in finding bugs, and instead they do complement each other by finding different bugs. We will analyze which factors affecting these tools in bug finding in RQ3.*.

Stability. Table 4 gives the breakdown of which bugs were successfully found in how many runs, which indicates the stability of these tools in bug finding. Row n/5 (0≤n≤5) denotes which bugs were triggered in n runs out of the five runs. For example, row "1/5" and column "M" means there are 5 bugs of Monkey were triggered in only one run out of five runs. This is another important metric to consider when adopting a testing tool, which indicates how random a GUI testing tool could be in detecting bugs. However, this metric has not been reported by the prior studies [7, 50] or by the authors of these tools. We can see *a non-negligible number* of bugs were only found in one run but missed in the other four runs (see row "1/5"). For example, TimeMachine and Ape found 7 and 6 bugs, respectively, in only one run. In detail, Monkey, Ape, Humanoid, ComboDroid, TimeMachine and Q-testing have 22.7%, 25%, 22.2%, 19%, 46% and 18.2% bugs, respectively, which were detected in only one run. *It indicates that existing tools have obvious randomness in bug finding, and a non-negligible number of bugs were actually detected by chance.*

Efficiency. Fig. 1(b) gives the bug detection time of individual tools on their found bugs. We can see Ape, ComboDroid and Q-testing

