DEMO: An Effective Android Code Coverage Tool

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

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

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

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

Artsiom Kushniarou
SnT, University of Luxembourg
artsiom.kushniarou@uni.lu

ABSTRACT

The deluge of Android apps from third-party developers calls for sophisticated security testing and analysis techniques to inspect suspicious apps without accessing their source code. Code coverage is an important metric used in these techniques to evaluate their effectiveness, and even as a fitness function to help achieving better results in evolutionary and fuzzy approaches. Yet, so far there are no reliable tools for measuring fine-grained bytecode coverage of Android apps. In this work we present ACVTool that instruments Android apps and measures the `smali` code coverage at the level of classes, methods, and instructions.

Tool repository: https://github.com/pilgun/acvtool

ACM Reference Format:
Aleksandr Pilgun, Olga Gadyatskaya, Stanislav Dashevskyi, Yury Zhauniarovich, and Artsiom Kushniarou. 2018. DEMO: An Effective Android Code Coverage Tool. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), October 15–19, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3243734.3278484

1 INTRODUCTION

Android is the dominant mobile platform today, with millions of devices running it and millions of third-party applications (apps for short) available for its users. Unfortunately, this huge ecosystem suffers from proliferation of malicious [14] and buggy [10, 11] apps. Not surprisingly, techniques for automatic detection of malicious and faulty Android apps are in high demand.

One of the critical aspects in Android app analysis and testing is that apps are submitted to markets, including the main market Google Play, being already compiled and packaged. Their source code is not available for inspection neither to security researchers, nor to Google. Thus, automated analysis and testing tools need to operate in the *black-box* manner, without any knowledge of the expected behaviors of apps and with no access to their source code.

In this work, we specifically focus on measuring code coverage of Android apps. This metric is an integral part of software development and quality assurance activities for all programming languages and software ecosystems, and it has become a critical

metric for Android application analysis. Fellow researchers and practitioners evaluate the effectiveness of tools for automated testing and security analysis using code coverage (e.g., [3–5, 8, 10, 16]).

However, obtaining this metric is not a trivial task. Without the source code, code coverage is usually measured by instrumenting the bytecode, and this process is not straightforward for Android [8].

Related Work. Today, several tools for measuring code coverage over the bytecode of Android apps already exist, but they all have limitations. One of these limitations is the **coarse granularity** of the metric. For example, ELLA [6] and InsDal [9] measure code coverage only at at the method level.

Another limitation of the existing tools is **low instrumentation success rate**. For example, the tool by Huang et al. [8] measures code coverage achieved by popular dynamic analysis tools at the class, method, basic block and line granularities. However, the authors reported that they have been able to successfully instrument only 36% of apps to measure code coverage. Another tool for black-box code coverage measurement is BBoxTester [17] that has achieved the successful app instrumentation rate of 65%. It reports coverage at the class, method and basic block granularities.

Furthermore, the existing tools suffer from **limited empirical evaluation**, with a typical evaluation dataset of less than 100 apps. Often, research papers do not even mention the percentage of failed instrumentation attempts.

Remarkably, in the absence of a reliable fine-grained code coverage measurement tool, some frameworks integrate their own libraries that perform this task, e.g. [2, 10, 13]. However, as code coverage measurement is not the core contribution of these works, the authors have provided no information regarding their instrumentation success rates and performance.

Contribution. In this paper we present our ACVTool that measures code coverage of Android apps without relying upon their source code. ACVTool produces detailed coverage reports that are convenient for either visual inspections, or automatic processing. Our tool also collects crash reports that facilitate the analysis of faults within apps. We have empirically validated ACVTool against a large dataset of third-party apps. ACVTool has successfully instrumented 96.9% of apps in our experiments. Average time required to instrument an app with ACVTool is 36 seconds, i.e., it is negligible for the standard testing and analysis purposes. ACVTool is self-contained and transparent to the testing environment, and can be integrated with any testing or analysis tool. We have released ACVTool as an open source project to support the Android research community.
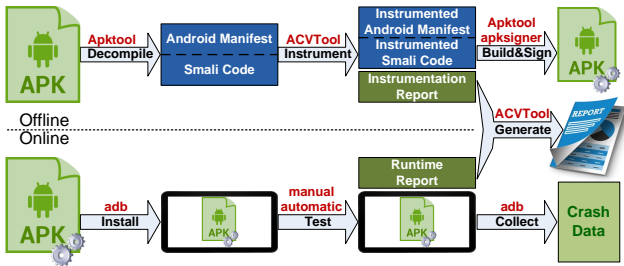
Figure 1: The ACVTool workflow

2 ACVTOOL ARCHITECTURE

ACVTool allows to measure and analyze the degree to which the code of a closed-source Android app is executed during testing, and to collect crash reports occurred during this process. The tool instruments an app and measures code coverage at instruction, method and class granularities. We designed ACVTool to be self-contained in order to make its integration with various testing solutions as easy as possible. In addition, the tool is helpful in manual analysis for investigating which code parts of a third-party application have been executed.

