EDATA
Energy Debugging And Testing for Android

Blokland, Erik; Cruz, Luís; van Deursen, Arie

DOI
10.1109/MOBILESoft66462.2025.00017

Publication date
2025

Document Version
Final published version

Published in
Proceedings - 2025 IEEE/ACM 12th International Conference on Mobile Software Engineering and Systems, MOBILESoft 2025

EDATA: Energy Debugging And Testing for Android

1st Erik Blokland
TU Delft
Delft, The Netherlands
me@erikblok.land

2nd Luís Cruz
TU Delft
Delft, The Netherlands
L.Cruz@tudelft.nl

3rd Arie van Deursen
TU Delft
Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

***Abstract*—Energy consumption of software is becoming increasingly important in today's mobile-focused world, but knowledge and techniques with which to measure energy consumption have lagged behind. This paper introduces a methodology for measuring the energy consumption of Android apps at the method level, and a concrete implementation of this methodology: EDATA. We evaluate EDATA by revisiting several Android code smells found in prior work to increase energy consumption, and by using a novel evaluation technique which allows us to generate a ground-truth for energy consumption using run-time as a proxy. Finally, we perform a case study on a real-world energy bug found in Adyen's Android point of sale (POS) software. Our findings show that EDATA is able to accurately order methods by their energy consumption, and distinguish between different versions of a method. We also observed that debug mode has inconsistent effects on energy consumption, and that energy efficiency may not be consistent between devices. Finally, our case study shows that while developers and stakeholders agree that energy consumption is important, a lack of awareness and easy-to-use profiling prevents it from becoming a first-class metric in the development process.**

***Index Terms*—Android, Green IT, Energy consumption**

I. Introduction

With the rise in popularity of mobile devices, developers must increasingly concern themselves with the energy use of their software. In contrast to traditional desktop computing devices, and even laptops, users have high expectations of their smartphone's battery, and insufficient battery life will negatively affect their satisfaction [1]. While developers have good intentions regarding improving their energy consumption, information on how to do this has historically lagged behind [2], [3].

Johnson et al. found that missing one or more of the "what, why, and how to fix" is a significant barrier in the adoption of analysis tools [4]. Nevertheless, mobile developers wishing to measure the energy consumption of their app face barriers such as a lack of actionable information and coarse granularity. Use of instrumentation-based measurement approaches to overcome this barrier, as explored in previous works [5]–[7], comes with a significant drawback: instrumentation of method calls incurs overhead, meaning that the performance and energy consumption of the application under test (AUT) may not be representative of normal behavior. ALEA, developed by Mukhanov et al. [8], addresses these barriers by using statistical sampling, which incurs low overhead while retaining high accuracy and provides insight into run-time behavior.

Our cooperation with Adyen provided insight into how experienced developers approach energy testing and debugging. Developers cited a lack of information on the energy impact of their design choices as a barrier to writing energy-aware code. We also performed a case study in cooperation with a developer who was assigned to solve an energy bug in Adyen's point of sale (POS) app, after they were unable to identify its cause using the Android Studio energy profiler.

Our goal is to overcome the limitations of currently available energy testing and debugging tools for Android by measuring the energy consumption of apps running on consumer hardware at a fine granularity, without high overhead or complicated set-up processes. These properties both decrease the barrier to adoption and give developers and stakeholders alike the ability to make an informed decision between energy consumption, functionality and effort. The importance of informing stakeholders of the energy impact of their decisions has been discussed by Jagroep et al. and Grosskop and Visser [3], [9].

With these criteria in mind, we developed Energy Debugging And Testing for Android (EDATA), an energy profiler for Android that can be used on any modern Android device. EDATA builds on ALEA [8] by modifying its approach to have method-level granularity and use platform tools available on standard Android devices. EDATA is designed to support both energy testing – the identification of energy bugs – and energy debugging – the process of fixing energy bugs.

Our evaluation of EDATA is focused on its practical use cases of energy debugging and energy testing, rather than the absolute precision of its estimations. In order to verify that EDATA can be used in real-world software development, we identified three research questions:

- **RQ1:** Can we use information collected from on-device sensors on Android devices to identify energy bugs through energy regression testing?
- **RQ2:** Can we rank methods within Android apps by their energy consumption using a callstack-sampling approach?
- **RQ3:** Does providing developers with an ordered list of methods ranked by energy consumption aid in identifying and fixing energy bugs?

In a case study with the fintech organization Adyen, we work with a non-energy-expert developer assigned to an energy bug causing unacceptable battery drain and evaluate the effectiveness of EDATA's output. This case study provides insight into whether implementing the criteria we used as the basis for EDATA has a positive effect on the ability of developers to perform energy profiling, and how it influences the decision-making process in a real-world environment.

Our evaluations show that EDATA is able to provide accurate method-level energy consumption estimates, enabling developers to profile their apps without difficult set-up or high overhead. Additionally, our case study shows the critical need for this information in the development process, and the influence that EDATA has in solving energy bugs in real-world software.

This paper makes the following contributions:

- We describe a methodology to estimate energy consumption of Android apps at the method level using on-device sensors, Section III.
- We concretely implement our methodology in EDATA, Section IV.
- We empirically evaluate the effectiveness of EDATA, showing that it is able to detect differences in energy consumption between versions of an experiment workload, and accurately order methods within an app based on their energy consumption, Section V.
- We describe a novel technique with which to generate a ground truth for energy-consumption-based method ranking, Section V-E.
- We perform a case study at Adyen using EDATA in which we assist in the energy debugging process to solve an energy bug present in Adyen's Android POS app, and discuss the current awareness of energy efficiency in the software development process, Section V-F
- We revisit some of the code smells evaluated by Palomba et al. [10] using release mode, and discuss implications for future work, Section VI.

II. Prior Work

A. Measuring Energy Consumption

There exist a number of techniques with which to measure and estimate the energy consumption of software, and to attribute energy consumption to units of source code. External hardware is commonly used to provide accurate, high resolution energy consumption measurements of both mobile devices [11], [12] and traditional desktop and laptop devices [13].

To avoid the difficulty of setting up dedicated monitoring hardware, many prior works use power consumption metrics reported by the hardware being measured. Commonly used tools include Intel's Running Average Power Limit (RAPL) [8], [14], [15] or Qualcomm's Trepn and related tools (now discontinued) [7], [16]–[19]. In general, these tools provide a relatively high level of accuracy, and can be used on any supported device without additional setup.