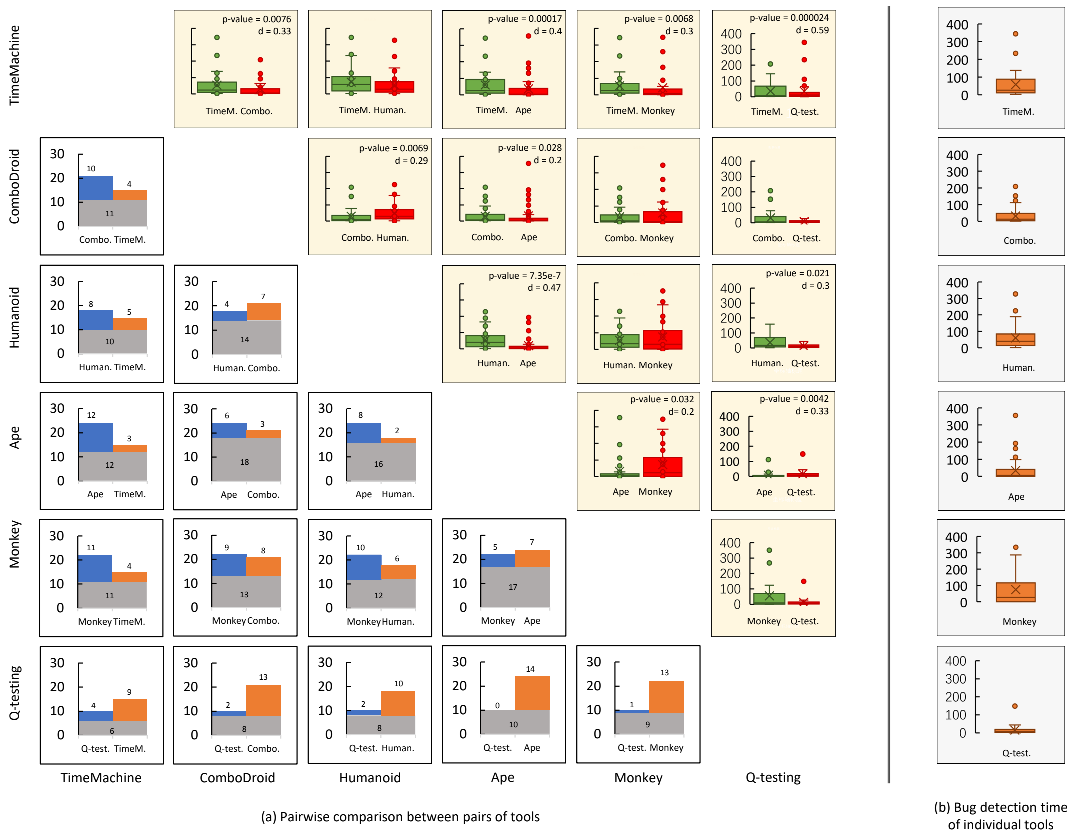

Figure 1: (a) Pairwise comparison of the tools in terms of found bugs and bug detection time (in minutes). The plots on bottom-left section report the differences of found bugs between the pairs of tools (the grey bars shows the common bugs found by both tools), while the plots on top-right section report the bug detection times between the pair of tools on the common bugs (the *p-value* and the standardized effect size d are reported on top-right within each plot if the comparison result is statistically significant). (b) The bug detection time (in minutes) of individual tools on their found bugs. We did not include Sapienz and Stoat in Figure 1 because comparing the recent tools listed in Table 2 are our main focus.

are relatively faster than the other tools in bug finding. Specifically, Ape, ComboDroid and Q-testing detect 20/24, 19/21 and 9/10 bugs within the first one hour respectively, while Monkey, Humanoid, and TimeMachine detect 14/22, 14/18, and 10/15 bugs respectively.

Fig. 1(a) (the top-right section) reports a pairwise comparison between these tools in boxplots on the bug detection time. Note that (1) The comparison reports the running times on the bugs found by *both tools*. We did not consider the bugs found by only one tool because that is unfair. (2) The detection time is the offset between the first bug triggering time and the exact start running time of a tool. For example, TimeMachine takes around 10 minutes to create and setup the VM image before it actually starts the testing. We excluded such preparation time for any tool. Thus, the bug detection time we measured is head-to-head. We can see *the detection times of these tools have obvious differences*. To validate the significance of these differences, we used Mann-Whitney U test [1], a non-parametric statistical hypothesis test for independent samples, to compare the detection times between two tools. We report the *p-value* and *standardized effect size* at the top-right corner for any pairwise comparison which is statistically significant. Here, the significance level α is set as 0.05 (*i.e.*, if p-value<0.05, the difference is big enough to be statistically significant). The standardized effect size d indicates the magnitude of the difference (d<0.3 is small, 0.3≤d<0.5 is medium, d>0.5 is large). From the results, we can see Ape is more efficient than all the other tools in finding bugs. ComboDroid is more efficient than Humanoid and TimeMachine, while Monkey is more efficient than TimeMachine. The major reason of such results is due to the differences of testing strategies and tool implementations.

4.2 RQ2: Common Challenges and Weaknesses

This section aims to identify the common challenges for existing GUI testing techniques and tools in finding the collected bugs.

Analysis Methods. To achieve this analysis, we focus on the 18 bugs (listed in Table 5) which have not been found by any tool. Specifically, we used the following analysis methods to identify the challenges. *First*, we carefully reviewed the 18 bugs to understand their features from both the GUI and code levels. *Second*, we examined the implementations of these tools to understand their testing strategies. *Third*, we conducted the online discussions with the tool authors: we show the bug videos, discuss the possible reasons why their tools miss these bugs, and confirm our observations.

Table 5: Characteristics of the 18 bugs missed by all tools. "#STR" is the number of shortest steps to reproduce.

#Issue Id	#STR	#Distinct Transit.	Text Inputs	Setting Changes	Interact. Patterns	Exter. Interact.
#118	9	6			✓	✓
#261	17	17				
#4200	10	7			✓	
#5638	4	3	✓			
#6145	17	17		✓		✓
#1232	8	8				✓
#1558	6	3			✓	
#3244	8	8				✓
#1581	6	5		✓	✓	✓
#1391	10	9		✓	✓	
#1385	6	6			✓	
#4792	9	8				
#637	30	27	✓		✓	
#112	1	1			✓	
#8659	10	9	✓		✓	
#7182	2	2			✓	
#6530	24	10			✓	
#10876	5	4	✓		✓	

Analysis Results. Table 5 summarizes the characteristics of the 18 bugs via our analysis methods. We distilled five major challenges: (1) *deep use case scenarios*, (2) *specific text inputs*, (3) *changing system or app settings*, (4) *specific user interaction patterns*, and (5) *external app interactions*. Note that one bug may impose multiple challenges at the same time, any of which could block a testing tool. We illustrate these challenges as follows.