Figure 1 shows the main phases of ACVTool workflow: *offline*, *online*, and *report generation* phases, as indicated in upper, lower and right-most parts of the scheme, correspondingly. The offline phase handles the preprocessing of an app prior to its installation and instruments it. During the online phase, ACVTool installs the instrumented app and enables the data collection during testing. Later, in the report generation phase, ACVTool produces the information about code coverage. Below we describe the workflow of ACVTool in more details.

2.1 Offline phase

Instrumentation of a third-party app is not a trivial task due to the absence of the source code. This task requires to inject specific instrumentation bytecode instructions (*probes*) into the original bytecode. We based our tool on Apktool [15] that utilizes the *backsmali* disassembler [12] under the hood. We use this tool to disassemble an app into smali code – a human-readable representation of Android bytecode. Then we insert probes that log the execution of the original smali instructions. Afterwards, ACVTool builds a new version of the app and signs it with *apksigner*. Thus, ACVTool can instrument almost all applications that Apktool can repackage. Moreover, we do not worry about new versions of DEX files that might introduce new types of bytecode instructions: updating ACVTool to handle these new instructions will require little effort.

Smali code modification. Due to the stack-based Android architecture, modifying the smali code is not straightforward. We use the *direct instrumentation* approach previously introduced by Huang et al. [8] and Liu et al. [9], in conjunction with a *register management* technique. However, our approach is more efficient.

In our solution, we have optimized the mechanism of registering the execution of probes by maintaining a binary array. A probe writes a value into the corresponding cell of the binary array when executed. Writing binary values into specific array cells is a very

fast operation in comparison to manipulating string identifiers as in previous solutions [8, 9]. Moreover, this basic operation is much faster than creating new class objects or calling the Android API. In the end, the relatively large amount of the added probe code does not lead to any visible degradation of the application.

In its core, ACVTool puts a tracking probe right after each original instruction and label, excluding some corner cases. For instance, due to Dalvik-related limitations, the probes could not be inserted in-between some instruction pairs [7], such as the `invoke-*` or `filled-new-array` instructions followed by `move-result*`, and the `catch` label followed by the `move-exception` instruction. Moreover, the Android Runtime has the `VerifyChecker` component, which ensures that the exception-free part of the Java `synchronized` implementation (usually generated by the Android compiler) cannot raise an exception. The verifier scans the corresponding parts of the bytecode for unsafe instructions. To avoid failings at runtime, we wrap our tracking code by a `goto/32` call and return the execution flow straight back after the probe was registered.

Along with the instrumented apk file, the *offline* phase produces an *instrumentation report*, which matches the cells of the binary array onto smali code. It is a serialized code representation saved into a binary file with the `pickle` extension. This report will be applied in the *report generation* phase.

Orchestration. ACVTool injects a `Reporter` class and a special `Instrumentation` class in an apk. The first allocates memory to log probe execution and also enables pulling of code coverage information from memory into the external memory of the device. The second class provides the capability to monitor and save application-level crashes. The `Instrumentation` class also contains a broadcast receiver. Through this receiver ACVTool can trigger the `Instrumentation` class to initiate saving the code coverage information. To enable this functionality, ACVTool adds the `WRITE_EXTERNAL_STORAGE` permission and an `instrument` tag pointing to the `Instrumentation` class into the app manifest file.

2.2 Online phase

During this phase, we install the instrumented app on a device or emulator. We first activate the broadcast receiver implemented in the `Instrumentation` class. Then we can exercise the app manually or automatically, while logging the code coverage data.

After the testing is over, ACVTool generates another specific broadcast to consolidate the runtime information into a *runtime report* stored within the external storage of the device.

2.3 Report Generation phase

During this phase, ACVTool pulls the *runtime report* from the device and applies the *instrumentation report* generated during the *offline* phase. ACVTool generates code coverage report in the `html` and `xml` formats. The `html` report demonstrates the smali representation of the app code with appropriate coverage information in an easy to navigate browser view. Figure 2 shows an example of an `html` report, where individual smali code files available with the executed code are highlighted. The report gives the following information by columns: name of a smali file or a package, visualized numbers of missed vs. covered instructions, the code coverage value. The last six columns indicate the amount of the code that was not

Figure 2: The ACVTool code coverage report

Table 1: ACVTool performance evaluation

Parameter	F-Droid benchmark	Google Play benchmark	Total
Total # selected apps	448	398	846
Average apk size	3.1MB	11.1MB	6.8MB
Instrumented apps	444 (99.1%)	382 (95.9%)	97.6%
Healthy instrumented apps	440 (98.2%)	380 (95.4%)	96.9%
Avg. instrumentation time (total per app)	24.7 sec	49.6 sec	36.2 sec

executed during testing and the information about total amount of lines, methods and classes correspondingly. We generate also an `xml` version of the report containing the same code coverage information suitable for integrating in automated testing tools.

