GreenSource: Repository tailored for Green Software Analysis

Rui Rua
Departamento de Informática
Universidade of Minho
R. da Universidade, 4710-057 Braga
rui.a.rua@inesctec.pt

Abstract— **Energy consumption analysis and energy-aware development have won the attention of both developers and researchers over the past years. The interest is becoming more notorious due to the proliferation of mobile devices, where saving energy is a key concern.**

In the last years, a considerable number of studies aiming at analyzing the energy consumption emerged, with objectives such as measuring/estimating the energy consumed by an application or code block, or even detecting energy-expensive coding patterns. However, when it comes to actually improving the energy efficiency of an application, the amount of information provided about source code energy consumption that can be used by developers to reduce it in the development phase, is still very low.

In this paper we present GreenSource, a publicly available repository containing more than 600 Android Projects extracted from open-source repositories. This infrastructure contains static and dynamic metrics obtained from the execution of stress and unit tests over the projects' applications. The results were obtained using a tool developed in this work context, the AnaDroid framework. This tool uses testing frameworks and an energy profiler to instrument, build and execute tests over applications in a physical device, while monitoring its energy and resources consumption/usage.

Processing each one of this projects is a time-consuming task, due to the lack of tools capable of gather all this information and the large size and complexity of the projects. With this work, we intend to openly provide the resultant metrics and metadata obtained from the process of analyzing the projects and its execution. The queryable and minable data provided by the GreenSource can be used for further studies and researches, helping developers community to reason about energy consumption in software and relate it to source code.

I. INTRODUCTION

With the advancement of the technological age, the software engineering community has been focusing on how software is developed and continually progressed in this direction. Efforts in this regard have been made at various levels, from hardware [1] level to compilers [2], programming languages [3] or integrated development environments (IDE's). These intended to increase the productivity of software development, abstracting inherent development processes, allowing developers to focus on the most essential functional aspects of their software product.

However, since the beginning of the century we have witnessed a revolution in the computer systems portability. The portability factor became much valued by users, and thereafter also for its manufacturers. Consequently, given the limited capacity of the battery of such devices, the optimization of energy consumption for these has proved to be a crucial aspect for producers of these, as well as for the developers of software for these platforms.

Accompanying the mobile market growing, the Android ecosystem keeps evolving at an impressive pace as well. Since this operative system can run on a wide variety of devices, from smartphones, tablets or weareables, its widespread usage in the last decade was significantly notorious. This is the most used operative system for mobile devices, having in 2018 around 84,8% devices running its platform[4].

Over the last few years, the interest in analyzing energy consumption of the Android platform and respective applications has been increasing significantly. Energy-greedy mobile apps that drain the battery of devices are perceived as being of poor quality by users [5]. As a consequence, users are likely to uninstall an energy-inefficient app, and sometimes are even recommend to do so.

Due to this recent interest, in the last decade several works in this sense appeared. These aimed at analyzing the energy consumption in multiple ways, such as measuring/estimating the energy consumed by an application or block of code [6], [7], or even detecting energy expensive coding patterns [8] or API's [9]. In order to perform optimization in terms of energy at software level, we face a whole new challenge, which can only be achieved through source code improvements that can take advantage of energy saving techniques. Nonetheless, in order to identify energy-greedy code and propose techniques/solutions to avoid it, a significant and characterizing amount of information regarding the code energy consumption has to be analyzed.

The versatility and continuous evolution of the Android platform, with a constantly changing architectural and functional environment, leads to the increasing challenge of gather characterizing information about its energy and resources consumption. Since this platform runs in a countless number of devices [10], with different hardware components and architectures, running different versions of the system, its almost unfeasible to find solutions with satisfactory results for all configurations.

In this paper, we present an approach to gather useful metrics and data about the execution of portions of applications' source code in physical devices. We reused the GreenDroid [11] concept to instrument and monitor the execution of source code portions, providing an extensible

framework that can be used by developers to estimate energy consumption of application. We executed this framework over more than 600 Android applications, extracted from the MUSE repository[1]. The results of the execution of the extracted applications was then centralized in an open repository. This infrastructure was designed to store metrics and metadata relatively to executed code of Android applications. The obtained information is related to the characteristic of the executed application and respective platform and device.

