EDATA: Energy Debugging And Testing for Android

Erik Blokland

TU Delft

EDATA: Energy Debugging And Testing for Android

by

Erik Blokland

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday June 20, 2023 at 2:00 PM.

Student number:	4475798	
Project duration:	November 14, 2022 – June 20, 2023	
Thesis committee:	Prof. dr. ir. A. van Deursen,	TU Delft, supervisor
	Dr. L. Cruz,	TU Delft, daily supervisor
	Dr. R. Hai,	TU Delft

TU Delft

Preface

This thesis was written in during my internship at Adyen as the final project of the Master's program in Computer Science at the TU Delft. In my Bachelor's thesis, I compared the energy consumption of three Android UI frameworks, and I chose to continue working on the subject of energy consumption on Android as it is relevant both to Adyen and the greater Android community. In this thesis, I review the state-of-the-art of energy consumption tools for Android, compare three approaches to attributing energy consumption to code at a fine-grained level, and implement one of these in a tool that can be used to analyze release-mode Android apps. I then empirically evaluate this tool, and apply it at Adyen in a case study in which I cooperate with a developer to solve an energy bug. As part of the empirical evaluation, I also perform a preliminary comparison of release and debug mode with respect to energy consumption, using three code smells identified in prior work as having an effect on the energy use of methods containing them.

The results of the empirical investigation show that statistical sampling can be applied to Android devices to attribute energy consumption to methods within a reasonable margin of error. It further shows that this approach can be used to identify differences in energy consumption between different versions of software. The comparison between release and debug mode showed that the overhead caused by the use of debug mode is not consistent, and varies between both different code smells and different devices. This has significant implications for further work measuring energy consumption on Android, as it implies that results obtained using debug mode cannot be generalized to release mode, which all apps will use in production environments. Finally, the case study showed that this approach is able to significantly assist the energy debugging process, and revealed that even today, developers often lack the information necessary to make informed decisions over the energy consumption of their software. However, once this information is made available, both developers and stakeholders are willing to adapt their decision-making to incorporate energy efficiency.

Dr. Luís Cruz has been my daily supervisor at the TU Delft for my thesis, and has helped greatly with writing and revising of this thesis. He is joined on the thesis committee by Prof. Dr. Arie van Deursen, my thesis advisor, Dr. Rihan Hai, who is the third and final member of the core committee, and Dr. Michael Weinmann, the fourth member of my thesis committee.

I would also like to thank everyone from Adyen who helped me during this thesis: Chang-lun Wang, who volunteered to act as a company supervisor for my thesis, and has helped greatly with ideas and feedback throughout the whole process. Çağri Uslu, my team lead at Adyen, who mentored me throughout my internship and made sure I had everything I needed to make this project a reality. Andrei Petrescu, who I worked with to perform the case study, which formed a substantial part of this thesis. Lastly, the IPP Mobile team at Adyen, who I worked alongside during my internship, and who were always willing to help when needed.

Erik Blokland
Delft, June 2023

Contents

List of Figures

List of Tables

Acronym Definitions

AUT	application under test
JIT	just in time
JVM	java virtual machine
AOT	ahead of time
NDK	Native Development Kit
ADB	Android Debug Bridge
IS	internal setter
FOR	slow for loop
MIM	member ignoring method
ART	Android Runtime
POS	point of sale
MMRE	mean magnitude of relative error
HPC	high performance computing

1 Introduction

Traditionally, when investigating the efficiency of software, developers consider execution time as the most important factor. The performance of a given software is judged based on how quickly it performs its tasks, and optimizations are made in order to reduce run-time. There are many tools available for software developers to closely inspect the performance of their code, from high-level suggestions regarding operation ordering and data structure selection, to tools that give insight into how compiled instructions are run on the underlying hardware, enabling developers to tune their code for faster execution on a given platform.

However, another factor has become increasingly significant: energy efficiency. Energy efficiency has risen in importance for a number of reasons, one of the most prominent of which being the smartphone. 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 [33]. In order to provide a good experience for their users, developers have increasingly concerned themselves with the energy efficiency of their apps.

While developers have good intentions regarding improving their energy consumption, information on how to do this has historically lagged behind. In 2014, a study of *StackOverflow* questions found that, while there was significant developer interest in energy efficiency, there were few tools available to assist in profiling apps' energy use, and questions regarding energy consumption tooling often went unanswered [41]. In addition, questions over best practices related to battery life were often answered with 'folklore' that was not always correct. Many developers failed to understand the difference between energy and power, incorrectly assuming that reducing the instantaneous power draw would always reduce the energy consumed. Similarly, Grosskop and Visser found in 2013 that stakeholders of the software being developed often are not aware of the effect that software has on the energy consumption of a system [20], and were thus failing to take software energy efficiency into account when making design decisions.

As the mobile device ecosystem has grown and matured, there have been many improvements to developer tooling, documentation, and APIs to assist in writing energy efficient code, as well as academic research into the different ways that software can affect device energy consumption, and how to measure and predict these effects. Software can affect the energy consumption of a mobile device in a variety of ways, not all of which may be immediately obvious to an inexperienced developer. Code smells applicable outside of the field of energy efficiency have been shown to have an effect on energy consumption [38], which highlights the importance of adhering to best development practices. There are also a number of patterns specific to mobile devices that influence the energy consumption of apps [13], such as the use of WiFi vs cellular networks, or different UI colors on devices with an OLED display.

A number of different approaches to measuring the energy consumption of software have been defined, for both mobile and non-mobile settings. These approaches vary in both how they collect their data, and the goal of the approach. Many approaches use on-device sensors that report data through a common API, such as Intel's *RAPL*[1] or Android's *BatteryStats*[2]. Others use external hardware, which

[1]`https://01.org/blogs/2014/running-average-power-limit-%E2%80%93-rapl`
[2]`https://developer.android.com/topic/performance/power/setup-battery-historian`

often reports more precisely and at a higher rate than built-in sensors, at the cost of added expense and complexity, particularly when used with mobile devices. Some approaches rely on static code analysis rather than run-time measurements. These approaches use models, often utilizing machine learning, to estimate the energy consumption of arbitrary code after being trained on a set of apps. This has the benefit that, once trained, they can be run using only source code as input, without the need to run a predefined test on a real device. This greatly increases the speed that a developer can receive feedback on their changes, with the drawback that results are likely to be less accurate, won't show differences between devices (unless trained per-device), and may not be consistent between different apps.

There have also been many improvements and design changes to the Android API over time to help developers optimize the energy consumption of their code, such as the *Google Services Location API*[3], which allows developers to easily implement energy efficient location tracking and reduce reliance on energy-expensive GPS hardware. Google has also developed the *Android Studio* profiler[4], which is one of the most common methods for tracking power use on Android. This profiler can be run during app execution, and shows the developer a time series of approximated energy consumption split between the CPU and different hardware components. This allows developers to perform actions in their app and observe the relative power consumption of their test device. The tool also displays some other relevant information: wakelocks held by the app, accesses to the system's location API, and any Alarms/Jobs created by the app. While it provides substantial information to developers working on the energy efficiency of their app, it lacks granularity in measuring power use – developers must choose which features to run, but don't see exactly where energy is consumed, only a ternary value of low, medium, or high consumption at a given time. This is a significant barrier to developers looking to make energy efficiency a first-class requirement of their app, as a lack of information on "what, why and how to fix" provided by analysis tools is known to be a significant barrier in their adoption [25]. Another barrier is a tool's ease of use; The *Android Studio* profiler is designed for active developer use, and needs to be actively controlled by the user, making it less than ideal from this perspective.