C1 (*event trace*): hard to reach deep use case scenarios. Table 5's column "#Distinct Transit." denotes the number of *distinct GUI page transitions* along the bug triggering trace. This number approximates *how deep a bug resides in the app*. We can see that 12 out of 18 bugs (≈66.6%) can only be reached after bypassing more than 5 distinct page transitions. Specifically, *nextcloud*'s #4792 and *and-bible*'s #261 are the two bugs that pose this sole challenge for the selected tools. For example, *nextcloud*'s #4792 has 8 distinct page transitions, and its search space of event traces is at least 16×12×2×1×7×5×2×3=80,642 (each number denotes the number of executable events on one distinct GUI page). This big number blocks any tool from finding the bug within six hours.

Insight: It remains an open challenge for existing tools to reach deep use case scenarios, although some tools like ComboDroid and TimeMachine were designed to reach deep app states; Humanoid was designed to act like humans to cover more app functionalities.

C2 (*text inputs*): no careful design of text input generation. Text inputs are important to trigger some bugs in addition to the GUI actions. In Table 5, 4 bugs out of the 18 bugs require text inputs, and 3 out of these 4 bugs (≈75%) require corner-case (or invalid) text inputs rather than meaningful (or valid) ones. In detail, *AnkiDroid*'s #5638 requires to input the backslash codes (*e.g.*, "\", "\"); *osmeditor*'s #637 requires to fill two invalid, 1-length characters "*" and "0" into the text fields of *value* and *age*, respectively; *WordPress*'s #10876 requires that the content of a post under writing is left as *empty*; only *WordPress*'s#8659 requires to input a valid text (not necessarily meaningful) that can obtain non-empty search entries.

However, existing tools usually generate pure random texts without careful designs, and thus hard to detect these bugs. For example, ComboDroid and TimeMachine simply inherit Monkey's text generation strategy, which generates random texts of digits, letters, or other symbols; Ape optimizes Monkey by additionally generating random integer/float numbers and time/date formatted strings. Humanoid randomly picks texts from the training data.

Insight: *Testing tools should improve the text input generation strategies for bug finding.* In addition to generate meaningful text inputs [22], they should also stress test apps with corner-case or invalid text inputs by analyzing app code or the meaning of text fields, or defining a list of risky text inputs [28]. *Note that the prior studies [3, 7, 52] only suggest generating valid text inputs because they aim for improving code coverage rather than bug finding.*

C3 (*system/app settings*): no dedicated consideration of changing system/app settings. Changing system or app settings are common user behaviors [23]. However, we find none of the selected tools dedicatedly considers the necessity of such changes in bug finding, especially for system settings (because changing system setting usually requires interacting with system app `Settings`). This leads to the incapability of detecting such bugs. In Table 5, 3 bugs out of the 18 bugs involve setting changes, and 2 out of these 3 bugs (≈66.6%) involve system settings. Specifically, *AnkiDroid*'s #6145 requires changing the default system language from English to another language *and* turning on one app preference option; *commons*'s #1581 requires that the system location service is turned off before entering into the *Nearby* page and then is turned on to use GPS for location; and *commons*'s #1391 requires turning on the app's "night mode" theme in the middle of a specific event trace. None of the tools can detect these bugs.

