On the Effectiveness of Random Testing for Android

or How I Learned to Stop Worrying and Love the Monkey

Priyam Patel Gokul Srinivasan Sydur Rahaman Iulian Neamtiu
Department of Computer Science
New Jersey Institute of Technology
Newark, NJ, USA
{pp577,gs387,sr939,ineamtiu}@njit.edu

ABSTRACT

Random testing of Android apps is attractive due to ease-of-use and scalability, but its effectiveness could be questioned. Prior studies have shown that Monkey – a simple approach and tool for random testing of Android apps – is surprisingly effective, "beating" much more sophisticated tools by achieving higher coverage. We study how Monkey's parameters affect code coverage (at class, method, block, and line levels) and set out to answer several research questions centered around improving the effectiveness of Monkey-based random testing in Android, and how it compares with manual exploration. First, we show that random stress testing via Monkey is extremely efficient (85 seconds on average) and effective at crashing apps, including 15 widely-used apps that have millions (or even billions) of installs. Second, we vary Monkey's event distribution to change app behavior and measured the resulting coverage. We found that, except for isolated cases, altering Monkey's default event distribution is unlikely to lead to higher coverage. Third, we manually explore 62 apps and compare the resulting coverages; we found that coverage achieved via manual exploration is just 2–3% higher than that achieved via Monkey exploration. Finally, our analysis shows that coarse-grained coverage is highly indicative of fine-grained coverage, hence coarse-grained coverage (which imposes low collection overhead) hits a performance vs accuracy sweet spot.

CCS CONCEPTS

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

KEYWORDS

Mobile applications, Google Android, Random testing, Stress testing, Code coverage

ACM Reference Format:
Priyam Patel Gokul Srinivasan Sydur Rahaman Iulian Neamtiu. 2018. On the Effectiveness of Random Testing for Android: or How I Learned to Stop Worrying and Love the Monkey. In *AST'18: AST'18:IEEE/ACM 13th International Workshop on Automation of Software Test , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3194733.3194742

AST'18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5743-2/18/05…$15.00
https://doi.org/10.1145/3194733.3194742

1 INTRODUCTION

Android runs on 85% of smartphones globally [7] and has more than 2 billion monthly active devices. Google Play, the main Android app store, offers more than 3 million apps [12]. At the same time, Android apps are updated on average every 60 days. Testing all these releases requires scalable, portable, and effective tools. One such tool is Monkey [4], a random testing tool that ships with Android Studio (Android's IDE). Monkey simply generates random events from a predefined distribution and sends the events to the app. Many other, sophisticated, automated testing tools for Android have been proposed – tools that require static analysis, dynamic analysis, or symbolic execution. Surprisingly though, a prior study that has compared the coverage attained by such tools on 64 apps has shown that, on average, Monkey manages to achieve the *highest coverage level* (48% statement coverage, on average) across all tools [5].

Given Monkey's unparalleled combination of ease-of-use, negligible overhead, and portability, we study whether Monkey is also *effective*. Specifically, we answer several research questions:

(1) Can Monkey crash popular apps via stress-testing?
(2) Can Monkey be more effective (yielding higher coverage) when appropriately "tuned"?
(3) Is coarse-grained (e.g., class/method) coverage indicative of fine-grained (e.g., block/line) coverage?
(4) Can manual exploration lead to higher coverage than Monkey's?
(5) Does inter-event time (throttling) affect coverage?

2 BACKGROUND

2.1 Android Platform

The Android system is based on the Linux kernel; on top of the kernel sits middleware written in C/C++ and Java. Android applications ("apps"), written in Java and (optional) C/C++ native code, run on top of the middleware. Android apps are high-level, event-driven, and with shallow method stacks. Below the application layer is the Android Framework which provides libraries containing APIs. Due to the variety of hardware devices and the constant evolution on the platform, Android developers should test their apps on multiple Android versions and multiple hardware devices, to ensure that other platforms are also supported – this however, raises portability issues, as the testing tool must run a variety of Android versions and devices (which makes Monkey an attractive choice).

Table 1: Monkey's Default Events and Percentages

Event ID	*Description*	*Frequency (%)*
TOUCH	single touch (screen press & release)	15
MOTION	"drag" (press, move, release)	10
TRACKBALL	sequence of small moves, followed by an optional single click	15
NAV	keyboard up/down/left/right	25
MAJORNAV	menu button, keyboard "center" button	15
SYSOPS	"system" keys, e.g., Home, Back, Call, End Call, Volume Up, Volume Down, Mute	2
APPSWITCH	switch to a different app	2
FLIP	flip open keyboard	1
ANYTHING	any event	13
PINCHZOOM	pinch or zoom gestures	2

Table 2: Bytecode size statistics for the app dataset.

