Towards Benchmarking the Coverage of Automated Testing Tools in Android against Manual Testing

Ferdian Thung
ferdianthung@smu.edu.sg
School of Computing and Information Systems
Singapore Management University, Singapore

Ivana Clairine Irsan
ivanairsan@smu.edu.sg
School of Computing and Information Systems
Singapore Management University, Singapore

Jiakun Liu
jkliu@smu.edu.sg
School of Computing and Information Systems
Singapore Management University, Singapore

David Lo
davidlo@smu.edu.sg
School of Computing and Information Systems
Singapore Management University, Singapore

ABSTRACT

Android apps are commonly used nowadays as smartphones have become irreplaceable parts of modern lives. To ensure that these apps work correctly, developers would need to test them. Testing these apps is laborious, tedious, and often time consuming. Thus, many automated testing tools for Android have been proposed. These tools generate test cases that aim to achieve as much code coverage as possible. A lot of testing methodologies are employed such as model-based testing, search-based testing, random testing, fuzzing, concolic execution, and mutation. Despite much efforts, it is not perfectly clear how far these testing tools can cover user behaviours. To fill this gap, we want to measure the gap between the coverage of automated testing tools and manual testing. In this preliminary work, we selected a set of 11 Android apps and ran state-of-the-art automated testing tools on them. We also manually tested these apps by following a guideline on actions that we need to exhaust when exploring the apps. Our work highlights that automated tools need to close some gaps before they can achieve coverage that is comparable to manual testing. We also present some limitations that future automated tools need to overcome to achieve such coverage.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

ACM Reference Format:
Ferdian Thung, Ivana Clairine Irsan, Jiakun Liu, and David Lo. 2024. Towards Benchmarking the Coverage of Automated Testing Tools in Android against Manual Testing. In *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MOBILESoft '24), April 14–15, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3647632.3651394

MOBILESoft '24, April 14–15, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0594-6/24/04
https://doi.org/10.1145/3647632.3651394

1 INTRODUCTION

Android is a popular mobile OS for smartphones. As such, many apps have been developed within its ecosystem. As of 2023, Google Play, the official Android app store, contains 3.718 millions apps available for users to download [3]. The store covers 33 categories spanning from Art and Design to Weather. Such popularity could be attributed to the open source nature of Android, which allows smartphone companies to adopt Android OS in their smartphones. The adoption allows Android to reach 3.6 billions users worldwide [2].

Due to Android's market size and competition, app developers need to maintain a high quality standard for their apps so that they can be successful commercially in the market [13, 15]. To ensure the quality, developers need to perform software testing that exercise the app functionalities [14]. However, testing the app manually is often laborious, tedious, and time-consuming. Therefore, many automated testing tools have been proposed to ease the work of testing Android apps. Techniques such as model-based testing, fuzzing, concolic execution, and mutation have been used to assist in generating test cases that can achieve a high code coverage for Android apps [9].

Despite the many automated testing tools available for Android apps, it is not very clear how they actually fare when compared to manual testing. Particularly, other than the obvious advantage of scalability, it is not apparent whether the automated tools can reach a similar coverage as manual testing. Such comparison is important since careful manual testing can arguably emulate user behavior, given the fact that testers are app users themselves. To fill this gap, we want to measure the coverage difference between automated testing tools and manual testing.

In order to measure the coverage difference, we selected 11 Android apps for preliminary experiments. We manually tested these apps by following a guideline we created. The guideline contains the list of actions that we would need to execute when trying to explore the apps. The guideline aims to ensure that we can explore the apps consistently and exhaustively. We then also tested the apps using available state-of-the-art automated testing tools for Android apps. We measure the coverage achieved for both manual and automated testing.

Our preliminary experiments highlight that manual testing could still achieve a much higher coverage when compared to automated testing tools. Manual testing on the 11 apps achieve an average code coverage of 65.78% while best automated testing tool we evaluated

can only achieve the average code coverage of 29.36%. Our analysis on the results indicate that the automated testing tools were often stuck when exploring certain types of content. Chief among them is advertisement; for which automated testing tools typically would need to wait for a countdown to zero so that they can close the advertisement before they can continue to explore the app.