Insight: *The key challenge of considering system or app settings during GUI testing is the large space of possible GUI tests caused by two major reasons.* One reason is the diversity of setting options. For example, Android 7.1 provides 9 main categories of system settings with over 50 concrete setting options [40, 41], all of which could affect app behaviors. But only limited types of system settings were considered before [23, 35]. Another reason is the interleavings between the setting changes and the GUI events. Prior work [23, 35] only changes settings before an app starts and does not change settings at runtime. However, all the 3 bugs require changing settings at specific points at runtime. *Note that the prior studies [3, 52] have not systematically observed this challenge.* Because they analyze the main app code (*i.e.*, Java code) coverage but we observe *not all* setting changes (especially for system settings) will lead to obvious coverage changes in Java code (*e.g.*, changing system languages mainly involves an app's XML resource code). In addition, the implication from prior studies (see Table III in [52]) to generate system events (*i.e.*, sending broadcast intents) cannot work on changing system settings (*e.g.*, security-related settings like permissions and location cannot be changed by sending intents) or app settings.

C4 (*interaction patterns*): no explicit consideration of specific user interaction patterns. Another major challenge which blocks these tools from finding bugs is the lack of generating specific user interaction patterns to pose adverse conditions. We can see that 12 out of the 18 bugs (≈66.6%) pose this challenge. For example, *WordPress*'s #6530 requires *uploading a number of pictures* (making the uploading takes some time) to publish a post and then deleting the post *when the uploading is still in progress*; *osmeditor4android*'s #637 requires *removing all entries but the last one* from its page of validator preference; *commons*'s #1385 requires *a rotation action at one specific page*; *WordPress*'s #8659 requires *scrolling down and back* the *sites* page (revoking the page loading of new items) and select some specific items. *AnkiDroid*'s #4200 requires putting one specific activity in the background for a while and returning back to it (making the Android system destroy and recreate the activity). Despite these bugs seem corner cases, the corresponding user interaction patterns are common in reality.

Insight: We carefully examined the relevant covered code of these bugs. *It reveals that manifesting these bugs requires exercising specific sequences of callback interactions.* For example, *WordPress*'s #6530 involves the interactions between the callbacks of GUI events (for deleting the post) and those of the background thread (for uploading the pictures); *commons*'s #1385 involves the interactions between the lifecycle callbacks of an activity. However, existing tools only focus on maximizing line or activity coverage, which is hard to stress test different callback interactions. One plausible way is to design specific coverage criteria (*e.g.*, callback sequence coverage [33]) or mutation operators [30] to guide testing. *Note that this insight cannot be obtained by prior studies [3, 52] because such bugs will not show differences in terms of line or activity coverage.*

C5 (external interactions): seldom consider the interactions with other apps. 5 out of the 18 bugs (≈27.8%) require interacting with other apps on the device to obtain the desired data (e.g., a picture file) to enable testing the follow-up functionalities. However, most tools do not explicitly consider the necessity of these interactions in bug finding, and instead they constrain the testing efforts within the app under test. For example, Humanoid will simply restart the tested app after certain steps of exploration if it is still exploring the other apps. C5 may be related with C1 and C3.

Insight: It is much desirable for these testing tools to construct external intents provided with desired data or files to simulate the purpose of external app interactions.

4.3 RQ3: Factors and Opportunities

This section discusses the factors we observed that affect bug finding on the collected real-world bugs and the opportunities for tool improvements. Specifically, we conduct the analysis based on the testing results of each tool in Table 3 and the pairwise comparison results in Figure 1. We follow the same analysis methods in RQ2, and summarize our major findings in the following aspects.

GUI exploration (testing) strategies affect bug finding. The tools we studied employ different GUI exploration strategies. *Indeed, these strategies show their unique advantages in finding specific bugs.* For example, Monkey, Ape, ComboDroid, TimeMachine found 4, 1, 2, and 1 bugs, respectively, which the other tools cannot find.