Some approaches forego hardware measurements entirely, instead using static/dynamic program analysis or machine learning to estimate energy consumption. Chowdhury et al. developed *GreenOracle* [20], which uses heuristics based on dynamic program behavior as the basis for an energy model, and later extended their work with *GreenScaler* [21], which implemented automatic test generation as a way to significantly speed the model training process. Other works [22] also implement similar models using dynamic program behavior. Alvi et al. [18] use source code metrics as their model features, allowing energy consumption to be estimated without any test cases or runtime environment. While this technique makes energy testing very convenient, it does not take into account the runtime behavior of the software under test.

Instrumentation provides precise information on when particular code units are used and can be implemented in source code, compiled/byte-code, or by using platform tracing tools . Potential instrumentation strategies include method-level [5], [7], path-based [6], and exclusively tracking API invocations [23].

Statistical sampling, in contrast to instrumentation, is exclusively performed at runtime, without modification to the AUT. Statistical sampling has been used on both mobile and desktop platforms [8], [13], [24], is integrated into the Android profiler[1], and provides a controllable measurement overhead regardless of the behavior of the AUT.

B. Software Engineering for Energy Efficiency

Prior work has tested the energy impact of code smells relevant to Android devices [10], [25] and explored the impact of automated refactoring on code smells found to negatively impact energy consumption [26]–[28]. Other work has investigated the effect of non-energy-aware refactoring on energy consumption [15] and the effect of energy-related commits on software maintainability [29].

Jagroep et al. [9] defined a methodology for comparing the energy consumption of different revisions of a software system at a system level. Hindle et al. [11] developed a concrete methodology for testing revisions of Android apps on real hardware.

Cruz and Abreu mined commits of open source apps to find common patterns used by developers to improve energy consumption [30], similarly to Moura et al. [31], who mined "energy-aware" commits to gather data on how developers approach energy issues.

C. ALEA

Mukhanov et al. developed ALEA, a tool used to measure the energy consumption of software at a basic-block[2] level [8]. ALEA uses a probabilistic sampling model, where the AUT is systematically sampled during execution, with each sample collects the value of the program counter and the

[1]https://developer.android.com/studio/profile/record-traces#configurations

[2]A basic block is a 'block' of code with no jump instructions - that is, it will always execute sequentially from entry to exit.

instantaneous energy consumption of the system. Each basic block is assumed to have a fixed probability of being sampled at any point in execution, and an estimate of this probability is generated based on the observed samples. The total execution time of a basic block is estimated from this probability estimate combined with the total execution time of the program. The average power consumption of a basic block is calculated by averaging its associated instantaneous measurements, using the simplifying assumption that the power consumption is only associated to the basic block executing at the time of sampling. Multi-threaded programs are modeled similarly, but use a combination of basic blocks across all sampled threads instead of a single basic block. ALEA was tested on two platforms: an Intel Sandy Bridge based server using two Xeon E5-2650 CPUs, and an ODROID-XU+E board with one Exynos 5 Octa CPU. The authors found that ALEA was able to estimate execution time and energy with an average error below 4% in all cases, for both single- and multi-threaded benchmarks.. In a follow-up to their original work, the authors improved ALEA with an additional measurement technique allowing basic blocks with a runtime down to 10 μs to be accurately estimated [32].

While ALEA was initially designed for exclusive use on traditional desktop systems, its approach offers key features worth considering in the context of mobile software development. The probabilistic sampling model allows overhead to be limited, which is critical to obtaining accurate energy consumption information when sampling must be performed on-device. There is also no need to instrument methods, which significantly alters the energy characteristics of the AUT. ALEA has also been shown to accurately attribute energy consumption to code based on built-in sensor measurements, which lowers the barrier to performing energy tests.

III. EDATA METHODOLOGY

EDATA is designed to accomplish two primary goals: providing energy consumption estimates for individual methods to facilitate energy debugging, and compare program traces to identify meaningful differences to facilitate energy testing. In this section, we describe the methodology we use to estimate the energy consumption of Android apps at the method level.

A. Data Collection

The first step in our methodology is to collect run-time energy consumption data, as well as stack samples, from a physical Android device. This step will be performed during execution of a workload, such as an automated test. The output of this step consists of a timestamped log of sensor measurements, where the measured data is sufficient to reconstruct the instantaneous power draw at each given timestamp, and a timestamped log of callchain samples.

1) Energy Consumption Data: Energy consumption data is obtained from on-device sensors, through the `BatteryManager` API. These sensors often have a very low update rate; we observed a rate of 10Hz or less for current sensors and 1Hz or less for voltage sensors.

2) Callchain Samples: To attribute energy consumption to methods, we need to use statistical sampling to collect **call stack**[3] samples, and record the currently executing method and its callchain. Android includes the *simpleperf*[4] utility, a fork of the *perf* tool for Android which is capable of performing stack unwinding on native code, as well as all forms (interpreted, ahead of time (AOT) compiled, and just in time (JIT) compiled) of JVM-based code.

B. Conversion to Intermediate Representation

To decouple the data collection methodology from the data analysis, we create an intermediate representation to be used with all following steps in our methodology. We define our intermediate representation as a series of ‘app states’, each of which contains the state of the AUT and device at the time of a callchain sample. Each entry in the series contains callchains of one actively executing thread from the AUT, the instantaneous power draw of the whole device, the time until the following sample (its ‘period’), and the entry’s timestamp.

This step contains three primary responsibilities. First, if timestamps between different logs (e.g. instantaneous power and callchain samples) are not based on the same clock, these must be corrected. Second, if the instantaneous power draw was not directly measured, it must be calculated based on the input to the conversion. Finally, the instantaneous power draw during each callchain sample must be determined, if power draw samples were not performed in sync with callchain samples. Since the sample rate of energy-related sensors may be much lower than the callchain sample rate, multiple callchain samples will likely occur during a single power measurement.

C. Intermediate Representation Analysis

The **data analysis** component reads the output of the pre-processor, and estimates the energy consumed by each method observed during execution.

When attributing energy consumption to methods, we categorize it as either **local** and **non-local**. A method’s local energy consumption is defined as the energy consumed by code directly located within a method. Non-local energy consumption is defined as energy consumed by callees (or callees thereof) of a method. For example, if method `foo()` contains a call to method `bar()`, which itself contains no method invocations, then code contained in `bar()` will be attributed as “local” to `bar()`, and “non-local” to `foo()`.

Each step in our analysis is performed twice per method – once each for local and non-local energy.

1) Determine Sample Count: First, the total number of either local or non-local samples is found from the intermediate trace representation.