To summarize, the developed work involves essentially two main artifacts:

- The **Anadroid** framework: Tool that resulted from the evolution of GreenDroid, which consisted in one of the starting points to carry out the energy consumption analysis of source code in Android. This framework was conceived in order to have ability to instrument the source code of any Android project, generate the respective APK (Android PacKage) and monitor its execution. The execution of the applications is done through stress or unit tests.
- **GreenSource** infraestructure: To demonstrate the power of the AnaDroid framework, we executed it over hundreds of Android projects. Having access to a large number of applications and a powerful tool like the AnaDroid , it was decided to build an infrastructure capable of store and organize information of all executions. As such, the GreenSource was built. It is a repository containing data and metrics related to the structure and performance of applications that can be related to the energy consumption of its source code.

The information that the resultant infrastructure contains is openly available for consultation (`http://greenlab.di.uminho.pt/greensource/`). The main goal of this work is to offer a relevant scientific contribution that can be significant and characterizing the Android platform, being able to be reused in later studies. In order to achieve these objectives, we intend to continue to populate this repository with information regarding more applications, tests and devices. The contained data can be subjected to analyzes and studies (e.g. Data Science/Machine-Learning) that can allow to correlate factors that can have a significant impact on energy consumption and obtain relevant conclusions about of the applications code.

The remaining of this paper is organized as follows. In Section 2 we introduce the main artifacts and methodologies followed to obtain the GreenSource infrastructure and the information contained in it. We then present the results of the experiment in Section 3. Section 4 presents the threats to the validity of this work. Finally, in Section 5 we present our conclusions and future work directions.

[1]Muse repository: `https://opencatalog.darpa.mil/MUSE.html`

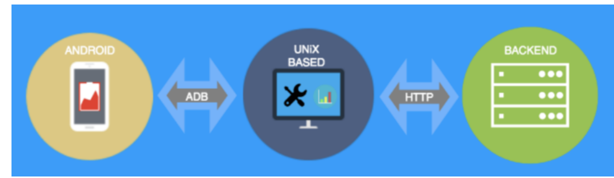

Fig. 1: GreenSource high-level components

II. GreenSource

A. Data provenance

In order to obtain diverse and relevant material for the accomplishment of studies that could reach significant magnitude, we needed to gather a significant number of Android Projects. The goal was to collect projects from a wide variety of sources in order to obtain a diverse set and whose projects/source code was openly accessible. Excepting the alternative of developing a tool that analyzes open-source repositories, which identifies Android projects and extracts that content, we reused previous works that had the same goal.

In order so obtain such corpus of Android projects, We took advantage of the work done during the development of GreenDroid[11], whose goal was to extract Android projects from the MUSE repository, an extension of the sourcerer repository[12]. These projects were collected from other open repositories and were developed in Java, the current leading development language for the Android platform.

Among the thousands of projects contained in the repository, we selected those that we could identify as Android projects, by executing queries on the repository database. Of all identified projects, we selected a subset that we identified as functional (i.e. compiled, built and executed without errors). Excluding all the problematic apps, we obtained a set containing more than 600 functional projects, which allow us to build applications that can be installed and run on Android devices. This set represents the starting point for the creation of the repository, already having a considerable size and minimally representative, and can be increased in future updates.

B. AnaDroid Framework

The AnaDroid tool was developed to offer a generic way of integrating the ability to measure the energy consumption of an Android application. This tool can be used during its development process, as well as to automate the procedure of executing it over a large set of applications. This framework comes as an evolution of the GreenDroid framework [11], making it more accurate, current and complete. Its workflow is quite similar, from the instrumentation phase to the test execution phase. Several changes were made to how GreenDroid performed the instrumentation, exercised and analyzed the code and energy consumption of applications. Its concept has been extended to be able to interact with more testing frameworks, as well as new energy profilers, such as Trepn Profiler. With the inclusion of these new tools and with changes made to its workflow and how it analyzed the

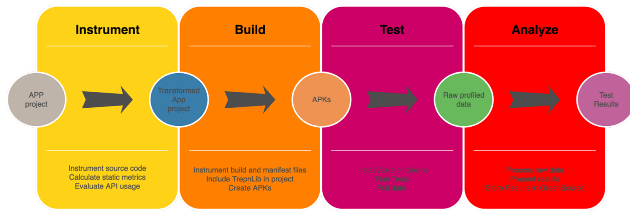

Fig. 2: AnaDroid Workflow

application code statically, it was possible to extract more information that can be associated and justify the energy performance of the applications.

