AIP: Scalable and Reproducible Execution Traces in Energy Studies on Mobile Devices

Olivier Nourry*, Yutaro Kashiwa†, Bin Lin‡, Gabriele Bavota‡, Michele Lanza‡, Yasutaka Kamei*
*Kyushu University, Japan, †Nara Institute of Science and Technology, Japan, ‡Università della Svizzera italiano, Switzerland

***Abstract*—Energy consumption in mobile applications is a key area of software engineering studies, since any advance could affect billions of devices. Currently, several software-based energy calculation tools can provide close estimates of the energy consumed by mobile applications without relying on physical hardware, offering new opportunities to conduct large-scale energy studies in mobile devices. In these studies, one key step of data collection is generating events, since it allows exercising specific parts of the code and, as a consequence, assessing their energy consumption. Given the fact that manually generating events by interacting with applications is time-consuming and not scalable, large-scale studies often use software-based tools to automate event generation to profile devices. Existing tools rely on randomly generated events, which undermines the reproducibility and generalizability of such studies.**

We present AIP (Android Instrumentation Profiler), an alternative to existing software-based event generation tools such as Monkey. AIP uses instrumented tests as a source of event generation, which enables the targeting of complex use cases for energy consumption estimations, as well as the creation of fully reproducible events and execution traces, while maintaining the scaling abilities of other state-of-the-art tools. The tool and demo video can be found on https://github.com/ONourry/AndroidInstrumentationProfiler.

I. Introduction

According to consumer data statistics [1], the number of mobile device users worldwide went from ∼3.6 billion in 2016 to over 6 billion in 2021 with no indications of such an increasing trend slowing down in the future. The popularity of mobile devices also boosted the market for mobile applications (or simply, apps), that is nowadays valued over $100 billion per year. These apps, besides facing all classic challenges of modern software development, must deal with one specific hardware restriction typical of mobile devices: the limited battery life. This results in the need for minimizing the energy consumption of mobile apps, which has been investigated by researchers through cataloging common energy bugs in mobile apps [2], [3], [4], [5], [6] and energy-greedy APIs [7], [8].

When conducting studies measuring the energy consumption of mobile apps, there are two possible alternatives. First, relying on hardware-based measurement approaches such as GreenMiner [9], [10]. Physical tools can precisely measure energy consumption but most of them require an external power meter and customizations of the mobile devices. In recent years, software-based approaches have been proposed, such as the PETrA technique presented by Di Nucci *et al.* [11], which uses the Android profiling tool to collect events/interactions data to estimate the energy usage of an application. While events are happening on the device, the profiling tool records every function call and outputs complete execution trace logs of every method executed when the application was running. PETrA can generate events on the device to trigger the source code and estimate the amount of joules consumed by each function, with a reported estimation error of ±4% as compared to hardware measurements [11].

Software-based solutions such as PETrA, while sacrificing some of the precision of energy consumption assessment, enable large-scale studies that would be impossible using hardware-based measurement. However, they still suffer from a major limitation posing a tradeoff between the scale of the conducted studies and their replicability. Indeed, PETrA only supports "Random Operations" or "Manual Operations" to exercise the application during the profiling process. The "Random Operations", which are generated using the Monkey tool provided by Android, make it difficult to reproduce the results of an experiment and the targeting of specific code components of interest. Whereas, "Manual Operations" and manual scenarios such as Monkeyrunner scripts result in replicable runs but make it difficult to scale out studies. Thus, both these approaches are not suitable for replicable studies involving measurements across thousands of revisions.

We present the **A**ndroid **I**nstrumentation **P**rofiler (AIP), a scalable and reproducible profiling automation tool to generate execution trace logs. AIP approximates the energy consumption during common scenarios provided by instrumentation tests, which are UI tests employed in Android application development. AIP automatically starts and finishes the profiling process by detecting the start and the end of the instrumented tests. Using our approach in combination with PETrA, it is possible to achieve reproducible and scalable energy consumption studies of mobile apps at the function-level granularity while also having the ability to target specific code components and use case scenarios. AIP is available at https://github.com/ONourry/AndroidInstrumentationProfiler together with installation and execution instructions.