But we also observe that the exploration strategies with more direct and fine-grained guidance seem more effective in finding bugs. For example, in Table 4, Ape, ComboDroid and Stoat detect more bugs than Humanoid, TimeMachine and Q-testing. Specifically, both Humanoid and Q-testing use trained deep neural network to guide exploration: Humanoid explores towards human-preferred pages, while Q-testing prefers exploring pages with different usage scenarios. TimeMachine heuristically deprioritizes those pages that have been visited more times (see Section 3.3 in [9]). Basically, these three tools are only guided to cover more GUI pages. However, this may not be directly linked with bug finding. In contrast, Ape differentiates and explores distinct app states by dynamically refining state abstraction, ComboDroid stress-tests the data-flow relations at the app code level, while Stoat optimizes different event compositions in GUI tests via the stochastic model. These three tools are informed by more fine-grained analysis, and thus are likely to detect more bugs.

Opportunities: Integrating fine-grained (program) analysis results into GUI exploration could be beneficial for bug finding.

State abstraction granularity affects bug finding. GUI layouts are usually used to abstractly represent concrete app states during testing. Due to the large search space of GUI pages, GUI state abstraction strategies (or GUI comparison criteria [2]) are commonly adopted by testing tools to improve testing scalability. *We observe that the bug finding abilities could be affected by the state abstraction granularity, which unfortunately has not been well-recognized by existing tools. Specifically, we observe that the tools with more fine-grained abstraction could detect more bugs, which corroborates the preliminary findings of [2] (see Section 6.3).*

For example, we observe that TimeMachine and Q-testing missed some trivial bugs like *WordPress*'s #11135 and *nextcloud*'s #1918. The tool authors of TimeMachine explained to us that one major reason could be TimeMachine's state abstraction criterion is too coarse. In practice, TimeMachine uses a variant of the C-Lv3 abstraction criterion [2] (which only uses layout widgets to abstract GUI states) to decide whether a given state is a (new) *interesting* state. However, this abstraction criterion could be too coarse, and TimeMachine thus fails to identify and snapshot some "critical" states (which are the preconditions of the bugs) into its state pool. As a result, it may miss the chance to trigger the bug. Q-testing uses a more coarse-grained abstraction criterion (between C-Lv2 and C-Lv3 [2]), which only differentiates two GUI pages if they are from two different app usage scenarios. In fact, TimeMachine and Q-testing find the least numbers of bugs, compared to other tools.

Meanwhile, all the aforementioned three bugs can be detected by Ape, Humanoid and ComboDroid. Because ComboDroid and Humanoid use the fine-grained C-Lv4 criterion (which uses both the layout and executable widgets to abstract states), while Ape dedicatedly proposes a dynamically refined state abstraction strategy to achieve better balance between state precision and scalability.

On the other hand, Monkey is pure black-box and does not do any abstraction. It treats *every* GUI page as *unique* and emits GUI events at any random screen coordinates, and thus sometimes suffers from scalability issues. For example, Monkey cannot detect *FirefoxLite*'s #5085, which only requires 5 GUI events. The reason is that this bug requires clicking a small widget at the bottom-right corner of the first GUI page, and then clicking one specific setting option among many others on the next page. As a result, Monkey has very low chance to bypass these two pages to trigger the bug.

Opportunities: Defining appropriate state abstraction criterion is important for bug finding but still an open problem. One possible solution is to define specific granularity for specific types of apps or functionalities to reduce the chance of missing important states.

Small heuristics affect bug finding. We find some tools implemented small heuristics. *Despite these heuristics are not the fundamental advantages of the core testing techniques, they do improve the bug finding abilities.*

For example, Monkey by default does not support *long-touch*, so it cannot detect *AmazeFileManager*'s #1796, which requires a *long-touch* event. But other monkey-based tools, i.e., Ape, ComboDroid, TimeMachine found it because they implemented *long-touch.*

Table 6: Concrete new insights obtained from our study and the comparison with prior studies on these new insights ("✓", "✗" and "?" denote that the corresponding study does, does not or only partially does obtain the new insight).