	Bytecode size (KB)
Max	3,793
Min	420
Median	554
Mean	818

2.2 Monkey

UI/Application Exerciser Monkey (aka "Monkey") is an open-source random testing tool included in the Android SDK [4]. Monkey can run on either a physical device or an emulator. The tool emulates a user interacting with an app, by generating and injecting pseudo-random gestures, e.g., clicks, drags, or system events into the app's event input stream. In addition to generating and injecting events into an app, Monkey also watches for exceptional conditions, e.g., the app crashing or throwing an exception. Monkey sends events from a predefined distribution described in Table 1, i.e., 15% of the events it sends are TOUCH events (first row), 10% are drag events, 2% are system key events, 2% are pinch or zoom events, etc. In addition to event probability distribution, Monkey can also vary the "throttle", i.e., the time delays between events; by default there is no delay between events (throttle = 0).

3 EMPIRICAL STUDY

We now present the approach and results of our study. We phrase each experiment as a research question (RQ); at the beginning of each subsection we state the RQ and summarize the finding.

Experimental Setup. *Android VM.* We conducted experiments on the same virtual machine setup used by Choudhary et al. [5] for their comparison of automated Android testers. Specifically, we used multiple Android virtual machines (Oracle VirtualBox) running on a Linux server. Each VM was configured with 2 cores and 6 GB of RAM. Coverage was measured using Emma [14] – an open source toolkit used to measure and report Java code coverage.

App datasets. Our app dataset covered 79 real-world apps, drawn from a variety of categories, and having a variety of sizes. The 79 apps included 15 "famous apps", e.g., Shazam, Spotify, and Facebook for which no source code was available (and many of which use anti-debugging measures hence we did not measure coverage); this set was used for stress-testing only. The remaining 64 apps were popular apps, e.g., K-9 Mail, Yahtzee, or MunchLife, whose source code is available on GitHub. Table 2 shows statistics of app bytecode (i.e., `.dex`) size across the 64 apps.

Monkey runs. Since Monkey's exploration strategy is random, we use the same seed value for analyses requiring multiple runs to ensure the same input sequences are generated. After Monkey completes its run on each app, the emulator is destroyed and a new one is created to avoid any side effects between runs.

3.1 Application Crashes

RQ1: Is stress-testing with Monkey an effective strategy for crashing apps?
Answer: Yes

Stress testing is used to study a program's availability and error handling on large inputs or heavy loads – a robust, well-designed program should offer adequate performance and continuous availability even in extreme conditions. In Android, stress testing reveals the ability of an app to handle events properly in situations where, due to low resources or system load, long streaks of events arrive at a sustained rate. The app should handle such streams gently, perhaps with degraded performance, but certainly without crashing. Therefore, for our first set of experiments, we have subjected the test apps to stress-testing via long event sequences at *0 throttle*, i.e., no delay between events, as this is the *default* Monkey behavior.

Popular commercial apps. For the 15 famous apps, we present the results in Table 3. The first column shows the app name; the second column shows app popularity (number of installations, in millions); the third column depicts the time for crash in seconds. Note that four apps have in excess of 1 billion installs, while 12 apps have in excess of 100 million installs. We found that we were able to crash *all 15 apps* by letting Monkey inject 7,513 events on average; this means apps crash after just 85 seconds, on average.

Open source apps. In this case we ran stress testing setting throttle value at 0 msec and 100 msec, respectively. We found that, on average, apps crash after 8,287 events when throttle is set at 0 msec and 16,110 events when the throttle is set at 100 msec.

To conclude, it appears that *it is not a matter of if, but when* Monkey manages to crash an app. This demonstrates Monkey's effectiveness at stress testing apps, and therefore we can answer RQ1 in the affirmative.

3.2 Biasing Input Distribution

RQ2: Can Monkey be more effective (yield higher coverage) when appropriately "tuned"?
Answer: No

We biased the event distribution by increasing the probability for a single event type to 75% while proportionally shrinking the probabilities of the remaining 25% events according to the distribution shown in Table 1. Doing so, we boost each event kind's relative

Table 3: Stress Testing Results for Top Apps: The Number of Events at which the App Crashes

App	*Installs (millions)*	*#Events*	*Time to crash (seconds)*
Shazam	100–500	1,270	36
Spotify	100–500	9,011	83
Facebook	1,000–5,000	9,047	197
Whatsapp	1,000–5,000	14,768	175
Splitwise	1–5	819	12
UC Browser	100–500	5,575	51
NY Times	10–50	24,358	329
Instagram	1,000–5,000	279	7
Snapchat	500–1,000	4,676	24
Walmart	10–50	6,510	56
MX Player	100–500	1,742	41
Evernote	100–500	8,733	91
Skype	500–1,000	17,435	95
Waze	100–500	7,261	38
Google	1,000–5,000	1,220	34
Mean		*7,513*	*85*

importance in achieving coverage. We perform this analysis for all "important" categories in Table 1 having a default probability of 10% or higher: touch, motion, trackball, navigation, major navigation. We then measure the difference in coverage when using the biased distribution compared to the baseline Monkey distribution (i.e., default Monkey behavior). We show the resulting coverage mean values in Table 4.