[3]The call stack contains information about the currently executing method and its callchain. https://en.wikipedia.org/wiki/Call_stack

[4]https://developer.android.com/ndk/guides/simpleperf

2) Determine Average Power Draw: Next, the average power draw for the method is calculated by averaging instantaneous power measurements associated with it, as shown in Equation 1.

$$\hat{pow}_{method} = \frac{1}{n_{method}} \cdot \sum_{i=1}^{n_{method}} pow^{i}_{method} \tag{1}$$

3) Calculate Probability of Observation: We use the method defined by Mukhanov et al. [8] to estimate the probability of each method being sampled while actively executing. We reproduce their methodology here, explaining differences where appropriate. The primary change made is that we replace basic blocks with methods, as our approach only uses method-level granularity. We begin by defining the random variable X_{method}:

$$X_{method} = \begin{cases} 1 & \textit{method} \text{ is the sampled method} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

In their probabilistic model, Mukhanov et al. define CPU ticks as units of a finite population (U), and instantiate X_{method} by sampling during a particular clock cycle. We instead use nanoseconds as our units, to avoid the difficulties associated with variable clock speed, where CPU ticks do not have a definite time period associated with them. The probability that *method* is sampled is thus:

$$p_{method} = P(X_{method} = 1) = \frac{C^1_{t_{method}}}{C^1_{t_{exec}}} = \frac{\sum_{j=1}^{k} latency^{j}_{method}}{t_{exec}} \tag{3}$$

$$\frac{\sum_{j=1}^{k} latency^{j}_{method}}{t_{exec}} = \frac{t_{method}}{t_{exec}} \tag{4}$$

We define t_{method} as the total CPU time of a given method. To approximate t_{method}, we multiply the probability of the method being observed, p_{method}, by the total CPU time of the app, as in Equation 5.

In contrast to Mukhanov et al [8], we define t_{exec} as the sum total of the *periods* of each sample. We do not sample using wall-clock time or a global metric such as `cpu-cycles`, and this change is necessary to ensure that t_{exec} keeps the same relationship to the callchain samples.

Equation 3 thus represents the probability of sampling a method equaling the ratio of its CPU time to the total CPU time of the app.

$$t_{method} = p_{method} \cdot t_{exec} \tag{5}$$

Using the same process as Mukhanov et al. we find the maximum likelihood estimator $\hat{p}_{method}$ for $X_{method} = 1$ in Equation 6, where n is the total number of samples collected, and n_{method} is the number of samples where *method* was sampled.

$$\hat{p}_{method} = \frac{n_{method}}{n} \tag{6}$$

With $\hat{p}_{method}$, we are able to estimate the method's total runtime, $\hat{t}_{method}$, in Equation 7.

$$\hat{t}_{method} = \hat{p}_{method} \cdot t_{exec} = \frac{n_{method} \cdot t_{exec}}{n} \tag{7}$$

4) Estimate Energy Consumption: Finally, we estimate the total energy consumed by a given method by multiplying the maximum likelihood estimators of its average instantaneous power draw and runtime.

IV. EDATA Implementation

In this section, we describe the concrete implementation of EDATA, including our assumptions and implementation choices that are not inherent to our methodology.

A. Data Collection

To perform the data collection described in Section III-A, we periodically sample the call-stack of the AUT along with the instantaneous power draw of the device. The data from Android devices' voltage and current sensors are often exclusively available through Java/Kotlin APIs, whereas *simpleperf* can only be used through the Android Debug Bridge (ADB) shell.

Due to these restrictions, we implemented our data collection with two sub-components: Power consumption data is collected through a companion app using the Android `BatteryManager` API, and app callchains are collected using *simpleperf*. Combined, our data collection tools output a timestamped log of current and voltage samples and the standard `perf.data` file output by *simpleperf*. By using the `monotonic_raw` clock with *simpleperf*, we are able to use the same system clock between both logs, removing the need to synchronize these samples in a later stage.

B. Intermediate Representation

We implement the process described in Section III-B as shown in Figure 1. As we obtain separate current and voltage measurements from our sampler app, we implement step 2, 'create log of calculated instantaneous power', by creating a time-series of instantaneous power draw updated each time either the voltage or current reported by the device changes. Step 5, 'Determine instantaneous power consumption during each sample', is performed by finding the most recent entry in the time-series **prior** to the sample.

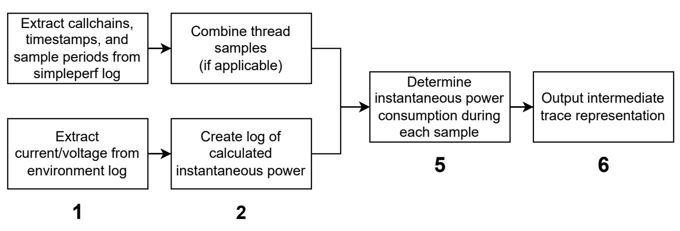

3
Extract callchains, timestamps, and sample periods from simpleperf log
4
Combine thread samples (if applicable)
Extract current/voltage from environment log
1
Create log of calculated instantaneous power
2
Determine instantaneous power consumption during each sample
5
Output intermediate trace representation
6

Fig. 1: Abstraction Process

In our implementation, we have chosen to simplify the attribution process by assuming that the AUT uses no more than one thread at a time. This allows us to assume that each sample reported by *simpleperf* was the only thread executing on the device at sample time, and allows EDATA to skip step 4 in Figure 1, 'Combine thread samples'. While our empirical evaluation exclusively uses single-threaded workloads, our case study does not impose any such restriction. Informal observation of other Android apps indicated that most time is spent either idling or executing a single thread, limiting the impact of using a single-threaded model.

In addition, we have chosen to use the sample period reported by *simpleperf* when using the `task-clock` event instead of computing the true wall-time between two samples. This sample period represents the amount of CPU time between samples of the same thread. This entails that two consecutive samples may have a different period than the total wall or CPU time between them at an application level.

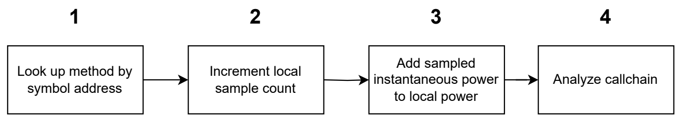
C. Data Analysis

Our implementation of the approach described in Section III-C consists of two phases: method extraction, and post-processing. The method extraction phase is responsible for collecting information on each method observed in the list of app states output by the process described in Section III-B. This phase corresponds with 'Determine Sample Count', Section III-C1, and 'Determine Average Power Draw', Section III-C2. Once this phase is complete, we have sufficient information for each method with which to perform the rest of the process. In this section, we use the definition of 'local' and 'non-local' energy consumption defined in Section III-C.

