The Influence of Code Coverage Metrics on Automated Testing Efficiency in Android

Stanislav Dashevskyi
SnT, University of Luxembourg
stanislav.dashevskyi@uni.lu

Olga Gadyatskaya
SnT, University of Luxembourg
olga.gadyatskaya@uni.lu

Aleksandr Pilgun
SnT, University of Luxembourg
aleksandr.pilgun@uni.lu

Yury Zhauniarovich
Qatar Computing Research Institute, HBKU
yzhauniarovich@hbku.edu.qa

Context

Automated testing and dynamic analysis techniques are critical for ensuring the reliability and the security of third-party Android apps.

One of the biggest challenges for these techniques is effective app exploration in the black-box setting. Android apps have many entry points, and their source code is unavailable for inspection. State-of-the-art tools utilize a wide variety of app exploration strategies that range from generating random GUI events to systematic exploration of apps models [1], but there is no agreement on the success criteria.

Code coverage is a common metric used to evaluate efficiency of automated testing and dynamic analysis tools [1], and some of these tools utilize **code coverage as a component of a fitness function** to guide app exploration and find more bugs [2].

Code coverage exists in many flavors, and there is currently no agreement in the community on which metrics to use in the fitness function. **Are they all the same, or is there a code coverage granularity that works best? We make the first step towards reaching this agreement.**

Hypothesis

Combining different granularities of code coverage can be beneficial for achieving better results in automated testing of Android apps.

Experiment setting

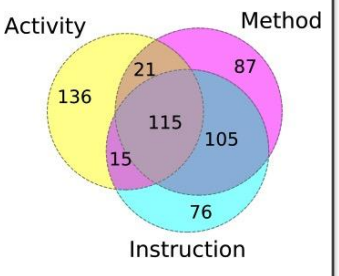
Sapienz [3] is a state-of-the-art bug finding tool for Android apps. It relies on Monkey [3] to generate random input events; and applies a genetic algorithm to event sequences. The test selection function combines code coverage, the number of found bugs, and the size of a test sequence. Sapienz is designed to utilize activity, method and statement coverage. We set out to evaluate how these metrics fare against each other in finding bugs.

Activity coverage was computed by Sapienz, and method and instruction coverage were measured with our own ACVTool (the tool is currently available at **https://github.com/pilgun/acvtool**).

Experiment 1: Comparing the metrics individually

We randomly selected 500 apps from Google Play and executed them with Sapienz using each coverage metric.

Coverage metric	# unique crashes	# faulty apps	# crash types
Activity	287	203	23
Method	317	231	23
Instruction	322	225	23
Total	555	295	26



Conclusion: Different metrics find different bugs.

Experiment 2: Evaluating the randomness impact

We randomly selected 100 apps, and ran Sapienz 5 times for each app using each coverage metric.

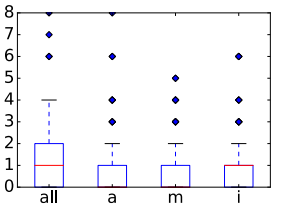
Coverage metrics	Crashes	
	$\mathcal{P}_1$: 1 run	$\mathcal{P}_5$: 5 runs
Activity coverage	54 (45%)	115 (58%)
Method coverage	72 (62%)	108 (55%)
Instruction coverage	65 (55%)	116 (59%)
Total	118	196

Conclusion: Even in multiple runs, no individual coverage metric was able to find all bugs detected by others.

Experiment 3: 1 run x 3 metrics vs 3 runs x 1 metric

For the 100 apps selected for the second experiment, we compute the number of faults detected jointly by 3 metrics and the number of faults found by each individual metric in 3 runs.

Statistics	1 run × 3 metrics	3 runs × 1 metric		
		activity	method	instruction
Min	0	0	0	0
Mean	1.18	0.95	0.85	0.95
Median	1	0	0	1
Max	8	8	5	6



We apply Wilcoxon test to evaluate the hypothesis that Sapienz with 3 metrics will on average find more faults than Sapienz with 1 metrics executed 3 times. The results of the test allowed to **reject the null-hypothesis** (that there is no difference) with p-values equal 0.008, 0.0005, and 0.007 for activity, method and instruction coverage, respectively.

Conclusion: The three metrics, when executed once, find on average more bugs in an app than each individual metric applied within 3 runs.

Conclusions

Our results show that different code coverage granularities find different bugs, and (given an equal execution time) that a combination of 3 different granularities will find more bugs than any of these metric granularities individually.

Open questions:

- How to reduce the total testing time?
- Will our findings hold for other tools?

References

[1] S. R. Choudhary, A. Gorla and A. Orso "Automated test input generation for Android: Are we there yet?" in ASE 2015
[2] K. Mao, M. Harman and Y. Jia "Sapienz: Multi-objective automated testing for Android applicatiions" in ISSTA 2016.
[3] Google, UI/App Exerciser Monkey, https://developer.android.com/studio/test/monkey

uni.lu UNIVERSITÉ DU LUXEMBOURG
Fonds National de la Recherche Luxembourg
QCRI Qatar Computing Research Institute جامعة حمد بن خليفة HAMAD BIN KHALIFA UNIVERSITY