Studies	RQ1 (bug finding abilities)			RQ2 (common challenges in finding bugs)					RQ3 (factors and opportunities)		
	False Negatives	Testing Stability	Testing Efficiency	C1 (event trace)	C2 (text inputs)	C3 (system/app settings)	C4 (interaction patterns)	C5 (external interactions)	Testing Strategies	State Abstraction	Small Heuristics
Choudhary *et al.* [7]	✗	✗	✗	✗	?	✗	✗	✗	✗	✗	✗
Wang *et al.* [50]	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
Zheng *et al.* [52]	✗	✗	✗	✓	?	?	✗	✓	✗	✗	✗
Behrang *et al.* [3]	✗	✗	✗	✗	?	✗	✗	✓	✗	✗	✗
Our study	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

In addition, Ape and ComboDroid implements a special strategy to input texts (one of the common user behaviors to input text): (1) long touch the target text field to select the original text, (2) clear the whole content, and then (3) input the new random text. Due to this heuristic, only Ape and ComboDroid found *MaterialFBook*'s #224, which requires a long-touch to invoke the copy-paste operation. All the other tools cannot find this bug because they input texts via directly overwriting the original text.

Some tools internally complement their core testing technique with some heuristics to improve testing effectiveness. For example, Ape and ComboDroid occasionally invoke the default Monkey to do random testing. As a result, they can trigger some bugs that are only likely to be triggered by Monkey. For example, Monkey may slide down the notification bar by random swipes and change some settings therein by random touches. As a result, all Monkey-based tools can detect *openlauncher*'s #67, which requires opening the "do not disturb" setting. Humanoid and Q-testing cannot detect this bug due to the lack of any Monkey-like random testing strategies.

Opportunities: Designing and integrating small heuristics by simulating human-app interaction patterns (*e.g.*, specific UI actions, text input styles, putting apps in the background and returning back to it) can improve bug finding.

4.4 Discussion

New insights obtained from our study. Table 6 summarizes the concrete new insights obtained from our study. We can see that most of the new insights have not been identified by the prior studies [3, 7, 50, 52]. Specifically, due to the lack of a ground-truth benchmark, the studies [7, 50] are difficult to do the in-depth analysis like RQ1~RQ3, while the studies [3, 7, 52] can only identify some or partial insights in RQ2 because they identify the tool limitations in achieving high code coverage rather than bug finding. We note that the prior studies [3, 7, 52] identified some other tool limitations like requiring account-login and collaboration with other devices. We excluded such limitations in the evaluation setup, *e.g.*, by providing auto-login scripts and focusing on "self-contained" apps, because these are not the limitations of the core testing techniques.

Applications of our study. Our study can have three major applications. *First*, the detailed analysis in RQ1~RQ3 distilled many important findings, which can help enhance, optimize and extend existing testing tools. It also pointed out some open research problems, *e.g.*, how to efficiently find system setting related crashes [38] and better balance between different GUI abstraction criteria. *Second*, the Themis benchmark can be used to quantitatively and qualitatively evaluate new testing techniques for Android in a controlled, rigorous environment like Defects4J for Java. For example, a new testing technique could compare itself with the results of selected tools to validate its effectiveness, and challenge itself with the 18 critical bugs (which no tool can find) to prove its advancement. *Third*, the infrastructure can be used to facilitate other research like bug reproducing [51], fault localization [10] and program repair for Android.

Threats to validity The validity of our study may be subject to some threats. *One threat* is the representativeness of our bug dataset and the generability of our findings. To reduce this threat, we interviewed the industrial practitioners to obtain the agreed-upon selection criterion of bugs that conforms to real industrial practices. The data collection is based on a large set of Android apps, and all the issues with critical labels are assigned by developers. We carefully inspected each issue and collect valid ones without any bias (see Section 3.1). Table 3 shows the apps are diverse, and the analysis in RQ2/RQ3 also shows the bugs have different features. Moreover, the interviewees observe that critical bugs do not have obvious differences from other less important ones in bug manifestation (*e.g.*, the difficulty of bug-triggering and the test length). Thus, our study findings based on critical bugs could be generalized to real-world bugs. In the future, we could incorporate more bugs to further mitigate this threat. *Another threat* is the correctness of evaluation and result analysis. To counter this, we made considerable effort to setup a rigorous experimental infrastructure, and resolved many tool issues before the deployment (see Section 3.2). We carefully examined tool implementations and discussed with the tool authors to analyze tool abilities and validate our observations. The experimental data and results were cross-checked by the two co-authors. We also made the Themis benchmark publicly available for replication.