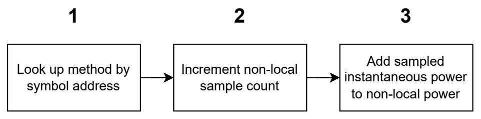
1) Method Extraction: The first step of our implementation is responsible for extracting information on each observed method from the sampled callchains. Our implementation follows the process shown in Figure 2a, which is performed once per sampled app state. During this process, we maintain two global variables: a dictionary mapping `Function` objects, which are our representation of methods, to their memory address, as well as a sum of the periods of each app state.

In step 4, we analyze the callchain of the sampled app state. For each unique entry in the callchain, we perform the process shown in Figure 2b. We filter duplicate callchain entries to ensure that methods that appear multiple times in the callchain do not have non-local run-time attributed to them more than once per sample.

2) Post-Processing: The second step of our implementation is the 'post-processing' phase, which implements the steps defined in Sections III-C3 and III-C4. To perform this phase, we iterate over each method in the dictionary, and perform the four steps shown in Figure 3 for local and non-local samples.


1
2
3
4
Look up method by symbol address
Increment local sample count
Add sampled instantaneous power to local power
Analyze callchain

(a) App State Analysis


1
2
3
Look up method by symbol address
Increment non-local sample count
Add sampled instantaneous power to non-local power

(b) Callchain Analysis

Fig. 2: Method Extraction Phase

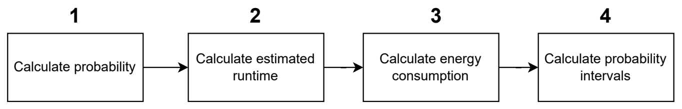

1
2
3
4
Calculate probability
Calculate estimated runtime
Calculate energy consumption
Calculate probability intervals

Fig. 3: Post-Processing Phase

V. Evaluation

The goal of our evaluation is to prove the effectiveness of EDATA in real-world energy testing and debugging scenarios, where a developer is using an off-the-shelf Android device in a relatively consistent environment. To this end, we defined the following two research questions:

> **RQ1:** Can we use information collected from on-device sensors on Android devices to identify energy bugs through energy regression testing?
>
> **RQ2:** Can we rank methods within Android apps by their energy consumption using a callstack-sampling approach?

Combined, these research questions show that EDATA is capable of both identifying whether an energy bug has been introduced into an Android app, and providing developers with an indication of where this bug may lie in their code. These two properties, combined with EDATA's low overhead, provide a significant advantage over current state of the art energy tooling for Android.

Finally, we perform a real-world case study to evaluate the effectiveness of EDATA on a real energy bug to determine whether the results of our empirical evaluation transfer to the real world, and pose the following research question:

> **RQ3:** Does providing developers with an ordered list of methods ranked by energy consumption aid in identifying and fixing energy bugs?

A. Test Devices

We used two devices in our evaluation: a Google Pixel 6a and an Adyen AMS1. These devices significantly differ in both hardware and software, which gives us some insight into how energy consumption characteristics can differ based on the device being used. Our case study was performed exclusively on the AMS1, as it involved purpose-built Adyen software. Though we originally targeted EDATA to the AMS1, we chose to include the Pixel 6a as it is publicly available commodity hardware and can be used by others to reproduce our results. It also allows us to compare our results across different devices, and gain insight into potential effects of hardware and software on energy consumption.

1) Device Specific Information and Specifications: **AMS1** The AMS1 contains some background services which cannot be disabled. It is possible, and likely, that these services affect the energy consumption of the device, and therefore disturb the measurements taken. As the AMS1 specifications are not public knowledge, we have intentionally not provided them here.

Pixel 6a The Pixel 6a test device was factory reset prior to testing, and was not signed into a Google account. Airplane mode was enabled, and no SIM card or eSIM was present in the device.

B. Empirical Methodology

We derive the following methodology from prior work, and follow it for each of our empirical tests. We perform 20 iterations of each test on the AMS1, and 30 iterations on the Pixel 6a. We performed fewer iterations on the AMS1 due to time constraints, as the specifics of the platform caused the testing process to take longer.

1) Pre-experiment setup: Previous work highlights the importance of reducing the impact of environmental factors when measuring energy consumption, and provide a number of guidelines on how to do this [5], [9], [12], [33]–[35]. Our approach, described below, is in line with prior work.

Following the examples laid out above, the test devices used for evaluation are set up as follows: We remove all third party apps to the extent possible, either through regular un-installation or by performing a factory reset. The cellular modem is disabled by use of airplane mode. Bluetooth is disabled through the system toggle, with the exception of test cases that make use of Bluetooth hardware. The device's display is off, except for tests involving Bluetooth scanning, as the display must be switched on to scan for devices.[5]

2) Pre-test setup: To prepare for a test loop, we first install the app under test using ADB. We then AOT compile[6] the full package on the device under test, and run a shortened version of our test case for one minute to allow the JIT compiler to optimize the app.

3) Test Loop: EDATA is used to run the test for the specified number of iterations, performing all necessary set-up and tear-down steps such as activating *simpleperf* and our sampler app, and copying their results to the host machine.

4) Post-Loop: This step is performed after a test loop has been completed, and is the last step taken by the test orchestrator. In this step, we remove the AUT from the device, clearing its data and JIT cache. This ensures that the environment will remain the same between different tests, making our process more consistent.

[5]https://developer.android.com/reference/android/bluetooth/le/BluetoothLeScanner#startScan(android.bluetooth.le.ScanCallback) (Accessed on February 2024)

[6]AOT compilation is used to reduce the performance impact of testing freshly installed apps, as most Android devices will partially or fully compile the app during device downtime. For more information, see https://source.android.com/docs/core/runtime/jit-compiler#architectural-overview (Accessed on February 2024)

C. Case Study Methodology

Our case study focuses on a mobile developer from Adyen who used our tool to identify energy "hot spots" in their code. To perform this case study, we identified a use case known to cause excessive battery drain in cooperation with developers responsible for maintaining a particular app. Once these use cases were identified, we manually performed test scenarios while running our tool and provided the results to the developers in CSV form. Due to restrictions regarding the app build process, it was necessary to perform these tests with the app compiled in debug mode.

D. RQ1: Can we use information collected from on-device sensors on Android devices to identify energy bugs through energy regression testing?

We evaluated RQ1 by defining two sets of test cases, with each test case having two or more variants with different expected energy consumption.

1) Code Smells: The first set of test cases involve code smells known to increase energy consumption [10]: slow for loop (for), internal setter (IS), and member ignoring method (MIM). The goal of these tests is twofold: as these code smells have been previously observed to increase energy consumption of Android devices, we expect to observe a similar increase using EDATA. Secondly, we want to determine whether or not the energy impact observed by Palomba et al. [10] still holds on modern devices running in release mode, to gain insight into the impact of debug mode on energy consumption. To this end, we test "fixed" and "unfixed" versions of each code smell using both release and debug mode.