II. Background

The Android Tools. The Android Profiler [12] along with Systrace [13] and Batterystats [14] are used in conjunction to collect real-time data about the usage of the CPU, memory, network, and battery resources. These Android tools do not directly measure the energy consumption of applications, but

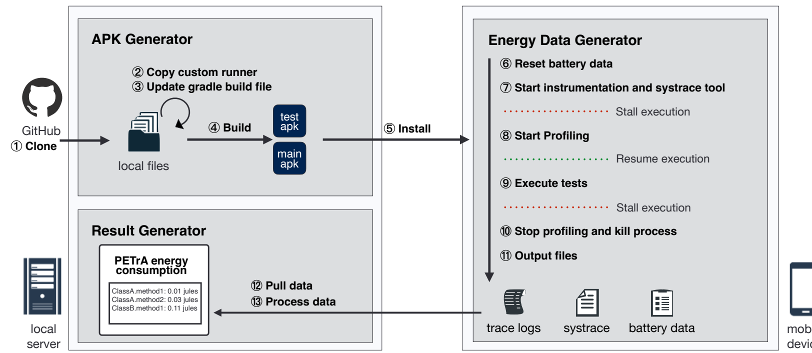

Fig. 1: Architecture of AIP.

allow the collection of data used by energy measurement models to approximate the energy usage of an application.

Instrumented Tests. Instrumented tests are tests that can run on a hardware device or an emulator [15] exploiting a special test execution environment which provides easy access to an application's context and Android framework APIs. Developers can manipulate the application under test from the test code. Instrumented tests are often used to automate user interactions. These tests are built into a separate APK (test APK) from the application APK (main APK). To detect if a repository has instrumented tests, developers just need to check whether the specific androidTest directory (located at `module-name/src/androidTest/`) contains test files.

There are multiple libraries supported by Android to write instrumented tests, such as Espresso [16], UI Automator [17] and Compose [18]. All these libraries provide a variety of APIs to test an application by interacting with its views and widgets. This makes instrumentation tests easier to implement and maintain compared to hardcoded input-by-input scripts such as those written using monkeyrunner [19]. The latter are unlikely to be maintained over time since they are often tailored to a specific app revision or specific coordinates of UI elements on the screen and are therefore very sensitive to UI changes which tend to be part of an app's evolution process. On the contrary, the flexibility offered by libraries such as Espresso which can detect and interact with an app's views directly rather than by using a coordinate system makes instrumented tests more robust against UI changes and easier to maintain and improve over time.

Because instrumented tests always generate the same events, they can consistently reproduce execution traces across runs for an application. AIP uses instrumented tests to invoke the methods of interest and investigate the energy consumption of these methods. Having access to the source code of instrumented tests also helps us target specific code components or estimate the energy consumption of complete use cases.

III. The AIP Tool

AIP - **A**ndroid **I**nstrumentation **P**rofiler, is a tool designed to combine the ability to target specific code components, generate consistent execution traces at the method level, and scale for large MSR-style studies on mobile devices. AIP enables scalability in studies aimed at measuring the energy usage of source code involved in complex use cases or requiring an interaction with multiple UI elements.

A. Relationship between AIP and PETrA

PETrA currently offers the option to use Android Monkey or monkeyrunner scripts to generate events to collect stacktraces, battery and CPU logs. It then uses an algorithm to estimate the number of joules used based on the previously collected data. However, the events generated by Monkey are random, which makes it hard to estimate the energy consumption of specific parts of code. Monkeyrunner, instead, is very sensitive to UI changes, making it difficult to scale over the historical revisions of an application. Therefore, we designed AIP as an alternative way to generate reproducible events which can target specific parts of the source code while maintaining the ability to scale over historical revisions.

B. Architecture

Fig. 1 illustrates the architecture of AIP and the interaction of its components. AIP features two main components: 1) APK Generator, and 2) Energy Data Generator. A Result Generator component such as PETrA's energy consumption model can be integrated in AIP to estimate the energy usage. The APK Generator runs on the local server, which is responsible for customizing original APKs to our needs. The customized APKs will then be installed on the mobile device. The Energy Data Generator executes the whole profiling process and generates the energy data, which will be pulled from the mobile device to the server and further processed by the Result Generator to produce the final energy consumption report.