The AnaDroid workflow is showed in the figure 2. It starts by instrumenting an Android project, both at the source code and building scripts level. This step is needed, in order to delimit the code execution interval and make calls to the energy profiler. In addition, during the instrumentation, it also collects static metrics and metadata about application methods and classes (Some of them are present in table **??**. Te next steps consists in generate the APK and install it on a physical device, using the ADB tool (Android Debug Bridge), allowing to perform and manage these tasks from the development machine. According to the intended testing framework, the tests are then executed on the device, the resultant data is collected and the results for each of the tests are generated. Before and after each test run, information about the resources (CPU, free memory and number of running processes) and status of the device which may interfere with the tests results are collected . In addition, in the final stage (Analyze phase), the AnaDroid can analyze and process the results obtained for each test executed. Then, it can send the obtained metrics and data to the GreenSource backend, in order to centralize results and contribute to the growth of knowledge regarding the power consumption and features of Android code.

C. *TrepnLib and Trepn*

Trepn Profiler[2] is a software-based artifact developed by Qualcomm that works on devices with Snapdragon chipset-based Android devices. It is a diagnostic tool designed for expert consumers, such as Android developers. It can be used to profile hardware usage (like GPS, WiFi and others), resources usage (memory, CPU) and power consumption of the system or standalone Android applications. This tool doesn't need external (hardware) tools, as it gets its power readings from the Power Management IC (PMIC) and the battery fuel gauge software.

Trepn can be used as an standalone application, or as a service (an unix-like daemon in Android), which allows invocations via source code or from the ADB tool. This versatility makes this profiler easy to integrate in Android-based tools and applications, in purpose of measure and profile portions or entire applications. It provides the capability of pin data points (application states) while monitoring, which can be used to log and mark specific events during the profiling timeline.

Given the capability of Trepn of being invoked via (Java) source code, we had to find a way to easily integrate his calls in Android Applications, abstracting the calls to the Trepn Service. We developed a Android Library (TrepnLib) for this purpose, providing an API that allows to isolate and profile portions/code blocks (like methods,loops or Activity's lifecycle) of any Java class present in the application source code. Instrumenting the source code with the API provided by the TrepnLib, it is possible to estimate the power consumption and profile the isolated block, as well log other relevant events, like the start/end of methods, identify recursive calls, and others. To provide all this capabilities, we designed the TrepnLib taking into account his use cases. We reached the conclusion that the most common blocks/portions of code that are more relevant to isolate in terms of debugging and development process were methods and (unit) test cases. The capability of estimate power consumption of Java code at instruction/line level is not reliable using Trepn Profiler, since his sample rate is never lower than 100 ms, difficulting the task of associate samples at an specific rate with executions of instructions that take only a few milliseconds to run.

Furthermore, we provided functions to start and stop the profiling process, given the type of instrumentation (method or test oriented), that start and stop the Trepn Service, as well creates auxiliary files that are used to manage several runs,states and contexts. Methods to trace usage of methods and log states/events are also provided.

D. *GreenSource Backend*

In order to give a greater purpose to the Anadroid framework, it has been integrated into the GreenSource repository. The main function of GreenSource's backend is to store and manage the information gathered through AnaDroid tool executions over Android projects. As such, the GreenSource contains a database within, having the function of providing an uniform way of communicating with it. In this way, mechanisms can be put in place that eases the processes of management, validation and manipulation of data at a higher architectural level, obtaining an abstraction level independent of the database engine used. The communication interface chosen consists in a RESTful API, which enables a uniform form of communication that provides the ability to consult, insert, change and delete data through HTTP requests. The database has been carefully designed to be expandable for future refinements and expansions of the AnaDroid tool. This database is a relational database and its schema consists of 21 tables, which refer to the elements that compose the application, as well as metadata and metrics related to the execution and analysis of the ones made on them.