2) Hardware Components: The second set of test cases involve device hardware components – the accelerometer and Bluetooth controller – that are expected to display different energy characteristics depending on how they are used by the AUT. To change the energy characteristics of these two hardware components, we vary the update interval requested from the accelerometer and perform Bluetooth LE scans of varying length. In order to make these differences visible to our tool, which requires actively running threads to measure energy consumption, we will run a basic CPU-based workload at regular intervals. Since this workload will be consistent across all configurations of the test, we do not expect it to influence our results. Though we do not believe that prior work has validated that these differences will affect the energy consumption of our test cases, Bluetooth LE scans are known to consume significant energy[7], and it is recommended to minimize use of sensors, including the accelerometer, to preserve battery life[8].

3) Results: In our code smell tests, we found clear differences in energy consumption for fixed and unfixed versions of the MIM and IS code smells in both release and debug mode ($p \ll 0.01$ between fixed and unfixed in both release

[7]https://developer.android.com/guide/topics/connectivity/bluetooth/find-ble-devices

[8]https://developer.android.com/guide/topics/sensors/sensors_overview

and debug mode), with debug mode having a significant – and inconsistent – impact. We additionally found that certain optimizations appear to be entirely disabled when using debug mode, as our IS test case using `private` visibility showed no additional energy consumption in release mode, but significant increase in debug mode.

In contrast to the other two code smells, we did not detect a statistically significant difference between fixed and unfixed versions of the for code smell on either device, but found a difference when using debug mode on both devices ($p \ll 0.01$). However, on the Adyen AMS1, we observed an *increase* in energy consumption in the fixed version instead of a decrease. These results indicate that the energy characteristics of the for code smell may have changed since it was first analyzed by Palomba et al. [10].

EDATA was able to detect statistically significant differences in energy consumption between each of the chosen update rates in the accelerometer test on each device, with $p < 0.01$ for all comparisons.

We were unable to obtain consistent results in our Bluetooth-based test – when using 60% sleep, we did not identify significant differences between scan lengths on either device. To make the difference in energy consumption more visible to EDATA, we re-ran our tests using 0% sleep on both devices, and were able to identify an *increase* in energy consumption on the Pixel 6a when using a longer active scanning period ($p = 0.001$), and a decrease on the AMS1 ($p = 0.047$). These inconsistencies may be caused by the lack of direct control of scanning behavior provided by the Android framework, as there is no guarantee that the actual active period matches the intended period used by the test app.

Given the positive results of our test suite, we conclude that the answer to RQ1 is **yes**: It is possible to use the information collected from on-device sensors on Android devices to identify energy bugs.

4) Impact of Debug Mode: In our experiments we observed that not only does debug mode cause significant overhead when compared to release mode, the overhead is inconsistent between different workloads and devices. Figure 4 shows the energy consumption overhead caused by the use of debug mode for each code smell test case on the Pixel 6a. The overhead is not only inconsistent between workloads on the same device, but identical workloads executed on different devices display different amounts of overhead. We observed that the for smell incurred no overhead, or even negative overhead, on the AMS1, in contrast to the Pixel 6a.

E. RQ2: Can we rank methods within Android apps by their energy consumption using a callstack-sampling approach?

We evaluated RQ2 using a novel technique which uses execution time as a proxy for energy consumption. This approach allows us to build a ground truth with which to compare our energy consumption estimates without the need for high-accuracy hardware measurements. To generate this ground truth, we use a random selection of six workload classes with identical workloads.

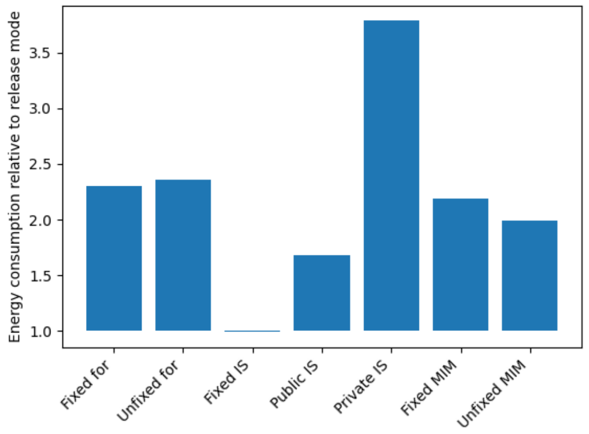


Fig. 4: Overhead Incurred by Debug Mode on Pixel 6a

At the beginning of the test, the random workload randomly generates a probability for each of the six classes to be chosen during each timestep. This ensures that there will be measurable differences between the different classes. During execution of the workload, in steps of the chosen time interval, a selection is made between actively working or sleeping. If active work is selected, then a function is called to randomly select one of the classes using the generated probabilities. The execution time of each workload class is recorded such that the total execution time of each class's workload is known after the test is complete.

We perform this evaluation with the following parameter values: with a two minute long test, we used 100Hz, 10Hz, and 2Hz sample rates with 10ms and 100ms timesteps. with a ten minute long test, we use 100Hz and 10Hz sample rates with a 10ms timestep, and a 2Hz sample rate with 10ms and 100ms timesteps.

1) Execution Time as Proxy for Energy Consumption: Relating execution time to energy consumption is controversial, and prior work can be found which both agrees [36] and disagrees [21], [37] with this relation. Corral et al. [36] used a set of CPU and RAM intensive benchmarks on an Android device, and concluded that for these cases, execution time was directly correlated with energy consumption. Hao et al. [37] investigate the energy consumption of real-world apps, which have different execution characteristics than synthetic benchmarks. For example, real-world apps generally make network requests, which are a known cause of high energy consumption [38]. They concluded that there is little to no relation between execution time and energy consumption in the context of mobile apps. Chowdhury et al. agree with this [21], and further note that decreasing the execution time by way of performance optimizations may cause the CPU to be put into a higher frequency state, increasing its power draw and raising (or failing to lower) energy consumption.