The contributions of this paper are:

- We compare the coverage of automated testing tools in Android with manual testing.
- In our preliminary experiments, we found that manual testing can produce a better coverage than automated testing. This is partly due to the tools inability to deal with advertisement.

The remainder of this paper is structured as follows. We describe our testing methodologies in Section 2. We then describe our dataset, experimental settings, and the results of our experiments in Section 3. We present related work in Section 4. Finally, we conclude the paper and mention some future work in Section 5.

2 TESTING METHODOLOGIES

2.1 Automated Testing Tools

We selected automated testing tools based on its popularity, recency, and availability. We exclude tools that are recent but unavailable as we would be unable to run it (e.g., Columbus [6] is one of the most recent and state-of-the-art work in automated Android app testing but the source code is not available in the provided repository[1]. We have emailed the first and the second authors but did not receive any response.). The following are the automated testing tools we selected.

2.1.1 Sapienz [12]. It is a popular automated Android app testing tool. It has been adopted by Facebook to automate its app testing at scale [4]. It utilizes a Pareto-optimal multi-objective search-based approach to optimize test sequences, with a goal to generate short sequences that maximize code coverage and fault revelation. It uses fuzzing, systematic, and search-based exploration. It was shown to significantly outperform existing tools such as Dynodroid [11] and Android Monkey [5]. It also successfully identified 558 unique and previously unknown crashes.

2.1.2 ARES [17]. It is a state-of-the-art Android app testing tool. It utilizes reinforcement learning (RL) to learn optimal exploration strategies in terms of coverage and fault revelation. Like any deep learning algorithms, it comes with a set of parameters to tune. It additionally comes with a several choices of exploration strategies such as random exploration and Q-Learning. To speed up the parameter tuning and the selection of exploration strategy, it utilizes a component called FATE. FATE allows for a rapid assessment of exploration strategy and parameter choice by running them on a synthetic version of the original app. The synthetic app keeps only the navigational parts of the app. On average, testing using FATE is 10-100 times faster than testing on the original app. Comparison with state-of-the-art Android app testing tools (e.g., TimeMachine [8] and Q-Testing [16]) shows that ARES can achieve a higher coverage and fault revelation.

[1]https://github.com/ucsb-seclab/columbus

2.2 Manual Testing

Careful manual testing has a good potential to simulate user behaviours. However, manual testing may also be error-prone. To minimize possible errors that we may make when testing an Android app, we create a guideline to ensure that we exhaustively explore the app consistently. We navigate to all activities (i.e., screens) inside the app and performs all possible actions on all the interactable elements within an activity. We also interact with the app notifications in the notification bar. We consider the following actions and provides examples of their common usages.

(1) *Tap.* It can be used to press a button and select items.
(2) *Double tap.* It can be used to zoom in or select items.
(3) *Long press.* It can be used to move objects or display a context menu that contains a list of app-specific commands that can be executed.
(4) *Swipe (left/right/up/down).* It can be used to navigate between screen pages and to scroll a page.
(5) *Twirl.* It can be used to spin selected objects.
(6) *Pinch.* It can be used to zoom in/out selected objects.
(7) *Type.* It can be used to provide a text input.
(8) *Press the Android buttons (Home/Overview/Back).* Pressing Home button returns the user to the Android Home screen. Pressing Overview button shows the list of opened apps. Pressing Back button returns the user to the previous screen (if any).
(9) *Change device pose/orientation.* It can be used to change an app content orientation to portrait/landscape when the screen is rotated (if supported by the app).

3 PRELIMINARY EXPERIMENTS

3.1 Dataset

We collected Android apps from the list of popular apps in Android Rank website[2] on April 2023. This website provides a list of top apps in Google Play covering 33 categories such as Entertainment, Photography, Shopping, and Finance. Since Google Play does not web crawlers to download apps, we downloaded the latest version for each app from APKCombo[3], which provides apps downloaded from Google Play.