With these barriers in mind, our goal was to measure energy consumption of Android apps at a fine granularity, without high overhead or complicated set-up processes. The end result of this thesis is EDATA, Energy Testing And Debugging for Android. EDATA uses statistical sampling, a low-overhead method of obtaining program execution information, in combination with on-device sensors to provide detailed, method-level energy consumption estimates. These estimates help guide energy debugging by showing developers which of their methods consumes the most energy, and by how much. They also help developers contextualize the energy impact of a given method or feature by providing an estimated cost in Joules over the duration of a test – letting them decide whether the energy saved is worth the trade-off in functionality or effort. This information is not only important to developers, but can help stakeholders understand the energy impact of their choices, the importance of which is also mentioned by Jagroep et al. and Grosskop and Visser [24, 20]. EDATA can be integrated into a continuous integration pipeline where tests are run on real hardware, and the results used to perform automated regression testing. By comparing the results of a consistent test suite between different releases of an app, the total energy of a given test can be evaluated, or split up at the method-level and evaluated method-by-method. Developers are then proactively alerted to changes in energy consumption, and determine if differences are significant enough to warrant further investigation.

In this thesis, we present EDATA, a low-overhead release-mode energy profiler for Android that estimates the energy consumption of apps at the method level. We also present our evaluation of EDATA, consisting of an empirical evaluation and a case study of the Adyen point of sale (POS) software. Our evaluation focuses on the following 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?

[3]`https://developer.android.com/guide/topics/location/battery`
[4]`https://developer.android.com/studio/profile/energy-profiler`

In our empirical evaluation, we test both energy debugging and energy testing scenarios. To evaluate energy testing, RQ1, we used three code smells found by Palomba et al. [38] to increase energy consumption on Android, and two workloads that activate hardware components of the device. We found that EDATA was able to reliably distinguish between the energy consumption of versions of a workload, which shows that our approach is capable of performing energy regression testing. Thus, we answer **yes** to RQ1.

We evaluated the energy debugging scenario, RQ2, by creating our own baseline using execution time as a proxy for energy consumption, through randomly selecting a class out of a set of six classes to run at fixed timesteps. We found that our approach using callstack sampling, based on that of Mukhanov et al. [34], was able to accurately order the six methods used, with the mean magnitude of relative error (MMRE) remaining below the 25% considered an upper limit [7]. We therefore answer **yes** to RQ2.

During our empirical evaluation, we found that testing apps in debug mode, as in many prior studies, can lead to inconsistent energy consumption overhead, reducing the applicability of results to real-world usage scenarios, and showing that certain code patterns may differ drastically between the two build modes. We consider these findings to have significant implications for the future of Android energy efficiency research – researchers should take care to note which build mode has been used in their evaluation, and try to use release mode if possible. Developers profiling their apps should also take care to test on representative hardware, as some devices, particularly older or lower-spec devices, may show different characteristics than others.

Finally, we performed a case study on Adyen's POS software to answer RQ3 and investigate whether the results of our empirical evaluation transferred to real-world software. In this case study, we found the root cause of an energy bug present in the POS software on Adyen's AMS1[5], using EDATA in combination with the Android Studio profiler. We worked with a developer assigned to this issue, and used the method-list output of EDATA to identify 'hotspots' in the code. We further used this output to determine that, given the total energy consumption attributed to the POS software, the root cause must not be related to code execution, and ultimately identified a wakelock as the source of the bug. We finally used EDATA to quantify the energy savings from removing this wakelock. We found that, given our observations and the feedback from the developer we worked with, we are able to answer **yes** to RQ3.

During the case study, we were able to see how developers approach energy debugging, and what capabilities are missing from existing tools. We also observed how the data provided by EDATA influenced the decision-making process, and how both developers and stakeholders were willing to take energy consumption into account when making architectural decisions. While significant research has been done in measuring and reporting energy consumption of software on mobile devices, developers in the real world often do not have sufficient information with which to make informed decisions on how their code affects the energy consumption of the devices it runs on. Future work should continue to build on EDATA, and provide developers and stakeholders with detailed, meaningful information that can be used in the decision-making process to make energy consumption a first-class metric.

[5]`https://www.adyen.com/pos-payments/terminals/ams1`

2 Prior Work

There is a significant body of research into the energy consumption of software, on both mobile and desktop/server platforms. While many concepts and findings can be shared between platforms, it is important to keep the fundamental differences between them in mind. For example, desktop and particularly server platforms typically use wired network devices, which are known to consume very little energy [35], whereas mobile devices expend significant energy to perform network requests and cannot approach networking optimizations in the same way. There are also fundamental differences in workload: Apps designed for mobile devices typically perform work in short bursts, and often follow the "race-to-idle" pattern [13]. Research into energy consumption on non-mobile platforms often focuses on high performance computing (HPC), where workloads are typically consistent over a long period of time, and the CPU speed may be fixed. When interpreting prior work, the research context must be taken into account to ensure that the conclusions drawn from one scenario transfer to the task at hand.

Energy consumption measurements often go hand-in-hand with energy testing - the identification of energy bugs. Once the energy consumed by a software application is known, be it at the application, method, or even line-level, different versions of that software can be tested to see whether energy bugs, causing an increase in energy consumption, have been introduced. Finer grained measurements can make identification of energy bugs easier, but this often comes at the expense of complexity and overhead, as most approaches to method-level or finer use instrumentation to collect information on which methods are called, which is an expensive operation.

Finally, we consider the application of existing software engineering concepts and techniques to the problem of energy efficiency. Concepts such as code smells are widely used to guide software development and improve quality, and some work has been done to study both the relevance of existing code smells in the context of energy efficiency, and to create new smells specific to the field. Work has also been done to determine the relationship between overall code quality and energy consumption, and to create a picture of how developers interact with energy in their development process, in order to guide researchers in creating tools to most effectively help developers.

2.1. Background

2.1.1. Measuring Energy Use

In this section, we consider works that focus primarily on directly measuring the energy use of software running on some specified hardware. These works measure at varying levels of precision, from a general overview of the energy consumed during a test case [23] to line-level measurements [28]. Measurements are taken in a variety of ways, including the use of external hardware-based collection [23] and leveraging internal power monitoring hardware [17, 34], and often combine these methods with offline analysis of the collected data.

2.1.1.1. Android Devices

Due to their widespread use and open-source nature, Android devices are often targeted by research into energy consumption of mobile apps.

Di Nucci, et al. created PETrA, a toolkit for Android that leverages tools built into Android devices since Android version 5 [17]. This tool provides accurate method-level energy estimations for arbitrary Android apps without the need for any hardware modifications or a 'rooted'[1] device. The authors generate energy usage estimations by running automated test suites for a given app, while collecting data using Android's built-in app tracing tooling, Batterystats, and Systrace. Combined, these data sources are able to provide information on the energy state of the device at a given time, along with the current methods being executed by an app. Using the built-in power profile of the device, estimated current draw is calculated for each energy state, and the power consumption of each method call is calculated based on the method entry and exit timestamps generated by the tracing tools. The resulting energy use calculations are shown to be accurate to previous work using a hardware power meter [29], with 95% of methods having an estimation error within 5% compared to the established baseline.

Li et al. developed a technique to find the energy consumption of Android apps running on the Dalvik virtual machine at the source-line level [28] Their approach consists of two phases: the "Runtime Measurement Phase" and the "Offline Analysis Phase". During the runtime measurement phase, their *App Instrumenter* instruments the app under test with probes that record precise timestamped path and API call information. Power samples are simultaneously collected by a hardware *Power Measurement Platform*. In order to collect path information efficiently, an extended version of the method used by Ball and Larus [4] is used. In the offline analysis phase, the collected data is analyzed in order to associate energy consumption with individual source lines, as well as identify 'tail energy' from API invocations that activate hardware components, leading to heightened energy consumption lasting longer than their runtime. Before the main stage of analysis, the data is first preprocessed - API invocations have their energy consumption manually calculated by summing measurements taken during their runtime, taking into account the potential presence of additional threads. Tail energy is attributed to one or more invocations depending on their timing and the information contained in the device's energy model. After preprocessing, Robust Linear Regression is used to solve $E = mX$, where E represents the adjusted power measurements and X represents the recorded path information, with each row in X representing a frequency vector of bytecodes in the path. They also include inserted instrumentation bytecode in their analysis, and remove it before displaying the result to the user. If there are insufficient samples available to solve the system of equations, the tool will notify the user that more samples must be taken, or optionally group similar bytecode entries together in order to make the system solvable with the current set of samples. The authors found that their approach was able to calculate energy information for each tested app within three minutes. They were able to attribute invocation energy consumption to within 10% of the ground truth, and the produced models fit the data with R^2 of 0.93. Finally, they detected high energy events, such as garbage collection and thread switching, with 100% accuracy.