Of these works, our validation process most closely resembles that of Corral et al. [36], ruling out the confounding

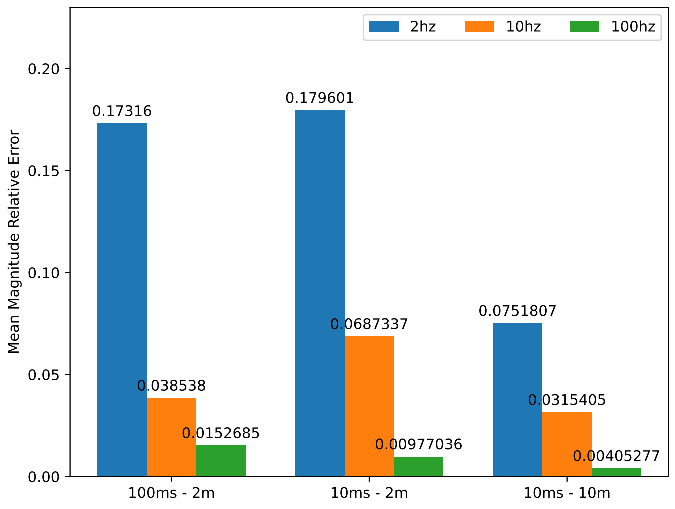


Fig. 5: MMRE of method-level energy estimates on AMS1

effects of network requests, display state, and processor power states

We also note the difference between execution time – commonly understood as the wall-clock time of a program or thread's execution – and CPU time – the time that a program/thread is scheduled on the CPU. Our test uses execution time as a baseline, as wall-clock timestamps are taken at the beginning and end of each timestep. The configuration we use with *simpleperf*, however, uses CPU-time-based timers, which do not increment if a thread is scheduled off-CPU. Since we are measuring energy consumption, not CPU-time, we do not consider this to be a threat to the validity of our evaluation.

As we have aligned our validation process with prior work [36] and ensured that execution time is not directly measured, we consider execution time to be a reliable proxy for energy consumption in the context of this evaluation.

2) Results: We obtained a mean magnitude of relative error (MMRE) under 0.25 for all tests on both devices, with a maximum MMRE of 0.213 on the Pixel 6a and 0.180 on the AMS1 (Figure 5). An MMRE of 0.25 is commonly considered to be an adequate upper bound for estimation accuracy [5], [39]. We observed that both increasing the run-time of the test and increasing the sampling frequency were effective in reducing the MMRE of the results, allowing greater flexibility in testing cases where either sample frequency or runtime are restricted. We were surprised to observe a lower MMRE in all scenarios on the AMS1 instead of the Pixel 6a, but attribute this to the lower current sampling rate of the AMS1 causing estimated power consumption to be more dependent on measured runtime, instead of measured current. This could lower the MMRE due to our use of execution time as a proxy for energy consumption.

In light of these results, we answer **yes** to RQ2, and conclude that we are able to rank methods within Android apps by energy consumption using a callstack-sampling approach. However, in light of our decision to use the run-time as a proxy for energy consumption, and the differences we found between the AMS1 and Pixel test devices, future work should investigate whether our results transfer to more complex, real-world apps in which run-time may not be directly associated with energy consumption.

F. RQ3: Does providing developers with an ordered list of methods ranked by energy consumption aid in identifying and fixing energy bugs?

1) Case Study Progression: In addition to our empirical evaluation, we performed a case study in cooperation with a developer at Adyen, where we worked to identify and fix an energy bug causing high battery drain during device idle. A developer (Developer A) had previously attempted to solve the issue, but was unable to identify the root cause.

We began our evaluation by defining a suitable test case: since the energy bug caused excessive battery drain at idle, we allowed the device to sit with its display off for a period of time while EDATA recorded the Adyen app's energy consumption. With this test case, we performed an initial analysis of the app, and presented our findings to Developer A. We identified one of the highest-consuming methods in the app, and Developer A created a build where the functionality implemented by this method was disabled. After testing, we observed a downwards shift in the energy consumption distribution with a reduction in mean energy consumption from 147.7J to 116.0J as depicted in Figure 6 by 'Fix 1'. However, the median energy consumption remained nearly identical, moving from 126.3J to 125.9J. In spite of this improvement, we were still unable to explain the excessive battery drain.

By comparing our total attributed energy consumption with the approximate total energy consumption of the device (based on the battery capacity and time to empty), we observed that the energy consumption attributed to the Adyen app was less than a third of the total energy being consumed by the device. We therefore concluded that the energy bug is not caused by code execution within the app, but by some other factor. We used the Android Studio profiler to identify a wakelock being held by the app and reported this information to Developer A, who confirmed that this wakelock was the cause of the energy bug. We performed another set of tests to confirm this, and a comparison between each version of the app can be seen in figure 6.

2) Conclusion: EDATA was able to identify the source of the energy bug, an architectural decision that was already identified as a candidate for refactoring unrelated to energy consumption by another developer familiar with the software. The high effort required to make these changes meant that they were previously unable to find support, but the detailed energy consumption information provided by EDATA convinced the stakeholders of the POS app that these changes were necessary.

While Developer A had previously observed energy consumption spikes in the Android Studio profiler, they lacked sufficient context to determine whether these spikes were related to the energy bug or if they were reasonable for the work being performed. EDATA's attribution of energy consumption to individual methods allowed Developer A to

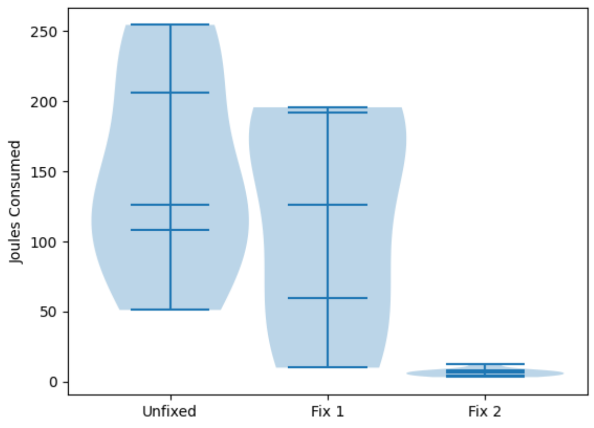


Fig. 6: Energy Consumption of Adyen POS app - AMS1

quickly identify energy hotspots and leverage their knowledge of the Adyen POS app to judge whether the relative energy consumption of a method was appropriate for its functionality.

Therefore, we conclude that we have affirmatively answered **RQ3**, and confirm that adding energy ranked data at the method-level can be an effective way to assist developers in improving the energy efficiency of their apps.

VI. Discussion

A. Implications

1) Build Mode and Code Smell Observations: The inconsistent overhead incurred by the use of debug mode, as discussed in Section V-D4, has significant implications for prior works in which debug mode was used and for practitioners measuring energy consumption of their Android apps. As the overhead caused by debug mode is unpredictable, not only are the absolute energy consumption values obtained from debug mode builds unreliable, but the ordering of methods with respect to their energy consumption cannot be assumed correct. As many existing approaches to measuring energy consumption on Android devices, such as the use of method instrumentation, require the use of debug mode, its inherent overhead poses significant challenges to energy testing and debugging on the Android platform. In addition, much of the existing literature on the energy consumption of code smells on Android does not specify which mode was used, and our observations imply that this information is necessary to interpret the results.

