An Empirical Comparison between Monkey Testing and Human Testing (WIP Paper)

Mostafa Mohammed
Virginia Tech
Blacksburg, Virginia, United States
profmdn@vt.edu

Haipeng Cai
Washington State University
Pullman, Washington, United States
haipeng.cai@wsu.edu

Na Meng
Virginia Tech
Blacksburg, Virginia, United States
nm8247@vt.edu

Abstract

Android app testing is challenging and time-consuming because fully testing all feasible execution paths is difficult. Nowadays apps are usually tested in two ways: human testing or automated testing. Prior work compared different automated tools. However, some fundamental questions are still unexplored, including (1) how automated testing behaves differently from human testing, and (2) whether automated testing can fully or partially substitute human testing.

This paper presents our study to explore the open questions. Monkey has been considered one of the best automated testing tools due to its usability, reliability, and competitive coverage metrics, so we applied Monkey to five Android apps and collected their dynamic event traces. Meanwhile, we recruited eight users to manually test the same apps and gathered the traces. By comparing the collected data, we revealed that i.) on average, the two methods generated similar numbers of unique events; ii.) Monkey created more system events while humans created more UI events; iii.) Monkey could mimic human behaviors when apps have UIs full of clickable widgets to trigger logically independent events; and iv.) Monkey was insufficient to test apps that require information comprehension and problem-solving skills. Our research sheds light on future research that combines human expertise with the agility of Monkey testing.

CCS Concepts • **General and reference → Empirical studies**.

Keywords Empirical, Monkey testing, human testing

ACM Reference Format:
Mostafa Mohammed, Haipeng Cai, and Na Meng. 2019. An Empirical Comparison between Monkey Testing and Human Testing (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3316482.3326342

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
LCTES '19, June 23, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6724-0/19/06...$15.00
https://doi.org/10.1145/3316482.3326342

1 Introduction

As Android devices become popular, many new apps are built every day. In year 2016 alone, developers published 1.3 million new Android apps on Google Play [1], which can be translated to 3,561 new apps every day. Traditionally, software professionals test such apps by randomly clicking buttons or exploring different features shown in the User Interfaces (UIs). As new apps significantly increase day-by-day, it is almost infeasible for humans to quickly test all apps or reveal all of the scenarios when app failures occur.

Tools were built to automatically test Android apps [4, 5, 10, 11, 14, 16, 17, 22, 23]. For instance, Monkey treats an Android app as a black box, and randomly generates UI actions (e.g., click and touch) as test inputs. GUIRipper [10] dynamically builds a GUI model of the app under testing to trigger GUI events in a depth-first search (DFS) manner [10]. ACTEve uses concolic testing to iteratively (1) create an event based on symbolic execution, (2) monitor any concrete execution triggered by the event, and (3) negate some path constraints to cover more execution paths.

Choudhary et al. conducted an empirical study to compare 14 automated testing tools [13]. The researchers showed that Monkey is the best because it (1) achieves the highest code coverage, (2) triggers the most software failures within a time limit, (3) is easy to use, and (4) is compatible with any Android framework version. Patel et al. further characterized the effectiveness of random testing by Monkey in five aspects (e.g., stress testing and parameter tuning), and reported that Monkey is on par with manual exploration in terms of code coverage at various granularity levels (e.g., block, statement) [19]. Machiry et al. showed that Monkey's coverage of UI events is comparable to that of the peer tools [16]. Nevertheless, ***it is still unknown how automated tools test Android apps differently from human testers***.

This paper is intended to empirically contrast automated testing with human testing. Prior work shows that Monkey generally works at least as effectively as other automated tools [13, 18, 20], so we used Monkey as a representative automated testing tool. Specifically, we chose five Android apps from different domains, and executed the apps with Monkey.

Meanwhile, we recruited eight developers to manually test these apps. With prior work Droidfax [12], we instrumented each app to log any UI event (e.g., `onClick()`), lifecycle event (e.g., `onCreate(...)`), or system event (e.g., `onLocationChanged()`) triggered in the execution. In this way, we collected the event traces for Monkey testing and human testing, and compared the traces. We investigated three research questions:

RQ1: Does Monkey always trigger more diverse events when it runs an app for a longer time?
RQ2: Does Monkey trigger more or fewer unique events than humans within the same period of time?
RQ3: Can Monkey produce certain events more or less effectively than humans?