The way the database was structured and developed, allows it to accompany the expansion of Anadroid, being easily extensible to support different test frameworks, devices and energy profilers.

[2] `https://play.google.com/store/apps/details?id=com.quicinc.trepn`

III. RESULTS

This section demonstrates some results obtained with the help of the AnaDroid tool, which were stored in the database of the GreenSource backend. Several types of results were selected for the execution of application tests with the UI/Application Exerciser Monkey test framework. These results allow to compare tests, applications and portions of these, as well its executions.

The process of running AnaDroid on a wide range of applications is an extremely costly process over time. This time is influenced by both the performance of the development machine and the Android device on which the tests are performed. Until the writing of this paper, this framework was successfully executed over a total of 352 Android projects. The features and specifications of the device in which the applications and tests were executed are described in table I.

Feature	Details
Chipset	Snapdragon 400 Qualcomm MSM8226
CPU	1.2 GHz Quad Core
GPU	Adreno 305
RAM	1 GB
Mem	8 GB
Screen	IPS LCD 720 x 1280 pixel 16M colors
Wifi	802.11b/g/n
Bluetooth	4.0 com A2DP/LE
GPS	A-GPS/GLONASS
Battery	2070 mAh

TABLE I: Android device specifications

We tried to run tests until we reached a relevant method coverage, approximately equal to or greater than 60%. These were done using the framework UI Application Exerciser Monkey, since its tests reach much higher values of method coverage than those obtained with the JUnit tests present in some projects. In order to reach this level of method coverage, 20 equal tests (generated from the same seeds) were carried out for each one of these applications. If this level of method coverage was not reached after 20 tests, the process would continue for more 30 tests. These tests were executed using the same seeds, in order to generate the same sequence of events for every application. The workflow of the test execution is represented in figure 3

In order to prevent the pseudo-random events generated by the Exerciser Monkey from turning on/off system resources or invoking other applications, some precautions had to be taken. The first consisted of using an auxiliary application called Simiasque[3], which hides status bar under an overlay mask, preventing monkey tests from clicking it. The second was to prevent Exerciser Monkey from generating system-events (pressing the Home, Back, Start Call, End Call, or Volume buttons) i to prevent generated events from being made outside the running application interface or prevent the phone from rebooting.

We then repeated this process for the 352 applications and analyzed the obtained results. By obtaining this type of

[3]https://github.com/Orange-OpenSource/simiasque

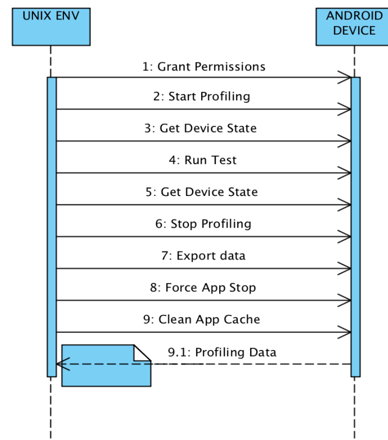

Fig. 3: Test Execution workflow

results for a large set of applications, it is possible to compare applications for each one of the obtained metrics, in order to be able to correlate them with the (energy) performance of these. To illustrate examples of the comparisons and conclusions that can be made (in the future, with much more confidence, when a more significant number of different applications were analyzed), we have selected the following applications:

- Android DisplayingBitmaps: due to being the application with more methods invoked during the execution of the tests.
- PkTest[4]: Application that has achieved considerable test runtime and uses above average amount of sensors/hardware usage.
- Material Library: It obtained a total execution time of tests quite similar to the PkTest application.

The nature of the results allow to visualize and compare applications according to the executed tests. For the Android DisplayingBitmaps, the table II shows some of the results obtained for each executed test.