We found a mild *decrease* in coverage for all types – class, method, block, and line. In certain isolated cases, we found small coverage increases, but these increases were not significant. Therefore, the hypothesis is rejected.

3.3 Coarse vs Fine-grained Coverage

RQ3: Is coarse-grained (e.g., class/method) coverage indicative of fine-grained (e.g., block/line) coverage?
Answer: Yes

We now turn to our analysis of correlation among coverage values. For the coverages obtained previously, we compute the pairwise correlation (Spearman's rank correlation coefficient). That is, for a certain experiment such as "Touch", for each app, we pairwise-correlate the class, method, block, and line coverage values. We show the results in Table 5. With two exceptions (Class v. Block and Class v. Line correlation for Motion events, where values are 0.78 and 0.82, respectively), the correlation values are *very high, greater than 0.9*. This finding allows us to make two observations about Android methods:

(1) Methods are shallow and are generally a good indicator of coverage without having to gather more detailed (but more expensive!) block or line coverage. For example, Android includes a method profiler [3] that could be used in lieu of specialized line coverage tools that require code instrumentation.

(2) A method's control flow graph appears to have a low number of paths: covering a method tends to cover all of its constituent blocks.

To conclude, this suggests a pragmatic approach for measuring coverage – low-overhead, coarse-grained coverage (at class or method level) is an accurate indicator of high-overhead, fine-grained coverage (block or line level).

3.4 Manual vs Monkey Coverage

RQ4: Within a 5-minute exploration time, does manual exploration achieve higher coverage than Monkey?
Answer: Yes

While manual exploration could be considered a "golden standard" at exploring an app thoroughly, we found that it might not to be the case for all apps. Specifically, we manually interacted with 62 apps[1] for 5 minutes each, trying to explore as many screens and functionalities as possible. We show the coverage mean values of all 62 apps in Table 6.

Although manual exploration achieves higher coverage, we found that Monkey obtained much higher coverage for 17–23 apps (depending on coverage type) e.g., A2DP Volume, Addi, aLogCat, Bites, Multi SMS, MunchLife, netcounter, PhotoStream and WeightChart – these apps fall in the categories of tools, health, lifestyle, media, and entertainment. Also Monkey has the same coverage as manual exploration for 12–22 apps (depending on coverage type).

K-9 Mail is an interesting case; the human "beat" the Monkey by a factor of 4.3x (39% vs 9% coverage), 9.6x (29% vs 3%), 13.5x (27% vs 2%), 6.6x (20% vs 3%) for class, method, block, and line coverage, respectively. This is due to a highly customized UI and the inability of Monkey to compose emails. The same holds for app Anymemostable where Monkey achieves lower coverage because it cannot compose notes.

Conversely, though, Monkey beat the human at Photostream by a factor of 3.1x (40% vs 13%), 3.1x (28% vs 9%), 29x (23% vs 8%), and 2.8x (25% vs 9%) respectively. To conclude, overall Monkey does exceedingly well: its coverage is just 2.3–3.9 percentage points lower than manual coverage, on average.

3.5 Throttling

RQ5: Does inter-event time (throttling) affect coverage?
Answer: No

Throttle is the delay between events sent by Monkey. We studied whether varying throttle affects coverage. We experimented with setting the throttle value to 0 (default), 100, 200, and 600 msec, while collecting line coverage only. We present the resulting mean coverages in Table 7. We found that, while 100 msec throttle leads to slightly higher coverage, the increase is not significant. Therefore, the hypothesis that throttle affects coverage is invalidated (at least for our chosen throttle values).

[1] 2 out of the 64 apps were not stable on our environment to permit prolonged testing.