2) Importance of Energy in the Development Process: Our case study highlights the importance of providing specific, contextualized information about energy efficiency in the development process, and how existing tools used by Android developers fall short of this goal. As discussed in Section V-F, EDATA is able to significantly ease the process of finding and fixing an energy bug in the Adyen POS software. We attribute this in large part to the detailed, prioritized information provided, which allowed us and Developer A to immediately see which parts of the app consumed the most energy in the test scenario, providing a clear path forward in the debugging process.

A common thread between our conversations with Developer A and other mobile developers at Adyen is that, while energy efficiency is a concern, there is a lack of knowledge available on how architectural decisions affect energy consumption. Due to this lack of information, energy consumption is thus not considered as a first-class citizen in the decision-making process. This finding mirrors that of Grosskop and Visser [3], who found that many stakeholders had a low level of awareness of the importance of software to energy consumption. This similarity, ten years after their study, shows that while mobile devices have become significantly more prominent in everyday life, awareness of energy consumption in the development process lags behind. Software development teams that are concerned with energy should consider appointing an energy advocate, who will make developers aware of the energy consumption of their code and the effect it has on the products they create. In addition, more incentives for teams to prioritize energy efficiency in their development process should be put in place.

B. Limitations

EDATA requires the use of a physical device and some form of workload to test. Though the specifics of the workload can be left to developers, there will always be non-trivial time and expense involved in using EDATA. In cases where this limitation is unacceptable, use of a static analysis based approach may be appropriate, as such an approach typically only requires access to source code.

EDATA also does not make any attempt to determine whether or not a change in energy consumption is acceptable for a given code change, leaving this determination up to the user. In many cases, trade-offs must be made between code quality, performance, and efficiency, and determining whether or not a trade-off is acceptable is out of the scope of EDATA. In Section II-B , we discuss existing work to catalog energy patterns and tools created to automatically identify and fix energy anti-patterns. The use of these tools, and knowledge of energy patterns, can help developers determine whether their energy consumption for some functionality is optimal, or whether there is room for improvement.

One of the difficulties posed by mobile platforms in general is the wide variation in operating conditions, with variations in Wi-Fi and cellular signal strength, display brightness, and other factors having a strong influence on energy consumption. This makes comparison of data collected from end-users extremely difficult. EDATA does not yet take these differences into account, and developers using it must ensure that comparisons are only made between similar conditions.

In our case study, one of the points of feedback we received was that the energy consumption of hardware is also a target for improvement, in addition to software. Prior work has attributed energy consumption to hardware [40]. Further, since the Pixel 6, Google's Pixel devices ship with sensors that report the power draw of individual hardware components[9], removing the need for estimation. The ability to account for energy spent on different hardware components would also allow EDATA to normalize its output based on the status of the hardware, improving the generality of results.

C. Threats

1) Internal Validity: As we do not have full control over the Android image on our test devices, software other than the AUT may run during our tests, introducing potential confounding effects. We mitigate this effect by performing each test at least 20 times to mitigate the impact of outliers. In addition, we factory reset our Pixel 6a test device before our evaluation.

The sampling method used with *simpleperf* requires a minimum threshold of run-time before a thread's activity is sampled, meaning that threads with short lifespans or methods only called at thread start may fall completely out of view. In our case study, we mitigate this by greatly increasing the sample rate used by *simpleperf*. We do not expect this to influence our empirical evaluation, as we do not use short-lived threads and the methods tested run for the full duration of the test.

2) External Validity: Our empirical evaluation consists of a set of isolated tests, where each test evaluates either a single change which has been isolated in a test case, or in the case of our method ordering test, a small known set of methods that are each executed in a controlled manner. We consider these tests to be an effective first step in validating EDATA, particularly in combination with our real-world case study. Nevertheless, due to the extensive potential combinations of app behaviors and devices, we cannot conclusively show that EDATA performs with the same level of accuracy in all scenarios.

Our case study was performed in cooperation with a single team, on one particular app (the Adyen POS app). We consider this to be a good representation of a real-world development environment – Adyen's POS software is mature, and the developers and stakeholders involved in our case study all have industry experience. Nevertheless, there are limitations to generalizing a single case study, and differing experience levels in development teams and software use-cases could influence their perception of EDATA, and what information they find most useful.

D. Future Work

As discussed in Section VI-B, hardware energy consumption is interesting to Android developers. Future work should develop an approach to perform this attribution without the high overhead costs of instrumentation inherent to prior approaches, and make use of the new 'power rails' available on some devices. Hardware manufacturers should provide individual power rails like those on Pixel devices, so that developers can make use of these features regardless of the device they choose to test on.

[9]https://developer.android.com/studio/profile/power-profiler#examples

In our case study, we observed that it is helpful to contextualize energy consumption in terms of the device's battery – something we did manually when discussing the results of test runs. Future work should ensure that the output of tools designed to measure energy consumption is easy to understand for developers who are not familiar with the field.

Use of a "clustering" technique, where environmental characteristics such as signal strength and display brightness are taken into account, can assist in comparing data collected from separate test runs. Such a technique could allow energy profiling of end-user devices and improve reliability of energy regression testing by detecting environmental changes.

Our observations in Section VI-A1 show that there are significant, measurable differences between the effects of debug mode on different code smells, and across different devices. Additional research is needed to confirm our observations, understand the origin of these differences, and determine whether the current understanding of how software metrics such as code smells relate to energy consumption transfers to release mode builds.

VII. Conclusion

This work introduces a methodology to measure the energy consumption of Android apps at the method level, as well as a concrete implementation of this methodology: EDATA. Our results show that, using statistical sampling of callchains combined with on-device power sensors, our approach is able to accurately identify differences in energy consumption between versions of a test case, order methods by their energy consumption, and has been used to find and fix a real-world energy bug in Adyen's POS software. This work is a first step in the process of building a fully-featured method-level energy profiler for Android, and we hope that future work will continue to build upon EDATA to realize the goal of elevating energy efficiency to be a first-class citizen in the development process.

VIII. Data Availability

Data produced by the Adyen AMS1 will not be made available to comply with relevant non-disclosure agreements.