Test Number	Consumption (J)	Time (ms)	Coverage (%)	Avg Mem Usage (MB)	Avg GPU Load (%)	Avg CPU Load (%)
89160419	74.919	33515	69.69	836987.535	0.88	61.789
11	48.04278	21551	40.40	843599.578	5.912	53.960
435986	90.456	24204	69.69	822303.591	7.407	56.781
40201	71.834	29017	69.69	825611.297	4.766	56.546
16	76.4522	27988	69.19	808640.368	4.689	54.929
231251	51.927	18337	69.69	820401.516	3.446	50.508
927139	58.230	26049	69.69	815680.168	5.056	59.879
123456789	60.152	25338	69.69	826982.464	5.305	55.934
256773292	59.510	23190	69.69	827095.791	5.625	54.002
330101	98.010	35165	69.69	827265.815	5.118	61.424
12131145	50.336	24695	69.69	833746.1977	1.9199	53.0411
1986	69.578	30640	63.63	811824.113	3.877	56.746
2018	49.380	22700	69.69	814691.094	2.554	52.995
1893	60.794	29015	62.12	847369.156	3.937	57.120
8913489	79.76	28309	69.69	830391.543	6.125	55.557
72929123	58.72	25635	69.69	821123.176	3.570	54.953
236236	68.39	25603	72.22	838037.96	4.640	56.984
37666	39.376	19059	69.69	827088.545	5.640	53.141
8894018411	57.80	24832	69.69	820214.612	3.927	53.182
5637	53.105	27954	69.69	820144.307	5.6738	56.569
Total coverage			72.22			

TABLE II: Some test results obtained for Android DisplayingBitmaps app

For instance, in figure 4 we can conclude that the PkTest

[4]https://github.com/zubietaroberto/AndroidKeyStoreTest

application has a lower (but similar) execution time performance than the Material Library application. However, it has a higher energy consumption, even invoking only 6280 methods during the execution of the test, well below the 311,236 invoked by the latter. The hardware usage values (CPU, GPU, Memory) are higher for PkTest, and for GPU, the average usage percentage value (5.44 %) of this feature is 777.24% higher than the registered for the Material Library application. In this we can conclude that the use of this type of hardware also can have a considerable impact on the energy performance of an application.

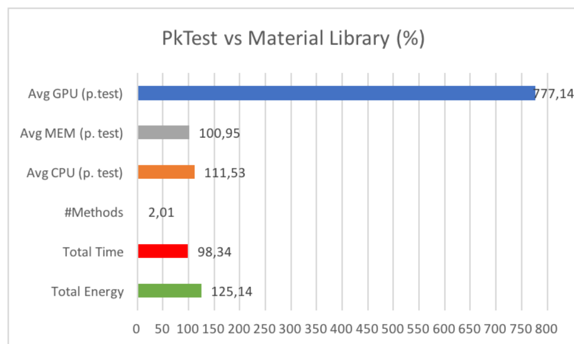

Fig. 4: Comparison between PkTest and Material Library

IV. Threats to validity

Measuring the energy consumption of a mobile device is complex [13]. This is mostly due to the fact that it is quite difficult to fully isolate the code or application under measurement.

Today's operating systems, such as Android, have the ability to run multiple processes and applications simultaneously. Due to the difficulty of ensuring that during the execution of each test, only the intended application is running on the device and having an impact on its consumption, we register the state of the device before and after each test Following this approach, information about the status of the device is collected, which may interfere with the performance of the tests, like the number of processes running, percentage of CPU used and memory available.

Moreover, we executed the application in a factory-reseted device, with the lowest brightness level, to ensure the energy consumed by the display was as low as possible. We didn't considered testing in a root device, since we wanted to emulate a more realistic environment of execution. However, in order to avoid interferences, we did not provide Google account credentials, in order to avoid minimize the computations of Google services, like checking for updates.

