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

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

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

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

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

ABSTRACT

Code coverage is an important metric that is used by automated Android testing and security analysis tools to guide the exploration of applications and to assess efficacy. Yet, there are many different variants of this metric and there is no agreement within the Android community on which are the best to work with. In this paper, we report on our preliminary study using the state-of-the-art automated test design tool Sapienz. Our results suggest a viable hypothesis that combining different granularities of code coverage metrics can be beneficial for achieving better results in automated testing of Android applications.

ACM Reference Format:

1 INTRODUCTION

Today Android enjoys immense popularity, with 3 millions of third party applications (apps for short) available in the official Google Play market as of 2018. While the end-users may be thrilled by this plethora of apps, they can start looking for alternative platforms if Google Play hosts too many faulty or even malicious apps. Not surprisingly, Google is interested in performing additional testing of submitted apps before they appear for the wide public [4]. However, the developers only supply compiled packages, while original source code of apps is unavailable for inspection. At the same time, given the amount of submitted apps, no market, even Google Play, possesses enough resources to perform exhaustive manual testing. Thus, frameworks for automated black-box app testing have become indispensable.

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CCS ’18, October 15–19, 2018, Toronto, ON, Canada
© 2018 Copyright held by the owner/author(s).
https://doi.org/10.1145/3243734.3278524

Automated testing and analysis tools for Android apps are typically designed to make the process more effective by finding more bugs or security issues; more comprehensive by achieving better code coverage; and faster by generating the smallest possible input sequences [2]. To achieve these goals, tools employ different approaches: from randomly generating sequences of input events to systematically exploring a model of the app under the test [2]. The algorithms used by these tools often rely upon a fitness function that promotes the selection of inputs that have the best performance with respect to a certain set of criteria.

For instance, a state-of-the-art search-based testing tool called Sapienz [7] recently acquired by Facebook uses a Pareto-optimal fitness function that depends on measuring code coverage. Sapienz employs three code coverage metrics that can be used interchangeably: activity coverage that is specific to Android apps, method, and statement coverage. However, there is no indication if either of these metrics is preferable. One may work with all of them, but this triples the time required for testing, already measured in hours per app. It is therefore desirable to understand whether there is an effect of different levels of granularity of code coverage metrics on the quality of automated testing, or if there is one coverage granularity that should be used.

In this paper, we make the first step in this direction. In particular, we hypothesize that there is an effect of considering different code coverage granularities on the effectiveness of automated test design tools, and report on experiments with Sapienz investigating the viability of this hypothesis.

2 BACKGROUND

Android apps are distributed in the form of Android package files (apk). These files are archives that, besides different resource files, contain bytecode executed by the Dalvik/ART Android virtual machine. Typically, automated black-box testing tools for Android work directly with these apk files. There exist many tools for functional testing and finding bugs [2, 6, 7] and for security-related testing and dynamic analysis, e.g., [1, 11]. The critical issue that all these tools have to address is efficient exploration of apps, i.e., triggering of GUI or system events that engage apps under the test [2]. Triggering is difficult for Android apps that have many different points of entry and are oriented towards interaction with the Android platform and the user. Lacking the knowledge
The importance of code coverage metrics for automated testing and dynamic analysis of Android apps is immediately evident from the aforementioned related work. Yet, we could not find in the literature any discussion on which specific code coverage metrics (or granularity levels) work best for Android. Therefore, the aim of our study is to fill this gap.

## 3 OUR STUDY

To investigate whether different levels of granularity of code coverage metric have an effect on the results of automated test design tools, we work with Sapienz [7]. It first generates a set of random “seed” test sequences, and then mutates them trying to improve a Pareto-optimal fitness function that depends on three criteria: code coverage, the length of a test sequence, and the number of app crashes that the test sequence has uncovered.

Sapienz can use three code coverage granularities. Statement coverage is measured by EMMA [9], a popular but outdated tool that works only for apps with source code available. Method coverage is measured by ELUA [3], another popular but no longer supported tool that often fails with more recent Android apps. In our experiments, we replaced EMMA and ELUA with ACVTool that measures bytecode instruction and method coverage [8]. Finally, activity coverage is measured by a plugin in Sapienz. Note that the code coverage measurement itself does not interfere with the search algorithms used by Sapienz.

As our dataset, we have randomly selected 500 apps from the Google Play market, and ran Sapienz against each of these apps, using its default parameters. Each app has been tested using the activity coverage provided by Sapienz, and the method and instruction coverage supplied by ACVTool [8]. On average, each app has been tested by Sapienz for 3 hours (for each coverage metric). After each run, we collected the crash information (if any), which included the components of apps that crashed and Java exception stack traces.

### 3.1 Descriptive statistics of crashes

Table 1 shows the numbers of crashes grouped by coverage metric that Sapienz has found in the 500 apps. We consider **unique crashes** as unique combinations of an application, its component where crash occurs and the line of code that triggered an exception, and a specific Java exception type.