Cornet and Gopalan extended the *Orka* [47] tool to provide source-line information to help developers understand where in their code API invocations were consuming energy [11]. They also extended the hardware usage accounting capabilities of the tool, providing detailed accounting of both active and tail energy consumed by the WiFi module of a device. They use the built-in power profiles included with Android devices in order to estimate this energy consumption based on the state of the hardware, which was determined using the statistics found in `/proc/net/xt_qtaguid/stats`, which was polled through adb every 45ms. They found that their approach was able to correctly identify methods that activating the WiFi antenna, but needed more complex accounting to handle tail energy that was consumed outside of the calling method. *Orka* also depends solely on the energy consumption of Android API invocations, as it is asserted that the bulk of energy spent in most Android apps is used by the Android API [47], meaning that energy consumed by non-API code will be ignored. However, this is likely not a concern for tracking WiFi energy consumption, as network traffic will likely always be sent through the Android API.

Farooq et al. developed MELTA [18], a tool for Android that combines the method tracing methodology of PETrA [17] with the *Constant Power Model* and *Variation Power Model* power models from ALEA [34] to deliver method-level energy consumption estimates using data collected *dmtracedump*[2], *BatteryHistorian*[3], and method-tracing instrumentation. They evaluate their approach by defining test

[1]'Rooting' an Android device refers to allowing user access to the 'root' superuser account, which is not subject to the same access and security restrictions as a regular user. Consumer Android devices do not allow access to the superuser account by default, and rooting the device typically requires either installing a third-party build of Android, patching the provided Android version to enable root access, or exploiting security vulnerabilities.

[2]`https://developer.android.com/tools/dmtracedump`

[3]`https://github.com/google/battery-historian`

cases for fifteen open-source Android applications, and run each test case ten times. Some of the methods considered by their test suite make use of network and location services, in addition to performing work on the CPU. They use the Qualcomm *Trepn* profiler to generate a ground truth for each of their test cases. They found that the average mean magnitude of relative error (MMRE) of MELTA's results compared to *Trepn* was lower than 0.15, implying that their results are accurate enough to be used for energy profiling.

2.1.1.2. Desktop and Server devices

In their 1999 paper, Flinn and Satyanarayanan developed *PowerScope*, a tool using statistical sampling to attribute the energy consumption of a laptop running NetBSD to functions within software, and likely the first tool developed to do so. Their approach works only with a single-core CPU, as laptop computers of the time generally did not have multiple CPUs available, making the added complexity of an algorithm to handle multiple CPUs unnecessary.

Their approach consists of three components: The *System Monitor*, which ran on the computer being profiled, and sampled the PC and PID of the process running on the CPU. The *Energy Monitor*, which ran on a separate data collection computer, and received current/voltage samples from a hardware multimeder connected to the power supply of the device being profiled. Lastly, the *Energy Analyzer*, which performed an offline analysis on the collected data. For their power sampling, they observed that the voltage provided was consistent enough such that the differences were negligible in practice, so their implementation assumed a constant voltage for the duration of each profile. To synchronize their two sets of samples, they connected the multimeter's external trigger input/output to pins on the laptop's parallel port, allowing the multimeter to trigger the *System Monitor* to collect a sample, and the *System Monitor* in turn to notify the multimeter when the sampling process had been completed. In order to control *PowerScope*, they modified the NetBSD kernel to collect additional information on the path of executing processes, when shared libraries are loaded, and to provide a set of system calls with which to start/stop profiling and read profiled data from a buffer. Once data has been recorded, they first attribute each sample to a "bucket", which is either related to the PID of the currently executing process, the kernel, or a general "interrupt handler" bucket. They then perform a similar analysis to attribute energy consumption to functions, by first reconstructing function memory address information using stored symbol tables and the information recorded by the *System Monitor*, and then comparing this to the PC recorded in each sample.

Their evaluation focused on a case study in which they optimize the energy consumption of a video player. They begin by measuring the effects of video compression and display size (in pixels), and find that while both higher compression and a smaller display size reduce energy consumption, the bulk of the remaining energy is used by idling hardware. They then finish their evaluation by modifying the network driver and their video player software to power down networking components and the hard disk, respectively, and find that this also significantly reduces energy consumption.

Pereira et al. developed SPELL, a tool for Java applications that combines energy consumption measurements collected from test cases with spectrum-based fault localization to rank methods based on their energy consumption [40]. In contrast to many prior works, SPELL does not attempt to show developers the precise amount of energy used by their code, but provides a ranked list of 'source code components', such as methods, based on the likelihood of those components being responsible for unusual or excessive energy consumption. SPELL uses spectrum-based fault localization, or SFL, a technique that works by tracking which program components are executed for each test case, and whether an error was detected during that test (an error vector). This information is then used to determine the probability of a component being faulty. SPELL modifies this technique by collecting, for each time a component is present within a test case, the energy consumption in joules of the various hardware components during the time that the component was executing, the number of times it was executed, and the amount of time the component was actively executing. Unlike standard SFL, there is no prior knowledge available to construct an error vector, the SPELL tool must create its own. The authors have defined an algorithm for the construction of this error vector based on prior work to assign responsibility for greenhouse gas emissions between countries. This algorithm sums the values of each component per test case, and uses these sums as the error vector. To determine which components in a test case are responsible for its energy use, the Jaccard similarity coefficient is calculated. A global 'error vector' is also calculated to allow comparison of components beyond a single test case, by taking into account the energy used by different hardware modules of the test system, the frequency of use, and the runtime of the component. This ‘global similarity’ score is intended to provide developers with more accurate guidance on where optimization efforts should be spent, without relying solely on the energy use of particular test cases.

To determine the effectiveness of their solution, the authors performed an empirical evaluation. In their experiment, 15 experienced Java programmers were asked to optimize the energy consumption of a project written by a student for an object-oriented programming course. These programmers were divided into five groups of three, with each group being assigned a single project. Notably, SPELL was used as part of the selection process to determine which projects would be included in the evaluation, with each project being considered as a “component”, and those with the highest global similarity, corresponding to the highest likelihood of an energy leak, being selected for analysis. Each participant was then given 30 minutes to analyze the project, and randomly assigned either the SPELL tool, the NetBeans profiler, or no tools. They were then given up to two hours to modify the project, and specify any further changes that they would make.

The authors found that SPELL enabled programmers to reduce the energy consumption of their assigned projects more effectively than using either no tool or the NetBeans profiler. They found that while SPELL did not greatly change the effectiveness of modifications, the modifications made more accurately targeted the most energy consuming components. Participants with access to SPELL also took less time on average, and were more confident in their improvements. However, in calculating the energy improvements for each project, the same test cases are used that were used both to select projects for analysis and to generate SPELL rankings. The authors do not specify if extra test cases were used to determine whether the results found hold for test cases outside of the input dataset.

Mukhanov et al. developed ALEA, a tool used to measure the energy consumption of software at a basic-block level [34]. A basic block is a ‘block’ of code with no jump instructions - that is, it will always execute sequentially from entry to exit. This property means that the energy consumption of a basic block will be relatively consistent when compared to a method, as the same operations will always be performed. ALEA uses a probabilistic sampling model, where the program under test is systematically sampled during execution. Each sample collects the value of the program counter and the 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. Multithreaded 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. On the Sandy Bridge platform, the authors found that ALEA was able to estimate execution time and energy with an average error of 1.1% and 1.4%, respectively, though error was higher when only considering parallel benchmarks, with 3.1% and 2.6% respectively. On the Exynos-based platform, higher errors were observed due to a difference in how the hardware handled sampling, but errors remained below 4% in all cases. 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.