Finding an adequate tool for energy profiling for the Android environment was also a challenge. The Android platform still lacks tools that allow developers to quickly and reliably monitor power consumption, as well locate energy hotspots in their code. Trepn is an accurate tool[14], capable of profile hardware usage (like GPS, WiFi and others), resources usage (memory, CPU) and power consumption of the system or even standalone Android applications, gets its power readings from the power management Integrated Circuit (PMIC) and the battery fuel gauge software. The main limitation of this profiler is that only gets accurate battery power readings from chipsets developed by Qualcomm, and the sampling rate can't be adjusted to less than 100 ms. However, we consider that is still the best free software-based alternative for this purpose, since this company dominates the smartphone SoC (System on a Chip) market due to date [15]. Another issue that we needed to solve was finding an approach to properly compare metrics applications of different types and domains. In order to avoid labeling and compare applications according to its domain and functionalities, we decided to exercise the User Interface of every applications and compare them by the respective obtained consumption. The tests were executed with the Application Exerciser Monkey, that allows to simulate user interaction and I/O events. In order to reach a relevant amount of method coverage to fairly compare executions, 20 equal tests (generated from the same seeds) were carried out for each one of the applications. If the defined level of method coverage was not reached after 20 tests, the process would continue for 30 more tests. These tests were executed using the same seeds, in order to generate the same sequence of events for every application.

V. Conclusions

The main contributions of this work go from a tool capable of gathering relevant metrics and metadata to justify the consumption of code blocks of Android applications, to the development of an infrastructure capable of automating and gathering executions of this tool. We successfully implemented our methodology, resulting in a global infrastructure containing so far more than 600 Android applications and results from over 6,000 tests executed over some of these.

We were able to extend the GreenDroid framework to be capable of gather more information about the source code structure and become more expandable and precise. With the capability of easily integrate new testing frameworks and new energy profilers, like the Trepn Profiler, it became a tool that can be used for generically process Android Projects. It can be integrated in the testing phase of the development lifecycle of Android applications, helping developers to observe the energy and resources consumption, relating it to metrics obtained from dynamic and static analysis.

As a form of providing and share the results obtained, as well to prove and take advantage of the power of the AnaDroid, an open repository was developed. It contains hundreds of Android applications and respective results and metrics obtained with the execution of them (or portions) in a physical device. By agglomerating a high number of results, we pretend to obtain a set of information characterizing the Android development paradigm. This will allow to relate consumption with levels resource usage, to energetically compare different applications and devices and to obtain quality metrics of tests and software. In addition, it is hoped that the information retrieved from this repository may be (re)used in further works and researches.

REFERENCES

[1] J. W. Yoo and K. H. Park, “A cooperative clustering protocol for energy saving of mobile devices with wlan and bluetooth interfaces,” *IEEE Transactions on Mobile Computing*, vol. 10, no. 4, pp. 491–504, April 2011.

[2] M. Pedram, “Power minimization in ic design: Principles and applications,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 1, no. 1, pp. 3–56, Jan. 1996. [Online]. Available: http://doi.acm.org/10.1145/225871.225877

[3] *WOLFHPC ’14: Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. Piscataway, NJ, USA: IEEE Press, 2014.

[4] (2018) Smartphone market share. [Online]. Available: https://www.idc.com/promo/smartphone-market-share/os

[5] (2018) Top frustrations that lead to bad mobile app reviews. [Online]. Available: https://bit.ly/2QLoDTA

[6] J. C. J. P. F. Tiago Carção, Marco Couto and J. Saraiva, “Detecting anomalous energy consumption in android applications,” 2014.

[7] D. D. Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, “Petra: A software-based tool for estimating the energy profile of android applications,” in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, May 2017, pp. 3–6.

[8] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, “Mining energy-greedy api usage patterns in android apps: An empirical study,” in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: ACM, 2014, pp. 2–11. [Online]. Available: http://doi.acm.org/10.1145/2597073.2597085

[9] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, “Fine-grained power modeling for smartphones using system call tracing,” in *Proceedings of the Sixth Conference on Computer Systems*, ser. EuroSys ’11. New York, NY, USA: ACM, 2011, pp. 153–168. [Online]. Available: http://doi.acm.org/10.1145/1966445.1966460

[10] (2018) There are now more than 24,000 different android devices. [Online]. Available: https://bit.ly/2NNfPHQ

[11] M. Couto, J. Cunha, J. P. Fernandes, R. Pereira, and J. Saraiva, “Greendroid: A tool for analysing power consumption in the android ecosystem,” in *2015 IEEE 13th International Scientific Conference on Informatics*, Nov 2015, pp. 73–78.