References

[1] R. Mu, Y. Zheng, K. Zhang, and Y. Zhang, "Research on customer satisfaction based on multidimensional analysis," p. 605, 2021. [Online]. Available: http://dx.doi.org/10.2991/ijcis.d.210114.001

[2] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," May 2014. [Online]. Available: http://dx.doi.org/10.1145/2597073.2597110

[3] K. Grosskop and J. Visser, "Identification of application-level energy optimizations," *Proceeding of ICT for Sustainability (ICT4S)*, vol. 4, pp. 101–107, 2013.

[4] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" May 2013. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2013.6606613

[5] D. Di Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. De Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?" in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 103–114.

[6] D. Li, S. Hao, W. G. J. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2483760.2483780

[7] M. U. Farooq, S. U. Rehman Khan, and M. O. Beg, "Melta: A method level energy estimation technique for android development," Nov 2019. [Online]. Available: http://dx.doi.org/10.1109/ICIC48496.2019.8966712

[8] L. Mukhanov, D. S. Nikolopoulos, and B. R. De Supinski, "Alea: Fine-grain energy profiling with basic block sampling," in *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 2015, pp. 87–98.

[9] E. A. Jagroep, J. M. van der Werf, S. Brinkkemper, G. Procaccianti, P. Lago, L. Blom, and R. van Vliet, "Software energy profiling," May 2016. [Online]. Available: http://dx.doi.org/10.1145/2889160.2889216

[10] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia, "On the impact of code smells on the energy consumption of mobile applications," p. 43–55, Jan 2019. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2018.08.004

[11] A. Hindle, "Green mining: A methodology of relating software change to power consumption," in *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 78–87.

[12] 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*, 2014, pp. 2–11.

[13] J. Flinn and M. Satyanarayanan, "Powerscope: A tool for profiling the energy usage of mobile applications," in *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*. IEEE, 1999, pp. 2–10.

[14] R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "Spelling out energy leaks: Aiding developers locate energy inefficient code," p. 110463, Mar 2020. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2019.110463

[15] Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Tales from the code# 2: A detailed assessment of code refactoring's impact on energy consumption," in *Software Technologies: 16th International Conference, ICSOFT 2021, Virtual Event, July 6–8, 2021, Revised Selected Papers*. Springer, 2022, pp. 94–116.

[16] R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 425–436.

[17] B. Westfield and A. Gopalan, "Orka: A new technique to profile the energy usage of android applications," in *2016 5th International Conference on Smart Cities and Green ICT Systems (SMARTGREENS)*. IEEE, 2016, pp. 1–12.

[18] H. M. Alvi, H. Majeed, H. Mujtaba, and M. O. Beg, "Mlee: Method level energy estimation — a machine learning approach," p. 100594, Dec 2021. [Online]. Available: http://dx.doi.org/10.1016/j.suscom.2021.100594

[19] M. Couto, R. Pereira, F. Ribeiro, R. Rua, and J. Saraiva, "Towards a green ranking for programming languages," Sep 2017. [Online]. Available: http://dx.doi.org/10.1145/3125374.3125382

[20] S. A. Chowdhury and A. Hindle, "Greenoracle," May 2016. [Online]. Available: http://dx.doi.org/10.1145/2901739.2901763

[21] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, "Greenscaler: training software energy models with automatic test generation," p. 1649–1692, Jul 2018. [Online]. Available: http://dx.doi.org/10.1007/s10664-018-9640-7

[22] S. Romansky, "Estimating fine-grained mobile application energy use based on run-time software measured features," 2020.

[23] 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," May 2014. [Online]. Available: http://dx.doi.org/10.1145/2597073.2597085

[24] N. R. Tallent, J. M. Mellor-Crummey, and M. W. Fagan, "Binary analysis for measurement and attribution of program performance," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 441–452, 2009.

[25] L. Cruz and R. Abreu, "Performance-based guidelines for energy efficient mobile applications," May 2017. [Online]. Available: http://dx.doi.org/10.1109/MOBILESoft.2017.19

[26] E. Iannone, F. Pecorelli, D. Di Nucci, F. Palomba, and A. De Lucia, "Refactoring android-specific energy smells," Jul 2020. [Online]. Available: http://dx.doi.org/10.1145/3387904.3389298

[27] L. Cruz and R. Abreu, "Improving energy efficiency through automatic refactoring," p. 2, Aug 2019. [Online]. Available: http://dx.doi.org/10.5753/jserd.2019.17

[28] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol, "Earmo: An energy-aware refactoring approach for mobile apps," p. 1176–1206, Dec 2018. [Online]. Available: http://dx.doi.org/10.1109/TSE.2017.2757486

[29] L. Cruz, R. Abreu, J. Grundy, L. Li, and X. Xia, "Do energy-oriented changes hinder maintainability?" in *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2019, pp. 29–40.

[30] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, vol. 24, pp. 2209–2235, 2019.

[31] I. Moura, G. Pinto, F. Ebert, and F. Castor, "Mining energy-aware commits," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 2015, pp. 56–67.

[32] L. Mukhanov, P. Petoumenos, Z. Wang, N. Parasyris, D. S. Nikolopoulos, B. R. De Supinski, and H. Leather, "Alea," p. 1–25, Mar 2017. [Online]. Available: http://dx.doi.org/10.1145/3050436

[33] F. Bouaffar, O. L. Goaer, and A. Noureddine, "Powdroid: Energy profiling of android applications," Nov 2021. [Online]. Available: http://dx.doi.org/10.1109/ASEW52652.2021.00055

[34] L. Cruz, "Green software engineering done right: a scientific guide to set up energy efficiency experiments," http://luiscruz.github.io/2021/10/10/scientific-guide.html, 2021, blog post.

[35] L. Cruz and R. Abreu, "On the energy footprint of mobile testing frameworks," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[36] L. Corral, A. B. Georgiev, A. Sillitti, and G. Succi, "Can execution time describe accurately the energy consumption of mobile apps? an experiment in android," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, 2014, pp. 31–37.

[37] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th international conference on software engineering (ICSE)*. IEEE, 2013, pp. 92–101.

[38] S. Rosen, A. Nikravesh, Y. Guo, Z. M. Mao, F. Qian, and S. Sen, "Revisiting network energy efficiency of mobile apps," Oct 2015. [Online]. Available: http://dx.doi.org/10.1145/2815675.2815713

[39] L. C. Briand and I. Wieczorek, "Resource estimation in software engineering," Jan 2002. [Online]. Available: http://dx.doi.org/10.1002/0471028959.sof282

[40] A. Cornet and A. Gopalan, "A software-based approach for source-line level energy estimates and hardware usage accounting on android," in *The Eighth International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies, Nice, France,(32-37)*, 2018.

[41] 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 *Proceedings of the 7th ACM european conference on Computer Systems*, 2012, pp. 29–42.