Noureddine et al. developed *E-SURGEON* [35], a power monitoring tool composed of two distinct subsystems: *PowerAPI*, a modular system that collects platform-dependent resource utilization information, and *Jalen*, which profiles running (Java) applications and estimates their energy consumption. The authors define these as both a generic architecture that can be implemented in different ways depending on the needs of the platform. Along with an architecture design, the authors defined a set of power models for both *Jalen* and *PowerAPI*, which define how the system attributes consumed energy between different hardware components, and to units of source code.

In addition to defining generic architectures, they have also concretely implemented both *PowerAPI* and *Jalen*, with *PowerAPI* modules being implemented for the CPU and NIC of a Linux system, and *Jalen* implemented for Java applications using bytecode instrumentation and delegator classes to log use of sockets.

They found that their tool was able to estimate energy consumption with a margin of error between 0.5-3%, depending on the test load. Using *Jetty* as a case study, they found an overhead of between

43-57%, depending on the workload being executed. While this overhead is considerable, it is less than that of the *Java Interactive Profiler*, and they consider it to be reasonable.

Finally, they investigate the energy consumption of a *Jetty* benchmark, and find that only 10 methods consume about half of the total power used during the benchmark, with most energy consumed by classes reading and writing HTTP requests. They conclude that their results are reasonable, taking into account the contents of the benchmark, and that the information provided by *E-Surgeon* would be effective in helping developers optimize the energy consumption of their software.

2.1.2. Estimating Energy Use

In addition to using direct measurements, efforts have been made to estimate energy consumption based purely on static analysis of source code. There are multiple benefits to the use of static analysis instead of runtime measurements. One such benefit is that estimations can be made independently of any test cases or test devices. This reduces the amount of effort needed to integrate an energy testing solution into a project, as developers would not need to manually create test cases in order to generate energy consumption data, as with many measurement-based solutions. Developers could also easily integrate such a solution into a continuous integration pipeline, as no physical test device would be required. Another benefit is the speed and ease of testing - changes made to software can be analyzed on the spot, without the need to compile and deploy code to a test device, and without the need to maintain an energy testing environment.

2.1.2.1. Machine Learning Approaches

Alvi et al. defined a machine-learning-based approach to estimating application energy consumption on the method level [3] Their approach uses source code metrics as features in a machine learning model, and energy consumption of individual methods as training data. They first collect the energy consumption of different methods using the Qualcomm Trepn profiler[4] to record real-time energy consumption, and `dmtracedump` to obtain method traces. Test cases are manually defined by the authors, and recorded using Espresso.[5] As Trepn updates every 100ms, they cannot directly measure the energy consumption of a method with a shorter execution time. To overcome this limitation, the sliding window approach is used. After collecting energy consumption information, the authors then collect method-level metrics, using the Metrics Reloaded plugin [6]. The authors found that some of their selected metrics had a high level of correlation with method-level energy consumption, though many of these metrics were either directly or indirectly related to the overall number of instructions in the method, so it is unclear whether these different metrics contribute to the overall strength of the model. They also found that their model was able to accurately predict the energy consumption of arbitrary methods, with about 94% precision, recall, f1, and accuracy scores. Notably, they also found that different devices did not differ significantly in energy consumption per-method.

Chowdhury, et al. proposed a software energy model, *GreenScaler*, built on random tests using CPU utilization as a heuristic for selecting tests, in place of the more commonly used code coverage heuristic. The authors used *GreenOracle* [9] as a baseline for their model, aiming to alleviate one of the key difficulties identified in this work: the need to write an extensive test suite. By using automatically generated tests, the training dataset for their model can be vastly increased with minimal manual work. The authors used the 'Leave-one-out' method to train 472 different models, using features such as CPU time, page faults, and UI colors as input, as well as system calls as a proxy for energy consumption from other components. As these models were found to be nearly identical, one was selected at random to be evaluated. Using this test generation method, they were able to significantly improve the performance of *GreenScaler* compared to *GreenOracle*, with the most significant improvements found on the randomly generated test suite, but improvements also noticeable in manually written tests used in the original *GreenOracle* evaluation. They also found that while accuracy varied among different apps, the error distribution remained closely similar between versions of a single app, allowing the tool to be used to find differences, such as energy efficiency regressions, between versions of an app. Finally, the authors conclude that increasing the number of apps used in their training set has a dramatic effect on the accuracy of the model, with the most notable gains occurring until about 300 apps are added to the

[4]`https://web.archive.org/web/20150716000746/https://developer.qualcomm.com/software/trepn-power-profiler`

[5]`https://developer.android.com/training/testing/espresso`

[6]`https://plugins.jetbrains.com/plugin/93`

dataset. They also conclude that the use of CPU-utilization as a test selection heuristic is sufficient, with the alternative energy model based heuristic offering little benefit.

In his 2020 Master's thesis, Stephen Romansky trains a selection of machine learning models to predict the 'time series' of energy consumption of Android apps [42]. The time series of energy consumption is a set of energy consumption figures over time (essentially, the data that would be collected when periodically sampling the power use of a device), and can be associated with software features, such as system calls, that occurred at a given time. Applications were selected from the *GreenOracle* dataset, and were open source with at least 30 unique revisions. The features used to train the model were all system calls made by the software, as well as the device resource utilization, collected using *strace* and procfs files located in `/proc/`. Each of these features was collected as a time series, so that they can be associated with the time series energy consumption measurements for training. After evaluation, the author found that timeseries models peformed slightly better at predicting total energy consumption than models built for the sole purpose of predicting total energy consumption. He also found that stateful models performed better at predicting time series of energy consumption than stateless models, and that the shape of the predictions better fit the shape of observed energy consumption. MLP models were found to perform worse than linear models. Finally, using a combination of CPU and procfs features performed better than using one or the other. In general, when given access to real hardware, a model that only estimates a time series of energy consumption measurements has limited use, as the features required as input require running a test case - in which case most hardware will be able to provide real energy measurements. However, one potential use case could be for estimating energy consumption when using an emulator to run test cases, reducing the need for testing on real hardware, and allowing energy tests to be run solely from a host machine.

2.1.3. Identifying Energy Bugs

2.1.3.1. Identifying bugs between software versions

Jagroep et al. defined a methodology to compare different revisions of the same software product, and developed a tool to attribute energy use to its different components [24]. They tested their approach empirically by performing a case study on two revisions of the *Document Generator* software, before and after an encryption function was added for GDPR compliance. This software consists of several components, running in separate processes, across two different servers - an 'application' server and a 'database' server. They compare *JouleMeter*, a tool published by Microsoft, to their own model, built using penalized linear regression. They found that their model outperformed *JouleMeter* at machine level prediction, with MAPE scores of 0.004 and 0.005, respectively, when compared with a hardware power meter. They also found that process level energy consumption prediction was overestimated, and conclude this is likely caused by external factors that are not detected as separate processes by their tool. Nevertheless, they are able to collect consistent and comparable data using their defined methodology, and identify both an overall increase in energy consumption between the two versions, as well as an unexpected increase in energy consumption in one of the components that was not affected by the addition of encryption.

2.1.4. Data Collection / Mining

Cruz and Abreu compiled a list of 22 design patterns used by mobile app developers when considering energy efficiency [13]. To create this list, they first defined a dataset of open source apps on both iOS and Android. They then automatically selected commits with messages containing energy-related terms, and further refined their selection by manually filtering out false positive matches. After defining their dataset, they use a defined process in order to categorize each commit depending on the code changes it contains, with the resulting energy patterns including UI changes (e.g. Dark UI colors used on OLED displays), hardware utilization changes (e.g. preferring to wait for WiFi instead of using mobile data), allowing users to take control of energy expensive activities, and others. In addition to compiling a list of patterns, they also investigate whether energy efficiency is addressed differently by iOS and Android developers. They found that there is a difference in the way developers implement energy patterns depending on platform, and suggest that it may be related to both hardware features (such as the prevalence of OLED displays on Android devices) and differences in developer documentation.