C. APK Generator

A source code repository must be provided to the APK Generator on the local server ①. We only focus on applications with instrumented tests since we rely on them to exercise the source code for profiling. AIP uses the Android Profiler to get energy consumption data. However, the profiler cannot be executed directly from the applications. Instead, it requires instructions from the server to start/stop the profiling process. As we aim to automate the profiling process, we need to customize the APKs (mainly test APKs) to notify the server when to send such instructions. To do so, the APK Generator integrates a custom test runner into the original test APKs ②.

The detailed communication process can be found in Section III-D.

Meanwhile, it updates the gradle build files so that the custom runner is used instead of the default `AndroidJUnitRunner` while running the instrumented tests ③. The command "`gradle assemble assembleAndroidTest`" is executed to build the main APK and the customized test APK ④. Both APKs are installed on the mobile device ⑤.

D. Energy Data Generator

Before profiling, the Energy Data Generator first runs an AST parser on the source files containing the instrumentation code in the androidTest folder to extract every instrumentation test method signature. The reason behind this process is that it allows AIP to generate a set of output files for each test case profiled to avoid creating overly large files.

After extracting the method signatures, the Energy Data Generator resets the battery data: the consumed energy in joules is reset to 0 ⑥. Then, the instrumented tests and the systrace tool are executed ⑦. Systrace [13] is a utility used for analyzing the application performance by collecting execution times of application processes and other Android system processes. Meanwhile, the server is alerted that the tests will be executed soon.

Choosing when to automatically start the profiler is challenging. In practice, the profiling process can only start after the profiled process (*i.e.,* an instrumentation test) is started, otherwise an error of no process will be given. The profiler however needs time to start profiling the instrumentation tests. Due to this delay, starting the instrumentation before the profiler results in a small period during which the instrumented tests are running but the profiler has not yet started profiling. Every method call during that period are therefore not profiled and missing from the execution trace.

To address this issue, the Energy Data Generator monitors the status of the instrumentation on the device; whenever the test process is created on the device, the test execution is temporarily suspended to give time to the profiler to start the profiling process. Android Debug Bridge (adb), a command-line tool which allows AIP to communicate directly with the android device running the tests, is used to start the profiler ⑧. After five seconds, we resume the test execution so that the profiler can record every method executed on the device ⑨. During the execution, the instrumented tests generate a separate test process from the main application process, and the test process is the one to be profiled.

In our experiments we found that if the profiling stops after the profiled process is killed, we often cannot get the data from the profiler. Thus, the profiling must stop before killing the profiled process. Once the tests are completed, the Energy Data Generator temporarily suspends the test process and notifies the server that the profiler can be stopped. In this way, the profiler can generate the complete execution trace. After each instrumented test is completed, the server stops the profiler via adb and the test process is killed on the mobile device ⑩.

The execution of each instrumented test results in three log files ⑪ which are pulled from the mobile device: trace logs, systrace, and batterystats.

Trace logs contains the entire execution trace showing every method call executed during the instrumentation process. Systrace contains all information related to the CPU activity (*i.e.,* idle/active state). Finally, batterystats contains all information related to the battery usage of the mobile device during the instrumentation. These output files can then be provided to a Result Generator such as PETrA's energy consumption algorithm to get fine-grained energy approximations at the method level.

IV. AIP Usage Instructions

We provide basic instructions to run AIP. The setup details and required template files can be found in the GitHub repository https://github.com/ONourry/AndroidInstrumentationProfiler.

Environment setup. The current AIP is only for Android devices. USB debugging must be enabled on the device in the developer menu and the Android debug bridge (adb) must successfully detect the plugged device. The source code of the subject app to analyze must be available on the server to which the Android device is connected. A historical analysis of the energy consumption of an app implies the availability of the code in the form of a git repository. A Java environment must also be installed on the server with the proper environment variables set in PATH to invoke adb functions and build commands. As described in Section VI, depending on the build systems and dependencies an application uses (or used in old snapshots), the JDK, SDK and gradle versions running on the computer may need to be modified. Additional details are available on the GitHub repository.