Our experiments show that there is no correlation between Monkey's event coverage and execution time. As execution time increases, Monkey inputs triggered more or fewer unique events between different runs of the same app. Additionally, we applied both Monkey testing and human testing to the same set of apps for the same periods of time (e.g. 15 minutes). Although humans were slower to generate test inputs, we observed no significant difference between the numbers of unique events triggered by different methods. Furthermore, we found Monkey to effectively trigger more system events but fewer UI events than humans, probably because Monkey has no domain knowledge of the app usage.

Our study corroborates some of the observations by prior work [13, 19]. More importantly, we quantitatively and qualitatively analyzed the behavioral differences between Monkey and humans. Our findings not only provide insights on how developers should test apps to complement Monkey testing, but also shed lights on future research that integrates human expertise into automated testing.

2 Methodology

For our study, we selected five Android apps from different application domains. These apps are:

- Amazon [2]: an online shopping app;
- Candy Crush [3]: a game app to earn scores by moving, matching, and crushing candies;
- Spotify [7]: an entertainment app to listen to music;
- Twitter [8]: a social media app for users to broadcast text messages and live streams; and
- Viber [9]: a call app for people to make free phone calls or send messages through the Internet.

The idea behind selecting these apps is to compare Monkey testing and human testing in different application scenarios.

Monkey Testing. We used Monkey to test the apps. Monkey ran each app five times with different lengths of execution time (i.e., 15, 20, 25, 30, and 60 minutes).

Human Testing. We invited eight students at Virginia Tech to use the apps. These students include five CS PhDs, an ECE PhD, a CS Master, and a CS Bachelor. To thoroughly compare Monkey and humans in various contexts, we intentionally asked the users to test apps in the following way (see Table 1). We required each user to spend in total two hours in testing all apps to limit their time contribution. However, to diversify the human-testing periods of each app, we asked each user to test 1 app for 15 minutes, 1 app for 20 minutes, 1 app for 25 minutes, and 2 apps each for 30 minutes.

Table 1. Minutes spent by each user to test the five apps

	Candy Crush	Amazon	Twitter	Spotify	Viber
User 1	30	15	20	25	30
User 2	15	20	25	30	30
User 3	30	15	20	25	30
User 4	30	30	15	20	25
User 5	25	30	30	15	20
User 6	20	25	30	30	15
User 7	15	20	25	30	30
User 8	30	30	15	20	25

Droidfax [12]. When Monkey or humans tested Android apps, we used an existing tool—Droidfax—to instrument apps and to generate execution traces. Given an app's apk file, Droidfax instruments the app's Dalvik bytecode and collects various dynamic execution information, such as triggered events, invoked methods, and involved Inter-Component Communications (ICCs). We used Droidfax's event tracing feature to log all triggered events, including UI events, lifecycle events, and system events. UI events (e.g., `onClick(...)`) are directly triggered by users' operations (e.g., button click) to use apps. Lifecycle events (e.g., `onPause()`) are triggered when an app's component has its status changed (e.g., stopped). System events are produced when any component of the Android system has its status changed (e.g., `onNetworkActive()`).

After instrumenting the apps with Droidfax, we installed the instrumented apps to an Android emulator [6] on a Linux laptop. When an instrumented app was tested by Monkey or a human, we used the command "*adb logcat*" to save log information to a file. For any run of any app, Droidfax created an event-trace log file. We analyzed these log files to investigate our research questions.

3 Major Findings

We present our investigation results for the research questions separately in Section 3.1-3.3.

3.1 Correlation between Monkey's Event Coverage and Testing Time

Table 2 reports the number of unique events covered by different runs of Monkey testing, with each cell corresponding to one Monkey testing run. From the table, **we do not see any correlation between Monkey's event coverage and testing time**. As the time period increases, Monkey testing triggered more or fewer unique events. For instance, the 20-minute run of Candy Crush covered 24 unique events, which number is larger than the number of events produced by any other run of the same app. Among these 24 events,

there were 2 events never triggered by any of the other 4 runs: `onBind(...)` and `onReceive(...)`.