Hindle et al. created a framework to automatically run tests on different versions of Android apps, and collect energy data through hardware instrumentation [23]. Their tool does not attempt to associate energy consumption with any specific source code feature, but instead provides energy consumption measurements during the runtime of a predefined test, with the goal of accurately and repeatably measuring the energy consumed by an app, with sufficient precision to compare measurements between different versions. In addition, the authors normalize measurements taken from different phones of the same model to one 'gold standard' phone, in order to account for differences in energy consumption caused by manufacturing tolerances. The authors test their framework using Firefox Fennec as a case study, loading a sample website using 685 different versions of the app. The versions tested were selected based on GUI changes - any change that might affect the UI being displayed, and thus the power consumption of an OLED display, was selected.

2.1.5. Code Smells

Code smells are a widely studied area of software engineering, with a broad range of applications. While there is far too much literature on the topic of code smells to list here, we are particularly interested in those applicable to Android, and their effect on the energy consumption of the app they are found in. Habchi et al. [21] studied the life cycle of Android code-smells in real-world open-source apps, and found that while some code smells can remain in a code base for years at a time, 75% are removed within 34 commits. They also found that larger projects tend to remove smells more quickly, and that smells detected and prioritized by Android Lint were removed more quickly than those that weren't, indicating the importance of automated detection.

2.1.5.1. Effect of code smells on energy consumption

Palomba et al. used their prior works, *PETrA* [17] and *ADOCTOR* [37], to investigate the energy consumption impact of 9 different method-level code smells on Android applications [38]. In order to associate these code smells with energy consumption, they first analyzed the apps involved in the study with *ADOCTOR* to identify instances of code smells, and analyze their distribution and co-occurrence. They found that four code smells were most common in their dataset, and that 62% of methods affected by a code smell were affected by more than one. They also found that high diffuseness, or the number of times a smell was detected, did not necessarily imply high co-occurrence, as the common *Transmission without Compression* smell generally did not co-occur with other smells. After determining which methods contained code smells, they then used the *PETrA* tool to estimate the energy consumed by a single call to each method in the app under test. Using this information, they were able to determine that 94% of the most consuming methods in their dataset were affected by at least one code smell. They also found that by refactoring four code smells, *Internal Setter, Leaking Thread, Member Ignoring Method*, and *Slow Loop*, they were able to significantly reduce the energy consumption of the associated methods, with the largest difference being 87 times.

Cruz and Abreu investigated the energy consumption impact of eight code smells from Google's list of best development practices for Android [15], to answer whether programming best practices for improving app performance can be blindly applied to improve energy consumption. They considered eight code smells found in the *lint* tool of the Android SDK, including both Java-based and XML-based smells. These smells are: *DrawAllocation*, where object allocations are made within UI drawing code. *WakeLock*, where wakelocks are used unnecessarily or held for too long. *Recycle*, where collections using singleton resources are not correctly de-allocated (missing `recycle()` call). *ObsoleteLayoutParam*, where UI views contain unused parameters. *ViewHolder*, a technique where data from previously drawn items is used to improve the efficiency of list views, reducing the number of calls to `findViewById`. *Overdraw*, where a single pixel on the display is drawn to multiple times. *Unused Resources*, where resources compiled into the APK are not used. *Useless Parent*, where a parent layout in UI is not used. To evaluate the effect of their chosen code smells, they performed an empirical evaluation in which they randomly selected six open-source apps, filtering out those which made heavy use of network operations or did not contain any of the selected code smells (as determined by *lint*). For each of the chosen apps, they wrote a selection of UI tests using a *Python* library that allowed programmatic manipulation of UI elements, allowing tests to be used across different devices. During each test, they collect energy consumption data through on-board power sensors, and integrate the sampled energy consumption over time to obtain the consumed energy. They found that, of their eight code smells, six of them had an effect on energy consumption when fixed, with one of the six increasing energy consumption rather than decreasing it. The unexpected increase in energy consumption was caused by the removal of the 'overdraw' code smell, where automated fixes typically require run-time checks. These checks sometimes cost more to execute than drawing the unnecessary pixels, leading to an increase in energy consumption. They found that, with exception of the 'overdraw' pattern, it was safe to apply automated fixes for each of their chosen code smells, as the rest showed either a decrease in energy consumption or no effect. From this result, they concluded, in this scenario, performance best practices generally have a positive effect on energy consumption, but further work is necessary to explore different scenarios, as results may differ. Finally, they concluded that, as they were able to measurably improve the energy consumption of three of their six tested apps, the best practices tested can improve the energy efficiency of real-world apps.

2.1.6. Energy Efficiency in Software Engineering

Moura et al. mined 2,189 commits from *GitHub*, and identified 371 "energy-aware" commits [32]. They selected commits from 'non-trivial' applications in order to obtain an accurate representation of how experienced developers interact with the energy efficiency of their code in the real world. They found that the majority of commits in their collection, 49.66%, targeted low-level code, such as kernel and driver code. These changes generally used hardware-based approaches, such as more effectively using sleep states or frequency scaling. Only about 16% of commits used techniques such as switching to more efficient data structures or libraries, and the authors suspect that there is significant room for improvement in this area.

They also observe that developers are not always certain that their changes save energy, even in commits made solely for the purpose of energy efficiency, and use "hesitating" language implying that they have not been able to definitively test their changes. In some cases, developers even reverted their energy saving commits. No definitive answer was given for these reversions, but many of the reverted commits had implemented the same feature, a work queue that traded performance for energy savings. The authors suggest that decisions regarding energy efficiency and performance must be made with care, and that the trade-off between them is often unclear. Finally, they found that energy bugs are commonplace in real-world software, with 11% of their studied commits fixing an energy bug.

Cruz et al. investigated the effect of commits intended to improve the energy consumption of Android applications on the maintainability of those applications [16]. To measure maintainability, they used *BetterCodeHub*, a tool by the Software Improvement Group[7], in combination with their own formula designed to make the output of *BetterCodeHub* comparable between different projects. Their formula ensures that the size of a project does not affect its maintainability, and weights the number of code lines violating guidelines by the severity level of the violation.

The authors evaluated a total of 539 commits over 306 separate apps. They split these commits into a number of categories, depending on the type of change being implemented. In all categories, they found that the majority (over 50%) of commits had a negative effect on maintainability, compared to a baseline of 33%. There was, however, an improvement in maintainability observed in about 30% of commits, with the rest having no change. The authors conclude that the bulk of maintainability issues are attributable to a lack of attention to maintainability by developers and 'insufficient support of energy-efficiency patterns by mobile platforms'. They find that mobile frameworks must improve support for "out of the box" energy-efficient programming, and provide more information to developers on best practices. Programming languages should also take energy patterns into account when designing features, and ensure that developers can easily implement these patterns without harming maintainability.

2.1.6.1. Refactoring

Ournani et al. investigated the opposite question as Cruz et al.: What is the impact of refactoring on energy consumption? [36] They focus exclusively on Java projects, and select well-established projects that have been hosted on GitHub since at least 2015, seven years prior to the publishing of their work. To find commits with significant refactorings, they use the *RefactoringMiner* tool to identify and summarize code refactoring commits for each project. They then manually select commits to be included in their analysis, and compile a version of the JAR for each commit to be tested. As a projects test cases may not be stable between commits, they define their own test suite using JMH[8] for each project to ensure that results are comparable between commits, and use Intel's RAPL to measure the power consumed by their test suite.