[12] J. Ossher, S. Bajracharya, E. Linstead, P. Baldi, and C. Lopes, “Sourcererdb: An aggregated repository of statically analyzed and cross-linked open source java projects,” in *2009 6th IEEE International Working Conference on Mining Software Repositories*, May 2009, pp. 183–186.

[13] A. Banerjee and A. Roychoudhury, “Future of mobile software for smartphones and drones: Energy and performance,” in *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2017, pp. 1–12.

[14] A. R. Bakker, “Comparing energy profilers for android,” in *Proceedings of 21st Twente student conference on IT, Enschede, The Netherlands*, 2014.

[15] (2018) Global smartphone system-on-chip (soc) revenue share by vendor. [Online]. Available: https://bit.ly/2yt4jMe

APPENDIX

Class	Method	Times invoked	CC	LoC	AndroidAPIs	N args
FlagsActivity	onCreate	198	2	6	2	1
BaseFlagFragment	validate	1071	6	18	0	0
VerifyPhoneFragment	onCreateView	198	1	4	5	3
CustomPhoneNumberFormattingTextWatcher	hasSeparator	781	4	8	0	3
BaseFlagFragment	onPostExecute	135	0	1	0	0
FlagsActivity	onOptionsItemSelected	19	3	6	4	1
BaseFlagFragment	onPhoneChanged	715	0	1	0	0
BaseFlagFragment	initCodes	198	1	2	1	1
CustomPhoneNumberFormattingTextWatcher	reformat	715	6	24	0	2
CountryAdapter	getView	4224	2	8	7	3
VerifyPhoneFragment	onActivityCreated	198	1	3	1	1
Country	getCountryCode	87757	1	2	0	0
CustomPhoneNumberFormattingTextWatcher	stopFormatting	67	1	3	0	0
CustomPhoneNumberFormattingTextWatcher	onTextChanged	2155	4	7	0	4
BaseFlagFragment	initUI	198	1	38	16	1
BaseFlagFragment	hideKeyboard	1071	1	3	9	1
Country	getCountryCodeStr	44	1	2	0	0
FlagsActivity	onCreateOptionsMenu	198	1	3	1	1
BaseFlagFragment	onItemSelected	44	0	1	0	0
Country	getPriority	61	1	2	0	0
BaseFlagFragment	doInBackground	198	0	1	0	0
CustomPhoneNumberFormattingTextWatcher	afterTextChanged	2155	11	28	10	1
CustomPhoneNumberFormattingTextWatcher	beforeTextChanged	2155	4	7	0	4
Country	getResId	4177	1	2	0	0
CustomPhoneNumberFormattingTextWatcher	getFormattedNumber	2700	1	2	0	2
VerifyPhoneFragment	send	1071	3	10	4	0

TABLE III: Static metrics obtained for each invoked method during a test.

Metric	Unit	Description
Consumption	J	Test or method the total consumption.
Time	ms	Test or method run time.
Method Coverage	%	For test-driven instrumentation, coverage at the method level is shown.
Wifi	0-1	If Wifi was used during the execution of the monitored block.
Mobile Data	0-1	If mobile data was used during the execution of the monitored block.
Screen State	0-1	If there was interaction with the screen.
Battery Charging	0-1	If the device was charging during execution.
Avg RSSI Level	dBm	average level of RSSI obtained.
Avg Memory Usage	B	Arithmetic mean of memory consumed.
Top Memory Usage	B	Peak of memory consumed.
Bluetooth	0-1	If Bluetooth was used during the monitored block execution.
Avg GPU Load	%	Average percentage of GPU usage.
Avg CPU Load	%	Average percentage of CPU utilization.
Top CPU Load	%	Max percentage of CPU utilization.
GPS	0-1	If GPS was used during the execution of the monitored block.

TABLE IV: All dynamic metrics obtained for each test.