Table 2. Numbers of unique events covered by different runs of Monkey testing

Time Span	Candy Crush	Amazon	Twitter	Spotify	Viber	Average
15 min	18	36	52	38	56	40
20 min	24	45	48	35	56	42
25 min	23	40	45	34	63	41
30 min	22	47	48	36	63	43
60 min	22	44	55	45	57	45

3.2 Event Coverage Comparison between Monkey Testing and Human Testing

Table 3 presents the average numbers of unique events covered by different runs of human testing. If one app was tested by multiple users for the same period of time (e.g., 30 minutes), we reported the average number of unique events covered in the multiple log files. If an app was tested by only one user for a certain period of time (e.g., only User 1 ran Amazon for 15 minutes), we reported the number of unique events covered by the single log file.

Table 3. Average numbers of unique events covered by different runs of human testing

Time Span	Candy Crush	Amazon	Twitter	Spotify	Viber	Average
15 min	26	46	51	39	43	41
20 min	15	46	48	39	54	40
25 min	17	38	49	36	54	39
30 min	31	41	52	37	57	44

Similar to the observation in Section 3.1, we also found that **there was no correlation between humans' event coverage and testing time**. As the time period increased, human testing triggered more or fewer unique events. One possible reason is that humans tested apps in a random way. Although users tried to investigate as many features as possible for an app, sometimes their investigation was still limited to a subset of the features to test. Consequently, longer testing time does not necessarily lead to higher coverage.

Comparing Table 2 and Table 3, we found that when testing the same app for the same period of time, Monkey covered more or fewer unique events than humans. To decide whether one approach outperforms the other in general, we conducted Student's t-test [15] to compare the event coverage metrics of both approaches for each app. Table 4 shows our comparison results. Mean $\Delta = Mean_{Monkey} - Mean_{human}$. To calculate **Mean** Δ for each app, we first computed the mean event coverage of each approach for each app, and then deducted $Mean_{human}$ from $Mean_{Monkey}$. **p-value** is a probability ranging in [0, 1]. Its value reflects whether the distributions of two data groups are significantly different. At the significance level of $\alpha = 0.05$, if p-value ≤ 0.05, the mean difference is significant.

Table 4. Mean event coverage comparison

	Candy Crush	Amazon	Twitter	Spotify	Viber
Mean Δ	-0.5	-0.75	-1.75	-2.0	7.5
p-value	0.9045	0.8210	0.3434	0.1289	0.0881

According to Table 4, we observed that all distribution differences are insignificant. On average, human testing covered more events than Monkey testing for four apps (i.e., Candy Crush, Amazon, Twitter, and Spotify) and Monkey testing covered more for one app (i.e., Viber). All p-values are greater than 0.05. This implies that **Monkey testing works indistinguishably differently from human testing with respect to event coverage**.

3.3 Event Frequency Comparison between Monkey Testing and Human Testing

To further compare how frequently Monkey and humans generated different kinds of events, we classified events into three categories: lifecycle events, UI events, and system events. We clustered the recorded events accordingly for each log file. Suppose a file contains N_1 lifecycle events, N_2 UI events, and N_3 system events, we further normalized the event frequencies by computing the ratio of each type of events among all logged events, i.e.,

$$ratio_i = \frac{N_i}{(N_1 + N_2 + N_3)}, (i \in [1, 3]).$$

In this way, we could uniformly compute the average event frequency distribution of each approach. As shown in Figure 1, **M** bars correspond to Monkey testing, while **H** bars are for human testing. Each stacked bar usually has three parts to manifest the ratios of lifecycle, UI, and system events. We observed three phenomena in Figure 1.