The authors found that, in general, structural refactorings do not significantly change the energy consumption of code. This implies that these refactorings are safe to perform, and increases in energy

[7]`https://www.softwareimprovementgroup.com/`
[8]`https://github.com/openjdk/jmh`

consumption are generally insignificant when compared with the total energy consumed by the program. They also found that the energy consumption of most studied projects steadily decreased over time. In contrast to structural changes, refactorings that change the work performed by code has a significant impact on energy consumption, both increasing and decreasing. This type of change should be made in combination with energy testing so that developers can make an informed choice on whether the tradeoff between their changes and the energy consumption of their program is reasonable.

Continuing their 2017 work [15], Cruz and Abreu developed a refactoring tool, *Leafactor* to automatically fix five ‘energy smells’ they had previously identified as impacting the energy consumption of Android apps [14]. Of these five smells, four are Java-based, and one is XML-based. They analyzed 140 apps from the open-source app store *F-Droid*, prioritizing recently released apps, and filtering to those written in Java. Of those 140, they found that 45, or 32%, contained at least one of their five energy smells, and generated 222 total refactorings. The XML-based energy smell, ‘ObsoleteLayoutParam’, where unused parameters are found in a UI view, occured 156 times in 30 projects, an average of about 5 times per project. The ‘Recycle’ energy smell, in which a collection using singleton resources is not properly disposed of, occurred in 58 times over 23 projects. The other three smells, ‘DrawAllocation’, ‘ViewHolder’ and ‘Wakelock’, showed only marginal improvements, with no fixable occurences of ‘DrawAllocation’ being detected. From their 222 refactorings, they created 59 pull requests across 45 apps, splitting different smells into different pull requests where necessary. Of these, 18 apps merged their pull requests before writing. They found that, in most cases, developers are open to suggestions from automatic refactoring tools, and were often unaware of the impact that their code can have on energy consumption. However, they note that a potential downside of their automated refactorings: in many cases, the refactored code is more verbose and harder to understand than the less efficient code, which can hinder the adoption of these automated refactorings by developers.

Morales et al. implemented *EARMO* [31], a tool that refactors anti-patterns common to mobile apps while taking energy consumption into account. Their work is split into two parts: The first of these is a preliminary study that considers a set of software anti-patterns split between generic OOP patterns and Android specific patterns. In this study, they investigate whether anti-patterns influence energy consumption and if different types of anti-pattern influence energy consumption differently. They found that, out of 24 manually corrected anti-patterns, 7 produced a statistically significant decrease in energy consumption, with the rest producing non-significant changes. Two patterns in particular actually reduced energy consumption, though not significantly: Long parameter list, and Speculative Generality.

With the results of their preliminary study, the authors developed EARMO, or Energy Aware Refactoring approach for MObile apps. EARMO uses an estimate of energy consumption as a separate objective in a multi-objective search, intended to minimize energy consumption as well as different structural metrics relating to the quality of the software. EARMO provides multiple options to developers, so that they may choose which of the objectives they prefer, or a “middle ground”. THey found that, based on their metrics, EARMO is able to improve the design quality of mobile apps by both correcting anti-patterns and improving extensibility and effictiveness. In order to test their refactorings in the real world, they submitted pull requests to several apps containing their refactorings. The acceptance rate of their pull requests varied by app and refactoring type, but most were accepted, and those that were rejected, outside of one app, were generally considered to be correct, but not desired for out of scope reasons.

2.2. Related Work

In this section, we summarize works from Section 2.1 that are closely related to EDATA, and highlight the differences between them.

ALEA, developed by Mukhanov et al. [34], was our primary inspiration for the methodology used in EDATA. ALEA uses statistical sampling to estimate the energy consumption of C/C++ programs at the basic block level, and was tested on both x86_64 and ARM to have average errors of around 4% or less in both single and multi-threaded benchmarks. ALEA focuses on analyzing CPU energy consumption on Linux, and uses benchmarks typical to HPC where a steady CPU load is maintained for the duration of the benchmark. In contrast, EDATA has been developed for use with Android devices, and is intended for use with Android apps, which generally have a variable CPU load depending on user interaction. Additionally, Android devices typically use wireless network connections, either WiFi or cellular, and may draw data from a variety of different sensors, meaning that whole device energy consumption – not just the CPU – needs to be taken into account.

DiNucci et al. developed *PETrA*, a tool for Android capable of estimating the energy cost of individual methods in an app [17]. *PETrA* uses method instrumentation to collect a trace of each method that is executed by an app, with precise timestamps. This data is combined with hardware state information collected by *Batterystats*, and power profile information from the included `power\_profile.xml` to generate method-level energy estimates. They then compared their estimates to a ground truth identified by prior work [29], and found that 95% of methods had an estimation error within 5%. PETrA is notable in that, in contrast to most works, energy consumption of the device is not directly measured, instead being calculated based on the `power\_profile.xml` file, which contains values measured and reported by the device manufacturer, and must be included in all Android devices. This methodology has several benefits: First, as power profiles contain power use information on individual components, it is possible to separate the energy used by each component of the device, although they do not implement this. Second, this methodology is independent of device sensors, and thus is not affected by differences in measurement quality when switching between different devices. However, there are some important drawbacks to this approach: Method instrumentation can cause significant overhead, which will likely affect the estimates generated. In addition, as the power values reported in the power profile are estimates, software that causes above average energy consumption in a particular component will not be detected. EDATA performs a similar function to *PETrA*, but with several key differences. In place of the power profile, we directly measure current using on-device sensors. This allows EDATA to determine the energy consumption of the full device without the need to track the state of the hardware components. To collect information on which methods are executed, EDATA uses statistical sampling instead of method instrumentation. This both lowers the overhead of the data collection, and allows native code to be analyzed in addition to Java and Kotlin code, as native code cannot be instrumented automatically[9].

Palomba et al. used *PETrA* to evaluate the energy consumption of a number of code smells, performing an empirical evaluation on 60 Android apps[38]. They select nine code smells from a catalog of 30 Android code smells, filtering those that did not appear to be directly related to energy consumption as well as those that were not related to the source code (such as the XML-based code smells mentioned by Cruz and Abreu [15]). The goal of their study was twofold: they first investigated the extent to which the nine code smells were 'diffused' within the source code of the 60 apps, and to examine the effect of the code smells on the energy consumption of their containing methods. They found that some code smells more commonly co-occurred in methods than other smells, and that some code smells could significantly increase the energy consumption of a method. They found that refactoring methods affected by the *Internal Setter*, *Leaking Thread*, *Member Ignoring Method*, and *Slow For Loop* code smells significantly decreased energy consumption. In this thesis, we make use of three of the code smells investigated by Palomba et al. [38]: *Internal Setter*, *Member Ignoring Method*, and *Slow For Loop*. We chose these specific smells as they are fully contained in source code, making them easy to measure with EDATA, and due to the significant differences in energy consumption that were shown. Our empirical evaluation differs from that of Palomba et al. [38] in that we have designed specific test cases for each of these code smells, instead of finding real-world apps containing them. We also do not attempt to measure the energy consumption of one execution of a method, instead estimating the total energy consumption of the method during a fixed workload. Finally, due to the time and numerous improvements to the Android Runtime (ART) between their work and this thesis, we cannot be certain that the effects of code smells observed in their work are still applicable.

Jagroep et al. defined a methodology to compare revisions of a software product, and used their methodology to compare two revisions of *Document Generator*, a commercial product consisting of multiple distinct components [24]. They implemented their methodology into a tool capable of attributing energy consumption at the process level, a level which provides relatively little detailed information on the location of potential energy bugs. However, in this use case, their choice makes sense: *Document Generator* is split across multiple processes on multiple servers, so unlike some software (such as Android apps), there is still some information available on the location of a bug, and not all of the components had changed between their compared releases. Their work also shows the importance of providing energy consumption information to stakeholders during the development process – some components of the tested software showed an increase in energy consumption, but new (mandatory)

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

features had been implemented that made this increase in energy consumption justifiable. We largely adopt the methodology described by Jagroep et al. [24] in our empirical evaluation and case study, and we also consider the importance of providing energy consumption information to stakeholders in our case study. EDATA differs from their approach by providing more specific attribution of energy consumption to source code, which helps developers determine whether or not an increase in energy consumption is reasonable for a change, and assists in the debugging process should an increase in energy consumption be deemed unacceptable.