We selected apps with the following criteria. Since we need to perform manual testing, we limit the app size based on the number of application methods. We define application methods as the methods that starts with the package name of the app. We consider only apps with application methods between 50 and 1,000 (both inclusive). We also only select apps that supports Android KitKat (API 19) since Sapienz mainly supported this version. It did not work on some new Android versions we tried. We exclude apps that fail to be installed in an Android emulator. We also exclude apps that Sapienz or ARES fail to run. In the end, we have 11 apps for preliminary experiments. These apps have an average number of methods of 302.27, ranging from 85 to 605 methods. They have an average size of 12.36 MB, ranging from 2.77 to 28.93 MB.

[2]https://www.androidrank.org/
[3]https://apkcombo.com/

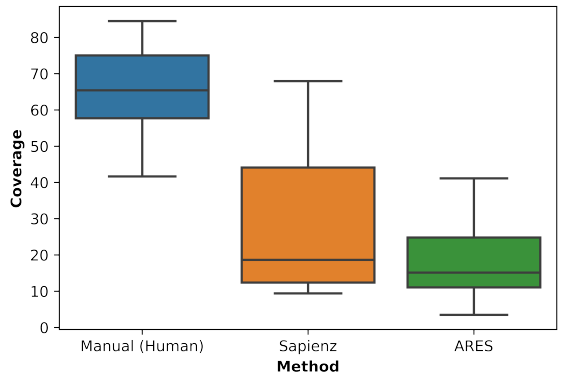

Figure 1: Coverage of automated and manual testing

3.2 Experimental Settings

Before running either the automated or manual testing, we instrument the app so that we can record app elements that have been executed. The percentage of executed elements across all elements is the code coverage. We use ELLA [1] to instrument the app. It is the only publicly available tools that we found that can work out-of-the-box to measure code coverage of binary Android apps (i.e., apk). ELLA reports the code coverage in terms of method coverage.

We run both automated and manual testing on an Android emulator. We run Sapienz and ARES with an hour timeout. This is the same number of timeout used by ARES in its paper and twice the number of timeout used by Sapienz in its paper. Our experiments were done on AMD EPYC 7643 @ 2.3GHz machine. We use default settings when running Sapienz and ARES. Meanwhile, manual testing were performed by the first and the second authors. Each author tests different sets of apps.

3.3 Results

Figure 1 shows the coverage of Sapienz, ARES, and manual testing. It is apparent that manual testing produces a much higher coverage when compared to both Sapienz and ARES, with Sapienz performing better than ARES. Manual testing produces a coverage of 65.78%, while Sapienz and ARES produces a coverage of 29.36% and 18.27%, respectively. It means that manual testing has 124.01% and 260.04% higher coverage when compared to Sapienz and ARES, respectively.

Upon a closer inspection, we found that Sapienz and ARES tend to have a lower coverage because they are often stuck when interacting with certain types of elements. We observe their executions on the Android emulator and found that they have difficulties of interacting with the following elements[4].

(1) *Advertisement.* Many Android apps are free and often monetize via advertisement. Advertisement can appear anytime we navigate within the app. When it appears, we often need to wait for a countdown to zero before we can close the advertisement and continue to explore the app. The evaluated automated tools often click on the advertisement and thus unable to continue to the screen beyond the corresponding advertisement.

[4]The list of elements we discovered is purely based on our preliminary observation and by no means exhaustive.

(2) *Constrained input.* Some inputs are constrained and expects a specific format before they can be accepted. For example, email address need to be input correctly in a form before we can continue to the next screen. The evaluated utomated tools cannot capture these semantics.
(3) *Paywall.* Some app features are hidden behind a paywall. We need to pay before any testing approaches can work. Some apps allow usage of paid app features by watching advertisement, where manual testing can work without paying.