5 CONCLUSION

In this paper, we take the first step to empirically evaluate automated GUI testing for Android against real-world bugs. We evaluate several testing tools on the 52 real, reproducible bugs, and reveal many new findings. We find a considerable gap in these tools finding the collected bugs. We identify five common major challenges that future work should address, and the factors that affect these tools in bug finding. Our study provides a new, complementary perspective from prior studies to analyze existing testing tools.

ACKNOWLEDGMENTS

We thank the anonymous ESEC/FSE reviewers for their valuable feedback and the generous helps from the studied tools' authors. This work is partially supported by NSFC Project No. 62072178 and No. 61690204. Ting Su and Zhendong Su were partially supported by a Google Faculty Research Award. Ting Su was also partially supported by a Swiss NSF Spark Award CRSK-2_190302.

REFERENCES

[1] Andrea Arcuri and Lionel C. Briand. 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Softw. Test., Verif. Reliab.* 24, 3 (2014), 219–250.

[2] Young Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE).* 238–249.

[3] Farnaz Behrang and Alessandro Orso. 2020. Seven Reasons Why: An In-Depth Study of the Limitations of Random Test Input Generation for Android. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1066–1077.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA).* 169–190.

[5] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of Android apps with minimal restart and approximate learning. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA).* 623–640.

[6] Wontae Choi, Koushik Sen, George C. Necula, and Wenyu Wang. 2018. DetReduce: minimizing Android GUI test suites for regression testing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE).* 445–455.

[7] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 429–440.

[8] Brendan Dolan-Gavitt, Patrick Hulin, Engin Kirda, Tim Leek, Andrea Mambretti, William K. Robertson, Frederick Ulrich, and Ryan Whelan. 2016. LAVA: Large-Scale Automated Vulnerability Addition. In *IEEE Symposium on Security and Privacy (SP).* 110–121.

[9] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel Testing of Android Apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE).* 481–492.

[10] Lingling Fan, Ting Su, Sen Chen, Guozhu Meng, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE).* 408–419.

[11] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 141–152.

[12] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *Proceedings of the 41st International Conference on Software Engineering (ICSE).* 269–280.

[13] Cuixiong Hu and Iulian Neamtiu. 2011. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST).* 77–83.

[14] Laura Inozemtseva and Reid Holmes. 2014. Coverage is not strongly correlated with test suite effectiveness. In *36th International Conference on Software Engineering (ICSE)*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). 435–445.

[15] Wang Jue, Jiang Yanyan, Xu Chang, Cao Chun, Ma Xiaoxing, and Lu Jian. 2020. ComboDroid: Generating High-Quality Test Inputs for Android Apps via Use Case Combinations. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE).* 469–480.

[16] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA).* 437–440.

[17] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating Fuzz Testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS).* 2123–2138.

[18] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *8th International Conference on Software Testing, Verification and Validation (ICST).* IEEE, 1–10.

[19] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé, and Jacques Klein. 2019. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Trans. Reliability* 68, 1 (2019), 45–66.

[20] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering (ICSE).* 23–26.

[21] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: A Deep Learning-Based Approach to Automated Black-box Android App Testing. In *34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1070–1073.

[22] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. 2017. Automatic text input generation for mobile testing. In *Proceedings of the 39th International Conference on Software Engineering (ICSE).* 643–653.

[23] Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. 2019. Preference-wise testing for Android applications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).* 268–278.

[24] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE).* 224–234.

[25] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective Automated Testing for Android Applications. In *International Symposium on Software Testing and Analysis (ISSTA).* 94–105.