Running AIP. AIP has one main launcher which executes both the APK Generator and the Energy Data Generator. The input arguments for the launcher are parsed from a configuration file that must be filled prior to executing it. Once the launcher is executed, AIP will first extract the instrumented test signatures from the subject app using the AST parser introduced in Section III. The APK Generator will then update the custom test runner with the app's corresponding package name, copy the custom runner into the app's androidTest folder and update the gradle build file. The APK Generator will then launch the gradle build whose logs will be displayed in the console. Once the app is successfully built, the test and main APKs will both be installed on the mobile device. After installing the APKs, the main launcher will call the Energy Data Generator which will coordinate the execution of the instrumentation process, the systrace tool and the Android profiler. Throughout the instrumentation process execution, AIP will log which tests are being profiled, when each tool starts and stops, and when the data files are pulled from the device. Whenever an instrumented test completes, the Energy Data Generator will store the data files for each test case in a directory corresponding to the test case signature.

V. Preliminary Evaluation

To evaluate the scalability of AIP, we profiled 401 different APKs from the historical revisions of the open source mobile application "KISS" [20] which can be built using gradle. The average time to profile a single APK was ∼7 minutes. However, the execution time of AIP varies based on how extensive the instrumentation tests of the subject apps are.

While experimenting on such a dataset of APKs, we found that the number of profiled methods tends to decrease when AIP is run continuously on several APKs. This means, for example, that running n times AIP on the same APK_i, we observe that x methods are profiled in each run. Then, by performing additional n runs, such a number x slightly decreases, resulting in the loss of a few data points. Reasons for this behavior are discussed in Section VI. We anticipate, however, that by rebooting the device when such a trend is observed, such variations can be minimized.

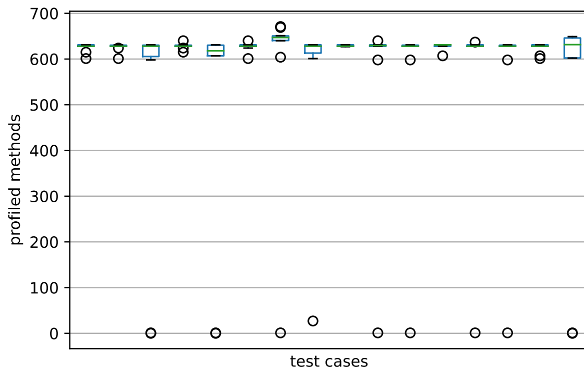

Fig. 2: Number of profiled methods by test case

To test the reproducibility of studies run through AIP, we took a specific APK from the KISS app and profiled it ten times looking at the number of methods profiled during each run for each of the 16 instrumentation tests present in the APK. A stable number of executed methods would imply stability in the data collection process. AIP generated very similar results across runs with some minor inconsistencies. Upon manual inspection, the latter was due to methods which were not consistently profiled since their execution is influenced by the application's context (*e.g.,* data handling functions whose execution varies from one run to another due to the processing time of specific queries).

Fig. 2 shows the number of methods profiled (y-axis) for each of the 16 test cases (x-axis) across the ten runs. During our manual result inspection, we also found 9 instances where the profiler was unable to properly profile a test case (out of the total 160 executed cases). These events seem to happen at random times and for no specific type of instrumentation test.

Comparison with alternatives. We developed AIP to generate reproducible results and have the ability to target source code for measurements while also being scalable. To highlight the need for a tool like AIP, we selected a real use case where a developer may want to measure the energy consumption of their application.

We compared AIP with Monkey: PETrA's current option for scalable executions. In this experiment, we used the Android application AnyMemo [21], with which users can design and create custom flash cards to memorize words, definitions and pictures. More specifically, we measured the energy consumption of "creating a new card", which is one of the most common use cases for this application. Since Monkey generates random events, we want to see if Monkey can navigate through multiple user interfaces to access the card creation menu and create a new card. For this experiment, we let Monkey generate over 30,000 interactions and then counted how many times Monkey was able to create a new flash card.