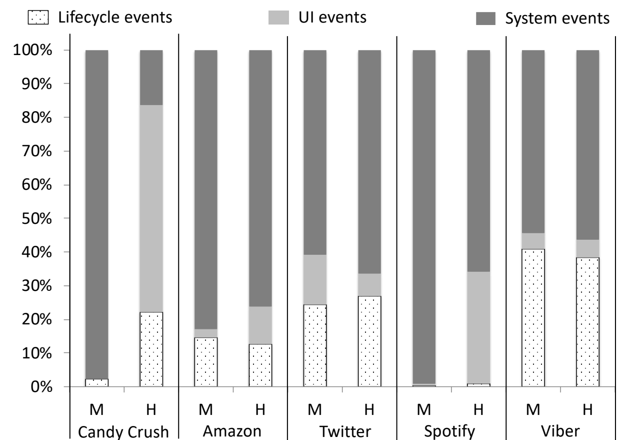

Figure 1. Normalized event frequency comparison

First, in most scenarios, system events were the most frequently generated events. To trigger such events, users may check for or modify the phone's status or settings (e.g., GPS and wifi). System events dominated the events triggered by Monkey, taking up to 54-99% of the overall generated events among the five apps. Although such events were also

the dominant events triggered by humans for four apps, we observed an exception for Candy Crush. As Candy Crush is an interaction-intensive video game, it requires users to (1) first respond to pop-up dialogs by selecting or entering information, and (2) frequently use the mouse to move objects.

Figure 2 presents the first UI to which users need to respond by clicking the "Play!" button. Although such testing behaviors look naïve to humans, they seem not well supported by Monkey testing. Two possible reasons can explain such insufficiency. First, Monkey cannot read dialogs, neither can it smartly make decisions or enter necessary information to proceed in the game. Second, Monkey does not know the game rules. It does not recognize candies, so it cannot move mouse around to drag-and-drop candies. Consequently, Monkey cannot play the game for testing.

Figure 2. Snapshot of Candy Crush's starting UI

Second, Monkey usually generated lower percentages of UI events. Among the five apps, Monkey triggered 0.1-14.7% UI events, while humans triggered 5.4-61.6% such events. In our categorization, UI events include any user's interaction with the UI elements like menu bars, dialogs, media players, views, and various widgets (e.g., information widget). As a general-purpose random UI exerciser, Monkey does not intelligently locate any operable UI element [21]; instead, it randomly tries different pixels in a screen.

When the business logic of different apps varies a lot, the UI elements related to application logic have various layouts. However, the UI elements related to system configurations usually have simpler and fixed layouts. Between the two types of UIs, it seems easier for Monkey to "accidentally" trigger system events than application-specific UI events. For example, we observed that Monkey frequently succeeded in opening and manipulating a panel of shortcuts for system settings. Such operations could produce many system events.

Third, Monkey could mimic human behaviors for certain apps like Amazon, Twitter, and Viber. In particular for Viber, Monkey obtained the percentages of lifecycle, UI, and system events as 40.8%, 4.8%, and 54.4%; while humans obtained quite close percentages: 38.3%, 5.4%, 56.3%. For Twitter, Monkey's event distribution is: 24.5%, 14.7%, and 60.8%; while humans' event distribution is: 26.8%, 7.0%, and 66.3%. For Amazon, the distributions are less similar. Monkey's event distribution is 14.5%, 2.5%, and 83.0%; and humans' average event distribution is 12.6%, 11.1%, and 76.2%.

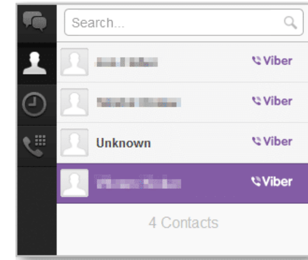

Figure 3. Snapshot of Viber's contact UI

Two possible reasons can explain why Monkey worked as effectively as humans for these apps. First, with lots of clickable items in each UI, these apps are simple to navigate. For instance, the Viber app only supports users to call or message people. It has a few UIs. Especially in the contact UI shown in Figure 3, almost every pixel corresponds to a clickable UI element. Therefore, even though Monkey knows nothing about the UI, its actions on different pixels can always effectively trigger events. Second, more importantly, there is no implicit constraint on the sequence of user actions. No matter how random the generated actions are, Monkey can always successfully trigger events and make progress.