[26] Michaël Marcozzi, Qiyi Tang, Alastair F. Donaldson, and Cristian Cadar. 2019. Compiler fuzzing: how much does it matter? *Proc. ACM Program. Lang.* OOPSLA (2019), 155:1–155:29.

[27] Atif M. Memon and Myra B. Cohen. 2013. Automated testing of GUI applications: models, tools, and controlling flakiness. In *35th International Conference on Software Engineering (ICSE)*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). 1479–1480.

[28] Daniel Miessler. 2021. *Big list of naughty strings.* Retrieved 2021-1 from https://github.com/danielmiessler/SecLists/blob/master/Fuzzing/big-list-of-naughty-strings.txt

[29] Monkey. 2020. *Monkey.* Retrieved 2020-5 from http://developer.android.com/tools/help/monkey.html

[30] Ana C. R. Paiva, Joao M. E. P. Gouveia, Jean-David Elizabeth, and Márcio E. Delamaro. 2019. Testing When Mobile Apps Go to Background and Come Back to Foreground. In *2019 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 102–111.

[31] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* 153–164.

[32] Priyam Patel, Gokul Srinivasan, Sydur Rahaman, and Iulian Neamtiu. 2018. On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In *Proceedings of the 13th International Workshop on Automation of Software Test (AST).* 34–37.

[33] Danilo Dominguez Perez and Wei Le. 2019. Testing Criteria for Mobile Apps Based on Callback Sequences. *CoRR* abs/1911.09201 (2019).

[34] John Regehr. 2021. *Responsible and Effective Bugfinding.* Retrieved 2021-1 from https://blog.regehr.org/archives/2037

[35] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. 2017. Patdroid: permission-aware gui testing of android. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE).* 220–232.

[36] Ting Su, Lingling Fan, Sen Chen, Yang Liu, Lihua Xu, Geguang Pu, and Zhendong Su. 2020. Why My App Crashes Understanding and Benchmarking Framework-specific Exceptions of Android apps. *IEEE Transactions on Software Engineering* (2020).

[37] 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 *The joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE).* 245–256.

[38] Jingling Sun, Ting Su, Junxin Li, Zhen Dong, Geguang Pu, Tao Xie, and Zhendong Su. 2021. Understanding and Finding System Setting-Related Defects in Android Apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA).* to appear.

[39] Shin Hwei Tan, Zhen Dong, Xiang Gao, and Abhik Roychoudhury. 2018. Repairing crashes in Android apps. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*. 187–198.

[40] Android team. 2021. *Android Developers Documentation*. Retrieved 2021-1 from https://developer.android.com/

[41] Android team. 2021. *Android Help*. Retrieved 2021-1 from https://support.google.com/android/

[42] F-Droid Team. 2021. *F-Droid*. Retrieved 2021-1 from https://f-droid.org/

[43] GitHub Team. 2021. *GitHub REST API*. Retrieved 2021-1 from https://docs.github.com/en/rest/

[44] Jacoco Team. 2021. *Jacoco*. Retrieved 2021-1 from https://www.eclemma.org/jacoco/

[45] uiautomator2 Team. 2021. *uiautomator2*. Retrieved 2021-1 from https://github.com/openatx/uiautomator2/

[46] Swapna Thorve, Chandani Sreshtha, and Na Meng. 2018. An Empirical Study of Flaky Tests in Android Apps. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 534–538.

[47] Porfirio Tramontana, Domenico Amalfitano, Nicola Amatucci, and Anna Rita Fasolino. 2019. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal* 27, 1 (2019), 149–201.

[48] Mario Linares Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do Developers Test Android Applications?. In *International Conference on Software Maintenance and Evolution (ICSME)*. 613–622.

[49] Mario Linares Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. In *International Conference on Software Maintenance and Evolution (ICSME)*. 399–410.

[50] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 738–748.

[51] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G. J. Halfond. 2019. ReCDroid: automatically reproducing Android application crashes from bug reports. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. 128–139.

[52] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, 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 *39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. 253–262.