4 RELATED WORK

Many approaches have been developed to automate Android apps testing [6, 8, 12, 16, 17]. Sapienz [12] performs a multi-objective search-based approach to generate test sequences that are short and maximize both coverage and fault revelation. TimeMachine [8] proposes a search-based testing that evolves a population of states instead of input sequences. The states should be able to be captured when discovered and can be resumed when needed. Q-Testing [16] mixes random and model-based approach to test Android apps. It utilizes curiosity-driven strategy to guide testing towards unfamiliar functionalities. ARES [17] is a reinforcement-learning based automated testing tool that aims to learn the optimal exploration strategies for an app. Columbus [6] is a callback-driven testing technique that combines under-constrained symbolic execution and type-guided dynamic heap introspection to generate inputs. To maximize coverage and triggered crashes, it uses data dependency and crash guidance as its feedback mechanisms. In this work, we evaluate Sapienz due to its popularity and usage in Facebook. On the other hand, we evaluate ARES due to its recency and availability.

Some work track the progression of automated testing tools in Android [7, 9, 10]. Linares et al. [10] surveys the state of mobile testing at the time (Android and iOS), in terms of frameworks, tools, and services available. Kong et al. [9] performs a systematic literature study specifically on Android app testing. Choudhary et al. [7] performs a thorough comparison of main automated testing tools for Android at the time. Our work belong to this line of work. Different than previous work that only compared automated tools among themselves, we assess how good the coverage of automated testing tools is when compared with manual testing.

5 CONCLUSION AND FUTURE WORK

Millions of Android apps can be downloaded from Google Play. Many tools have been proposed to automatically test these apps. However, whether the coverage of these tools are better than manual testing is unknown. Our experiments on 11 apps show that manual testing can achieve a better coverage than automated testing tools. To close the coverage gap, the automated tools should be capable of, among other things, mimicking humans to understand advertisement page, constrained input, paying by watching advertisement, etc. In the future, we plan to extend this preliminary work to investigate more apps. We plan to also extend the evaluation of automated tools so that we can exhaustively discover all elements that these tools have difficulty interacting with. We also plan to improve the coverage of automated tools by developing solutions based on AI foundational models, which can mimick humans and understand semantics.

ACKNOWLEDGEMENTS

This research / project is supported by the National Research Foundation, Singapore, and Cyber Security Agency of Singapore under its National Cybersecurity Research and Development Programme, NCRP25-P03-NCR-TAU. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

REFERENCES

[1] [n. d.]. ELLA: A Tool for Binary Instrumentation of Android Apps. https://github.com/saswatanand/ella. Accessed: 2024-01-04.

[2] [n. d.]. How Many Android Users Are There? Global and US Statistics (2024). https://www.bankmycell.com/blog/how-many-android-users-are-there. Accessed: 2024-01-04.

[3] [n. d.]. How Many Apps In Google Play Store? (Jan 2024). https://www.bankmycell.com/blog/number-of-google-play-store-apps/. Accessed: 2024-01-04.

[4] [n. d.]. Sapienz: Intelligent automated software testing at scale. https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/. Accessed: 2024-01-04.

[5] [n. d.]. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/other-testing-tools/monkey. Accessed: 2024-01-04.

[6] Priyanka Bose, Dipanjan Das, Saastha Vasan, Sebastiano Mariani, Ilya Grishchenko, Andrea Continella, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. 2023. COLUMBUS: Android App Testing Through Systematic Callback Exploration. In *Proceedings of the International Conference on Software Engineering (ICSE)*. 1381–1392. https://doi.org/10.1109/ICSE48619.2023.00121

[7] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 429–440.

[8] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 481–492.

[9] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability* 68, 1 (2018), 45–66.

[10] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 399–410.

[11] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[12] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*. 94–105.

[13] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. 2016. A survey of app store analysis for software engineering. *IEEE transactions on software engineering* 43, 9 (2016), 817–847.

[14] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. 2004. *The art of software testing*. Vol. 2. Wiley Online Library.

[15] Fabio Palomba, Mario Linares-Vásquez, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2018. Crowdsourcing user reviews to support the evolution of mobile apps. *Journal of Systems and Software* 137 (2018), 143–162.

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

[17] Andrea Romdhana, Alessio Merlo, Mariano Ceccato, and Paolo Tonella. 2022. Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 4 (2022), 1–29.