Monkey worked less effectively than humans when testing Candy Crush and Spotify. As mentioned in Section 3.1, Candy Crush requires users to first click the "Play!" button to start the game. If Monkey keeps wasting time clicking pixels irrelevant to the button or pressing keys, it cannot even start the game for testing. Similarly, the Spotify app requires users to first click a button "Play" to listen to a selected song, and then to navigate the Media Player buttons (e.g., "Fast Forward") to control how music is played. Such button-clicking sequence constraints are intuitive to humans, but unknown to Monkey. Therefore, Monkey did not work well for apps requiring for (1) UI element recognition, and (2) action sequences following certain patterns.

4 Conclusion

We observed that Monkey is good at testing simple apps (1) with UIs full of operable widgets and (2) requiring no sequential ordering between any events triggered in the same UI. Developers can apply Monkey testing to simple apps multiple times to explore distinct usage scenarios. Monkey is not good at testing complex apps that require for domain knowledge and decision making. Developers can focus their manual effort on such complex apps. In the future,while humans test complex apps, we will instrument apps to gather the user inputs and triggered events. By inferring the cause-effect relationship between inputs and events, we can further extract app-specific usage patterns, generalize those patterns, and build automated tools to better mimic human testers.

Acknowledgments

We thank anonymous reviewers for their valuable comments on the earlier version of our paper.

References

[1] 2017. App Stores Start to Mature – 2016 Year in Review. https://blog.appfigures.com/app-stores-start-to-mature-2016-year-in-review/. (2017).

[2] 2018. Amazon Shopping APK. https://apkpure.com/cn/amazon-shopping/com.amazon.mShop.android.shopping. (2018).

[3] 2018. Candy Crush APK. https://apkpure.com/cn/candy-crush-saga/com.king.candycrushsaga. (2018).

[4] 2018. Google. Android Monkey. http://developer.android.com/tools/help/monkey.html. (2018).

[5] 2018. Intent fuzzer. https://www.nccgroup.trust/us/about-us/resources/intent-fuzzer/. (2018).

[6] 2018. Run Apps on the Android Emulator. https://developer.android.com/studio/run/emulator.html. (2018).

[7] 2018. Spotify Music APK. https://apkpure.com/cn/spotify-premium-music/com.spotify.music. (2018).

[8] 2018. Twitter APK. https://apkpure.com/cn/twitter/com.twitter.android. (2018).

[9] 2018. Viber Messenger APK. https://apkpure.com/cn/viber-messenger/com.viber.voip. (2018).

[10] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*. ACM, New York, NY, USA, 258–261. https://doi.org/10.1145/2351676.2351717

[11] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated Concolic Testing of Smartphone Apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 59, 11 pages. https://doi.org/10.1145/2393596.2393666

[12] Haipeng Cai and Barbara Ryder. 2017. DroidFax: A Toolkit for Systematic Characterization of Android Applications. In *International Conference on Software Maintenance and Evolution (ICSME)*. 643–647.

[13] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 429–440. https://doi.org/10.1109/ASE.2015.89

[14] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: Programmable UI-automation for Large-scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '14)*. ACM, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390

[15] Winston Haynes. 2013. *Student's t-Test.* Springer New York, New York, NY, 2023–2025. https://doi.org/10.1007/978-1-4419-9863-7_1184

[16] 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 (ESEC/FSE 2013)*. ACM, New York, NY, USA, 224–234. https://doi.org/10.1145/2491411.2491450

[17] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: Segmented Evolutionary Testing of Android Apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 599–609. https://doi.org/10.1145/2635868.2635896

[18] 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.

[19] 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*. ACM, 34–37.

[20] 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 *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[21] C. Sun, Z. Zhang, B. Jiang, and W. K. Chan. 2016. Facilitating Monkey Test by Detecting Operable Regions in Rendered GUI of Mobile Game Apps. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 298–306. https://doi.org/10.1109/QRS.2016.41

[22] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2014. Execution and Property Specifications for JPF-android. *SIGSOFT Softw. Eng. Notes* 39, 1 (Feb. 2014), 1–5. https://doi.org/10.1145/2557833.2560576

[23] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-box Approach for Automated GUI-model Generation of Mobile Applications. In *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*. Springer-Verlag, Berlin, Heidelberg, 250–265. https://doi.org/10.1007/978-3-642-37057-1_19