2.3. Comparison of Energy Consumption Attribution Approaches

In this section, we present a comparison of three approaches to energy consumption attribution found in our literature review. We additionally motivate our choice to use statistical sampling in EDATA, and the benefits and drawbacks to each of these approaches, and in which use cases they may be a more appropriate choice.

2.3.1. Instrumentation Based Tracing

Prior to EDATA, nearly all state-of-the-art tools for estimation energy consumption on Android at a fine-grained level used some form of instrumentation, such as bytecode instrumentation, to collect information on exactly when methods are executed by an app. Some tools, such as *PETrA* [17] and *MELTA* [18] report energy consumption at the method-level, and instrument the beginning and end of each method to obtain complete data on when methods are executed. *vLens* [28] uses a similar instrumentation methodology, but uses a more complex path analysis to determine where to place execution probes, and further analyzes their collected data to estimate energy consumed at the line level. Implementation based approaches have several benefits: First, as the exact number of method calls is known, the energy consumed per-call can be accurately estimated, either by estimating the total consumption and dividing this by the number of calls, or by attempting to precisely measure the amount of energy consumed for an individual call. Second, instrumenting function calls ensures that no function, regardless of duration, will be missed, which can make the resulting estimations more accurate.

While there are several compelling benefits to the use of instrumentation-based tracing, there are a number of drawbacks. First, Android's built in tracing[10] has several limitations, most notably that it is unable to trace C/C++ code. If an app makes significant use of native libraries, this could entail an unacceptable loss of information. Trace-based collection also imposes a level of overhead, though apart from *vLens*, where the authors estimate their overhead at an average of about 4% [28], few works report this overhead, as it is a minor concern in testing environments compared to the overall accuracy of the results. However, if low overhead is desired, such as in a use-case where an analysis is performed on end-user devices, trace-based approaches may be unacceptable.

In spite of these drawbacks, instrumentation-based tracing may still be relevant, particularly combined with the use of Android's release-mode profiling[11], which allows the use of Android's built-in tracing, as used in works mentioned above, with release-mode apps. Though the overhead involved with release-mode tracing must still be investigated, the increased information available may be worth the cost, particularly if an app makes limited or no use of native code.

2.3.2. Static Analysis Estimation

In order to lower the barrier to entry for developers to analyze the energy consumption of their app, some approaches use static analysis of source code to estimate energy consumption, removing the need for a physical test device entirely. This approach has several key benefits: Since static analysis tools can be run at any time, developers do not have to wait to get feedback on their changes. Such a tool could also easily be integrated into a continuous integration system, providing feedback per commit or release. As these benefits are most apparent in the context of mobile devices, where development and use are done on completely different systems, it should come as no surprise that much of the literature around this approach focuses on mobile devices. *eLens* [22], the predecessor to *vLens* [28], builds on this by allowing use of collected execution traces in combination with a pre-defined energy model to provide energy consumption estimates of dynamic behavior. Aside from *eLens*, most code-analysis-

[10]`https://developer.android.com/studio/profile/record-traces#configurations`
[11]`https://developer.android.com/guide/topics/manifest/profileable-element`

based approaches make use of machine learning models, such as *MLEE* [3], *GreenScaler* [8], and the work done by Romansky [42]. In general, static analysis based approaches have been shown to be effective, and a good trade-off between ease of use and accuracy of estimations when compared with on-device measurements. In light of our results in Sections 5.1.1.3 and 6.2.1 which highlight differences between devices, it is possible that estimates generated by a machine learning model trained on one device may not transfer to the actual performance of another device. Without knowledge of particular device characteristics, such as CPU capabilities and runtime optimizations performed by the just in time (JIT) compiler, variations in accuracy between devices are inevitable. However, this does not necessarily detract from the usefulness of this methodology. Developers can, at the expense of some time, train a model on each device that they are interested in testing. Additionally, trained models can be shared, allowing a single developer to ”test” on many more devices than would be the case when requiring a physical test device.

2.3.3. Statistical Sampling

Statistical sampling-based energy consumption estimation is used by EDATA, ALEA [34], the Android Profiler (in certain modes)[12], and is a known technique in the context of HPC performance profiling to provide controllable overhead [46]. Instead of instrumenting function calls or paths to determine the exact time when each function or path was entered and exited, statistical sampling collects samples of program execution at some defined interval, and attempts to determine the probability, and thus total runtime, of each observed function. A more detailed explanation of a specific methodology employing statistical sampling can be found in Section 3.5.2.1, where we explain the approach used by EDATA. Statistical sampling has one primary weakness: it is reliant on the debug information provided in the executable being sampled, and on the output of the stack unwinder [46]. On Android, both missing debug information[13] and the output of the stack unwinder[14] are known limitations, and *simpleperf* already includes tools with which to mitigate their effects. While statistical sampling is limited in terms of call-chain completeness and detection of executed methods, its low overhead is particularly interesting in the context of mobile devices. As the overhead is decoupled from function calls, code areas with a large number of calls will not be disproportionately affected, and long-term battery life tests will be less impacted. Finally, a statistical sampling based approach could potentially see use in production environments, with the low overhead and lack of instrumentation are essential to use on end-user devices. In addition to low overhead, sampling on Android using *simpleperf* allows analysis of JIT compiled code in addition to pre-compiled native and java virtual machine (JVM) code.

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

[13]https://android.googlesource.com/platform/system/extras/\+/master/simpleperf/doc/README.md#Fix-broken-callchain-stopped-at-C-functions

[14]https://android.googlesource.com/platform/system/extras/\+/master/simpleperf/doc/README.md#Fix-broken-DWARF-based-call-graph

3 Tool Design

3.1. Our Approach

The primary goal of our approach is to estimate the energy consumption of Android applications at a method-level granularity, leveraging the built-in energy monitoring hardware/software found on virtually every modern mobile device. One of the primary goals of our approach is to reduce the measurement overhead to a sufficiently low level to be used in a production setting. As such, we will also take advantage of the `profileable` flag, added in Android 10, which allows us to measure apps compiled in release mode. We also will use statistical sampling, using *simpleperf* to collect samples, instead of the more common instrumentation based methods. The overhead of statistical sampling is consistent and can be reduced as desired by lowering the sampling frequency, which is desirable for our use case, as sampling overhead can influence the recorded energy consumption. Our approach consists of three main components, as well as a set of test scripts designed to automate the process of gathering test data. These components have been designed in a modular fashion, and can be replaced as desired if, for example, new capabilities are released in a future version of Android. We have further implemented an app containing each of the test cases outlined in chapter 4.

3.2. Statistical Sampling for Energy Estimation

In order to reduce the measurement overhead of our approach, we have opted to use statistical sampling to gather information on the runtime behavior of the application under test (AUT). As described in Section 2.3.3, statistical sampling has been used in the past as a low-overhead approach to performance profiling [46], and has been known to be effective in the context of energy consumption for many years, being used by Flinn and Satyanarayanan in their *PowerScope* tool, from their 1999 work [19]. Our approach is based on a more recent work by Mukhanov et al. [34], who implemented a probabilistic sampling model (ALEA) to estimate the energy usage of native programs on Linux at a basic-block level[1]. This entails that the instructions will always be executed in exactly the same order, and none can be skipped. In contrast to the programs analyzed by ALEA, Android apps run not only ahead of time (AOT) compiled C++ code, but also a mixture of AOT and just in time (JIT) compiled and interpreted code on the java virtual machine (JVM). These different compilation methods make deriving the source line related to a given instruction more difficult as the necessary information is not always available, particularly in release mode on a non-rooted device. In light of these restrictions, we chose to focus on providing energy attribution at the method level, as Android stack traces contain sufficient information to derive the method containing a particular instruction.