Table 4: Results for Different Events

Coverage type	*Mean*					
	Default	*Touch 75%*	*Motion 75%*	*Trackball 75%*	*Navigation 75%*	*Major Navigation 75%*
Class	**50.7**	49.3	49.9	47.1	48.0	48.5
Method	**40.8**	39.7	39.8	36.5	38.1	39.2
Block	**36.9**	35.7	36.4	32.5	34.1	35.6
Line	**36.7**	35.5	35.1	32.0	33.7	35.0

Table 5: Correlation Between Coverage Types: Class (C), Method (M), Block (B), and Line (L)

	Touch			*App switch*			*Motion*			*Trackball*			*Navigation*			*Major Navigation*		
	M	B	L	M	B	L	M	B	L	M	B	L	M	B	L	M	B	L
C	0.93	0.91	0.92	0.94	0.93	0.94	0.95	0.78	0.82	0.97	0.94	0.96	0.98	0.96	0.96	0.97	0.93	0.95
M	-	0.96	0.97	-	0.97	0.97	-	0.93	0.94	-	0.97	0.98	-	0.99	0.99	-	0.95	0.97
B	-	-	0.99	-	-	0.99	-	-	0.99	-	-	0.99	-	-	0.99	-	-	0.99

Table 6: Monkey vs Manual Testing

Coverage type	*Mean*	
	Monkey	*Manual*
Class	52.4	**54.7**
Method	42.2	**45.0**
Block	38.6	**42.5**
Line	38.1	**41.6**

Table 7: Line Coverage when Varying Throttle

Throttle (msec)	*Mean Coverage (%)*
0	40.8
100	**45.0**
200	42.5
600	42.6

4 RELATED WORK

While there has been substantial work on test generation for Android, prior approaches have seen Monkey as a competitor, rather than a tool that itself needs to be explored to see how to increase coverage. These approaches fall into two categories. First, tools that rely on manual effort. Specifically, GUI-oriented tools, e.g., Android Guitar [2, 11], Robotium [6], and Troyd [10] rely on developers extracting a GUI model and then combining this model with input scripts to exercise the app. Second, fully automated tools. For example, Dynodroid [1] uses symbolic execution while A3E [13] uses static analysis, to explore apps automatically and generate input events. Choudhary et al. [5] have compared several input-generation tools for Android. Their conclusion was that Monkey offers the highest coverage (on average), hence our focus on Monkey and its parameters. TAST [8] (an upgraded version of MobileTest [9]) was built on the services provided by Monkey. It provides a script-based and black-box test infrastructure to develop and execute abstract test cases. However, its main focus is stress testing of TV applications, rather than smartphone apps.

5 CONCLUSIONS

We have conducted a study which has revealed that, despite its simplicity, random testing for Android is effective at revealing stress-related crashes; and in terms of coverage is on par with laborious approaches such as manual exploration. Moreover, our study has revealed that, except for isolated apps and event kinds, Monkey's default event type distribution and settings are appropriate for achieving high coverage in a wide range of apps. Finally, our study has reveled that coarse-grained, but low-overhead, class or method coverage is nevertheless effective (representative of finer-grained, block or line coverage).

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1617584.

REFERENCES

[1] A. Machiry, R. Tahiliani, and M. Naik. 2013. Dynodroid: An Input Generation System for Android Apps.. In *FSE '13*.
[2] Domenico Amalfitano, Anna Rita Fasolino, Salvatore De Carmine, Atif Memon, and Porfirio Tramontana. [n. d.]. Using GUI Ripping for Automated Testing of Android Applications. In *ASE'12*.
[3] Android Developers. 2017. Method Tracer. (September 2017). https://developer.android.com/studio/profile/am-methodtrace.html.
[4] Android Developers. 2017. UI/Application Exerciser Monkey. (September 2017). http://developer.android.com/tools/help/monkey.html.
[5] 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
[6] Google Code. 2012. Robotium. (August 2012). http://code.google.com/p/robotium/.
[7] IDC. 2017. Smartphone OS Market Share, 2017 Q1. (May 2017). https://www.idc.com/promo/smartphone-market-share/os.
[8] Bo Jiang, Peng Chen, Wing Kwong Chan, and Xinchao Zhang. 2016. To What Extent is Stress Testing of Android TV Applications Automated in Industrial Environments? *IEEE Transactions on Reliability* 65, 3 (9 2016), 1223–1239. https://doi.org/10.1109/TR.2015.2481601
[9] Bo Jiang, Xiang Long, and Xiaopeng Gao. 2007. MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices. *Second International Workshop on Automation of Software Test (AST '07)* (2007), 8–8.
[10] Jinseong Jeon and Jeffrey S. Foster. 2013. Troyd. (January 2013). https://github.com/plum-umd/troyd.
[11] BaoN. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2013. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* (2013), 1–41. http://dx.doi.org/10.1007/s10515-013-0128-9
[12] Statista. 2017. Number of available applications in the Google Play Store from December 2009 to June 2017. (July 2017). https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.
[13] T. Azim and I. Neamtiu. 2013. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In *OOPSLA '13*.
[14] Vlad Roubtsov. 2017. EMMA: a free Java code coverage tool. (July 2017). http://emma.sourceforge.net/.