3 EVALUATION

We have extensively tested ACVTool on real-life third party applications. For the lack of space, we only report very basic evaluation statistics, summarized in Table 1.

Instrumentation success rate. For evaluation we have collected all application projects from the popular open-source F-Droid market, which is frequently used in evaluation of automated testing tools (e.g., in [10]). Among all projects on F-Droid, we were able to successfully build and launch on a device 448 apps. We have also randomly selected 500 apps from the AndroZoo [1] snapshot of the Google Play market, targeting apps released after the Android API 22. This sample is therefore representative of real-life third-party apps that may use anti-debugging techniques. Among the 500 selected apps, only 398 were launch-able on a device (the rest crashed immediately or gave installation errors). Thus, in total we tested ACVTool on 846 apps, with an average apk size of 6.8MB.

As shown in Table 1, ACVTool successfully instrumented 97.6% of these apps. Among the failures, 13 apps could not be repackaged by the Apktool, and the others have produced some exceptions during the instrumentation process. We then installed and launched all successfully instrumented apps, and found that 6 apps crashed immediately due to various errors. We thus can evaluate the total instrumentation success rate of ACVTool to be 96.9% on our dataset.

Overhead. As reported in Table 1, ACVTool introduces relatively small instrumentation-time overhead (36.2 seconds per app, on average) that is acceptable in the offline part of testing and analysis. We have also estimated the potential run-time overhead introduced by the added instrumentation code by running the original and repackaged app versions with the same Monkey scripts and comparing the execution timings. For 50 apps randomly selected from our dataset we have not found any significant difference in the execution time (median time difference less than 0.08 sec, mean difference less than 0.12 sec, standard deviation 0.84 sec), and we have not seen any unexpected crashes. This experiment suggests

that there is no drastic run-time overhead introduced by our instrumentation. We now work on an experiment to evaluate the run-time overhead precisely.

4 DEMO DETAILS

We plan an interactive demonstration of the whole coverage measurement cycle on real applications. In the demo, we will walk the audience through the ACVTool design. We will also show how the smali code looks like before and after injecting the probes.

5 CONCLUSION

We reported on a novel tool for black-box code coverage measurement of Android applications. We have significantly improved the smali instrumentation technique and consequently our instrumentation success rate is 96.9%, compared with 36% in Huang et al. [8] and 65% in Zhauniarovich et al. [17]. Furthermore, our implementation is open-source and available for the community.

REFERENCES
[1] K. Allix, T. Bissyandé, J. Klein, and Y. Le Traon. 2016. AndroZoo: Collecting Millions of Android Apps for the Research Community. In *Proc. of MSR*. ACM, 468–471.
[2] H. Cai and B. Ryder. 2017. DroidFax: A toolkit for systematic characterization of Android applications. In *Proc. of ICSME*. IEEE, 643–647.
[3] P. Carter, C. Mulliner, M. Lindorfer, W. Robertson, and E. Kirda. 2016. Curious-Droid: Automated user interface interaction for Android application analysis sandboxes. In *Proc. of FC*. Springer, 231–249.
[4] S. R. Choudhary, A. Gorla, and A. Orso. 2015. Automated test input generation for Android: Are we there yet?. In *Proc. of ASE*. IEEE/ACM, 429–440.
[5] S. Dashevskyi, O. Gadyatskaya, , A. Pilgun, and Y Zhauniarovich. 2018. POSTER: The Influence of Code Coverage Metrics on Automated Testing Efficiency in Android. In *Proc. of CCS*.
[6] ELLA. 2016. A Tool for Binary Instrumentation of Android Apps, https://github.com/saswatanand/ella.
[7] Google. 2018. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.
[8] C. Huang, C. Chiu, C. Lin, and H. Tzeng. 2015. Code Coverage Measurement for Android Dynamic Analysis Tools. In *Proc. of Mobile Services (MS)*. IEEE, 209–216.
[9] J. Liu, T. Wu, X. Deng, J. Yan, and J. Zhang. 2017. InsDal: A safe and extensible instrumentation tool on Dalvik byte-code for Android applications. In *Proc. of SANER*. IEEE, 502–506.
[10] K. Mao, M. Harman, and Y. Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proc. of ISSTA*. ACM, 94–105.
[11] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically discovering, reporting and reproducing Android application crashes. In *Proc. of ICST*.
[12] smali/backsmali. 2018. https://github.com/JesusFreke/smali.
[13] W. Song, X. Qian, and J. Huang. 2017. EHBDroid: Beyond GUI testing for Android applications. In *Proc. of ASE*. IEEE/ACM, 27–37.
[14] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou. 2017. Deep ground truth analysis of current Android malware. In *Proc. of DIMVA*.
[15] R. Wiśniewski and C. Tumbleson. 2017. Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/
[16] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X S. Wang. 2013. Appintent: Analyzing sensitive data transmission in Android for privacy leakage detection. In *Proc. of CCS*.
[17] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci. 2015. Towards black box testing of Android apps. In *Proc. of SAW at ARES*. IEEE, 501–510.