In total, Sapienz has found 295 apps out of 500 to be faulty (at least one crash detected), and it has logged 555 unique crashes with all three coverage metrics. Figure 1 summarizes the crash distribution for the coverage metrics that found it. As we can see, the intersection of all code coverage metrics’ results contains 115 unique crashes (20% of total found crashes). Individual coverage metrics have found 58% (instruction coverage), 57% (method coverage), and 51% (activity coverage) of the total found crashes. These findings suggest that different code coverage metrics are complementary and could be applied together in order to achieve the best testing results.

### 3.2 Evaluating behavior on multiple runs

Like many other automated testing tools for Android, Sapienz is non-deterministic, and our findings may be affected by this. To determine the impact of coverage metrics in finding crashes on average, we need to investigate how crash detection behaves in multiple runs. Thus, we have performed the following two experiments on a randomly selected set of 100 apks.

**Performance in 5 runs.** We have run Sapienz for 5 times with each coverage metrics for each of 100 apps. This gives us two crash populations: P1 that contains crashes detected in the 100 apps during the first experiment, and P2 that contains crashes detected in the same apps running Sapienz 5 times. Table 2 summarizes the populations of crashes found by Sapienz with each of the coverage metrics.

As expected, running Sapienz multiple times increases the amount of found crashes. In this experiment, we are interested in the proportion of crashes contributed by coverage metrics individually. If coverage metrics are interchangeable (they do not differ in capabilities of finding crashes, and they will, eventually, find the same crashes), the proportion of crashes found by individual metrics to the total crashes population can be expected to significantly increase: each metric, given more attempts, will find a larger proportion of the total crash population. However, as shown in Table 2, only the activity coverage has found a significantly larger proportion of total crash population (59% from 45%). The instruction coverage has slightly increased performance (50% from 54%), while the method coverage has fared worse (55% from 62%). These findings suggest that the coverage metrics are not

### Table 1: Crashes found by Sapienz in 500 apps

<table>
<thead>
<tr>
<th>Coverage metric</th>
<th># unique crashes</th>
<th># faulty apps</th>
<th># crash types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity</td>
<td>287</td>
<td>201</td>
<td>23</td>
</tr>
<tr>
<td>Method</td>
<td>317</td>
<td>231</td>
<td>23</td>
</tr>
<tr>
<td>Instruction</td>
<td>322</td>
<td>225</td>
<td>23</td>
</tr>
<tr>
<td>Total</td>
<td>555</td>
<td>295</td>
<td>26</td>
</tr>
</tbody>
</table>
Crashes

<table>
<thead>
<tr>
<th>Coverage metrics</th>
<th>1 run</th>
<th>5 runs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity coverage</td>
<td>54 (45%)</td>
<td>115 (68%)</td>
</tr>
<tr>
<td>Method coverage</td>
<td>72 (62%)</td>
<td>108 (55%)</td>
</tr>
<tr>
<td>Instruction coverage</td>
<td>65 (56%)</td>
<td>116 (59%)</td>
</tr>
<tr>
<td>Total</td>
<td>118</td>
<td>196</td>
</tr>
</tbody>
</table>

Table 2: Crashes found in 100 apps with 1 and 5 runs

Table 3: Summary statistics for crashes found per apk

<table>
<thead>
<tr>
<th>Statistics</th>
<th>1 run × 3 metrics</th>
<th>3 runs × 1 metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Median</td>
<td>1.18</td>
<td>0.95</td>
</tr>
<tr>
<td>Max</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

3.3 Discussion

Our results show that all three code coverage granularities are complementary for finding bugs with automated testing tools such as Sapienz. However, running Sapienz with 3 metrics can be too time-consuming. Thus, it may be rational to prioritize coverage metrics that yield better results. At this point, it is not yet clear which coverage metric works best, and whether there are other confounding factors. Figure 1 might indicate that instruction and method coverage tend to identify similar crashes, while activity coverage finds somewhat different results. Second, we acknowledge that our tools measuring code coverage at the method and instruction levels may introduce some additional bugs during the instrumentation process. We have manually inspected several randomly selected crashes to confirm that they do appear in the original apk as well, and we have not found any discrepancies between the original and instrumented apk behaviours, but a larger experiment is required to eliminate this threat to validity. Moreover, other testing tools that rely upon code coverage might not obtain better results when combining several metrics at different levels of granularity. We plan to investigate this further by including more automated testing tools in experiments.

4 CONCLUSION

In this paper, we reported on the first step towards a better understanding of the usefulness of different code coverage granularities for automated test generation. Our experiment shows that the joint usage of several granularities of code coverage metric leads to discovering more bugs.

Acknowledgements

This research was partially supported by Luxembourg National Research Fund through grants C15/IS/10404933/COMMA and AFR-PhD-11289380-DroidMod.

REFERENCES