As a result, we found that Monkey was unable to create a single new card. Unlike directed executions using scripts or instrumented test code, random events were unable to trigger a common use case for the application. Although simple, this experiment highlights the need for a tool like AIP that can both scale and target specific source code. Although Monkey may at some point be able to trigger the desired code, random events make it unreliable for use cases such as the one shown in this experiment: AIP offers valuable features to target energy measurements of specific code components and conduct large scale energy studies in mobile devices.

VI. Limitations

Currently, AIP is subject to technical limitations.

Speed. The instrumentation process is noticeably slower when executed by AIP than manually. The main causes for the slowdown are the profiler monitoring the method calls on the device as well as AIP generating and pulling the execution logs after each test to avoid bloating the device's memory.

Accuracy. As discussed, AIP is susceptible to flaky behavior due to the profiler being able to profile certain methods in some runs but not in others. As described in Section V, there are also instances where tests are randomly not profiled. While experimenting with AIP, we have also observed test cases that could not be profiled on any runs.

Emulators. AIP currently only supports physical devices.

VII. Conclusion

We presented AIP, a tool to trigger the source code of mobile apps for energy measurement studies. AIP is able to target source code and generate stable results across multiple runs and revisions which opens the door to reproducible large-scale energy measurement studies. Being software-based, AIP can be easily integrated with external tools such as git to run on multiple revisions of an app's history. We expect AIP to offer new possibilities for researchers conducting large scale energy evolution studies.

Acknowledgments

We gratefully acknowledge the financial support of JSPS and SNSF for the project "SENSOR" (No. 183587, JPJSJRP20191502), JSPS for the KAKENHI grants (JP21H04877, JP21K17725), Hitachi Global Foundation for the Kurata Grants, and Kyushu University for the Leading Human Resource Development Fellowship.

References

[1] S. O'Dea. (2021, Aug) Smartphone users 2026. [Online]. Available: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/

[2] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *Proceedings of the Tenth ACM Workshop on Hot Topics in Networks*, 2011, p. 5.

[3] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, 2012, pp. 267–280.

[4] Y. Liu, C. Xu, and S. Cheung, "Where has my battery gone? finding sensor related energy black holes in smartphone applications," in *Proceedings of the 2013 IEEE International Conference on Pervasive Computing and Communications*, 2013, pp. 2–10.

[5] J. Zang, A. Musa, and W. Le, "A comparison of energy bugs for smartphone platforms," in *Proceedings of the 1st International Workshop on the Engineering of Mobile-Enabled Systems*, 2013, pp. 25–30.

[6] Y. Liu, C. Xu, S. Cheung, and J. Lu, "GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, 2014.

[7] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof," in *EuroSys'12*, 2012, pp. 29–42.

[8] M. L. Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. D. 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*, 2014, pp. 2–11.

[9] S. A. Chowdhury, A. Hindle, R. Kazman, T. Shuto, K. Matsui, and Y. Kamei, "Greenbundle: an empirical study on the energy impact of bundled processing," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 1107–1118.

[10] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: a hardware based mining software repositories software energy consumption framework," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 12–21.

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

[12] T. A. Profiler. https://developer.android.com/studio/profile/android-profiler.

[13] Capture a system trace on the command line; android developers. [Online]. Available: https://developer.android.com/topic/performance/tracing/command-line

[14] "Profile battery usage with batterystats and battery historian, android developers." [Online]. Available: https://developer.android.com/topic/performance/power/setup-battery-historian

[15] A. O. S. Project, "Instrumentation tests; android open source project," 2022. [Online]. Available: https://source.android.com/compatibility/tests/development/instrumentation

[16] Espresso, android developers. [Online]. Available: https://developer.android.com/training/testing/espresso\#kotlin

[17] Write automated tests with ui automator, android developers. [Online]. Available: https://developer.android.com/training/testing/other-components/ui-automator

[18] Testing your compose layout, jetpack compose, android developers. [Online]. Available: https://developer.android.com/jetpack/compose/testing

[19] Monkeyrunner; android developers. [Online]. Available: https://developer.android.com/studio/test/monkeyrunner

[20] Kiss. [Online]. Available: https://github.com/Neamar/KISS

[21] Anymemo. [Online]. Available: https://github.com/helloworld1/AnyMemo