3.2.1. Simpleperf

Our approach makes use of the *simpleperf* [44] tool, a fork of *perf*[2] developed by Google for use with Android. Simpleperf provides a number of useful features - most notably the ability to record call-graphs

[1]A basic block is a unit of (compiled) code which contains no jump instructions, except at the beginning and end (such that the beginning may be jumped to, and the final instruction may be a jump).

[2]`https://perf.wiki.kernel.org/index.php/Main_Page`

of JIT compiled and interpreted JVM code in addition to AOT compiled JVM and native code.

Simpleperf is a tool developed by Google for use with Android, and is distributed both on Android devices and as part of the Native Development Kit (NDK). The NDK version can be sideloaded, allowing use of newer versions than were distributed with a device. *Simpleperf* is a fork of the Unix *perf* tool, with many of the same capabilities, though it does not implement all of the features found in *perf*. It has been modified for use with Android, adding the ability to use platform capabilities (such as the `profileable` flag) to profile apps without the need for root, and is capable of profiling not only native C++ software, but any software using the JVM (such as Java and Kotlin), and since Android 9 does not require JVM code to be AOT compiled. One of *simpleperf*'s primary functions is to perform event-based sampling – it can track a number of events, many of which are triggered by hardware, such as `cpu-cycles`, which is incremented per CPU clock cycle. For use in EDATA, we chose to use the `task-clock` event, which triggers at a given time interval and is tracked per-thread.

We invoke *simpleperf* as follows, using a configuration intended to emulate the sampling approach of ALEA [34].

```
    simpleperf record -e task-clock -f {sample_Hz} --call-graph dwarf --
clockid monotonic_raw --app {AUT}
```

The reasoning for our configuration is as follows:

3.2.1.1. DWARF callchain unwinding

DWARF[3] is a standard format for debugging information used by compilers to map information about the source code to the compiled binary[4]. *Simpleperf* supports both DWARF-based callchain unwinding and stack-frame-based unwinding. We selected *DWARF*-based unwinding as it is more portable across architectures (particularly 32-bit ARM), and works better with Java/Kotlin code. The primary drawbacks of *DWARF* unwinding is an increase in CPU usage and limited callchain size. In practice, we have not observed either of these to be a limitation, as we generally sample at well under the recommended maximum of 4000Hz [45], and *simpleperf* is capable of re-building lost callchain information based on callchains from different samples. We have observed that a high sample rate in combination with `trace-offcpu` can lead to "lost" samples due to *simpleperf*'s sample buffer being too small, but our current approach does not use this option. Passing the `--call-graph dwarf` flag to *simpleperf* enables DWARF-based unwinding, and is used in our evaluation. As suggested by Google [1], if profiling native code on 64-bit ARM, using stack-frame-based unwinding may be preferred. As most Android apps primarily use Java/Kotlin, we do not use it in our evaluation.

3.2.1.2. ClockID

Simpleperf allows selection from a set of different standard Linux system clocks[5] to timestamp samples. The clocks supported are: realtime, monotonic, monotonic_raw, boottime, and perf. We chose the `monotonic_raw` system clock as it is also available through the Android API, allowing our environment sampling utility to use the same clock for timestamps as *simpleperf*, removing the need for synchronization of the different records after the fact. Additionally, this clock is guaranteed not to jump backwards or ahead, and is not subject to NTP adjustments [10], which guarantees that we will be able to accurately measure the time between samples, and that there will be no overlaps in sample timestamps caused by clock adjustments.

3.2.1.3. Event selection

Ideally, we would sample the AUT using exactly the same approach as ALEA: systematic sampling, where samples are taken at a fixed rate, using the slight inaccuracies inherent to timers as "randomization" [34]. ALEA uses CPU clock cycles as a unit of time, which is a sufficient metric in their application, as it is common for servers and high performance computing (HPC) systems to have their clock rate fixed, at least under load. However, mobile devices do not have this property, and further cannot guarantee that a given workload will be run exclusively one on type of CPU core. We have therefore chosen to use the `task-clock` software event, which triggers based the runtime of each individual thread.

[3] `https://dwarfstd.org/`
[4] `https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf`
[5] `https://man7.org/linux/man-pages/man2/clock_gettime.2.html`

While there is no support for adding random offsets to the sampling time, we expect similar variation in the system clock as observed by Mukhanov et al [34]. As currently implemented in *simpleperf*, this method comes with the following drawback:

The `task-clock` timer is kept per-thread, and will trigger after exactly **n** nanoseconds on-cpu have elapsed. This means that, in the worst case, if a thread runs for **n-1** nanoseconds before exiting, it will never be sampled. Methods executed at thread start, running for less than the sample period, will similarly not be sampled. The thread-specific nature of the `task-clock` timer limit the ability of our tool to perform multi-threaded sampling, as each thread will be sampled individually at the moment its timer reaches the defined count. This prevents us from implementing the multi-threaded analysis method from Mukhanov et al. [34], and lead us to make the simplifying assumption where we model the AUT as single-threaded.

3.2.1.4. App targeting

Simpleperf is capable of automatically targeting an app's process when given its package name. To collect stack traces from running apps under the Android security model, it is necessary to first switch its user privileges to that of the app, using `run-as`. A side effect of this is that, without root access, a single *simpleperf* process is only capable of profiling a single app at a time. For the purposes of our approach, this is sufficient. Future work attempting to profile more than one application, or the whole system, would need to use a rooted device in order to overcome this limitation.

3.3. Energy/environment measurements

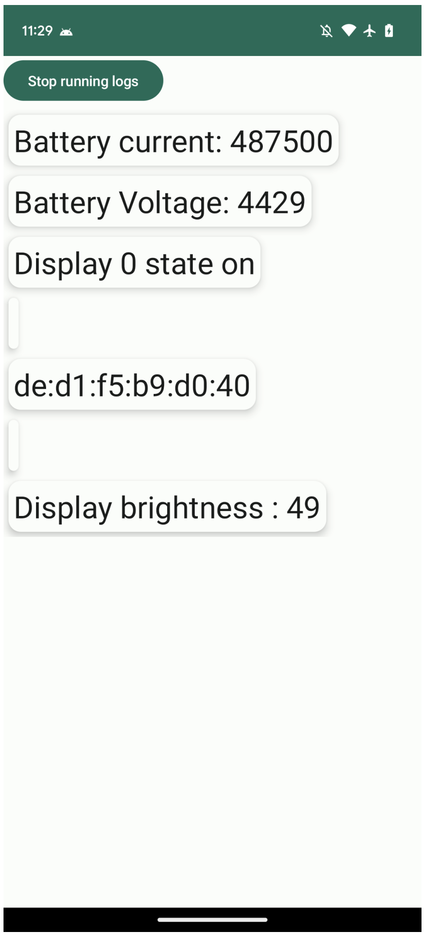
3.3.1. Measurement approach

To measure the energy consumption of an app, we need to obtain the instantaneous power usage of the device under test. There are a number of ways to do this, among which are external hardware instrumentation [27, 23, 24], use of built-in sensors [17, 34], and static-analysis-based estimation, often utilizing machine learning [42, 3] . To keep our tool simple, and not require any initial setup (e.g. pre-training a model), we chose to use Android's `BatteryManager`[6] API for retrieving real-time power measurements. While Android does not provide instantaneous power information, this can be derived by combining current and battery voltage. The update rate of the instantaneous current drawn from the battery varies by device; on our two devices, we observed update rates of 1Hz and 5Hz. This is in stark contrast to external hardware based power measurements, which can provide update rates in the KHz range[7].

3.3.2. Collection App

Unlike prior solutions, such as DiNucci et al. [17], we collect environmental data using a dedicated sampler app. This app records changes in energy consumption using the `BatteryManager` API, but also changes in the state of the phone: Display state and brightness, WiFi strength, and cellular signal strength. These states can be used to cluster energy consumption based on environmental factors, but is not currently used by our approach Our collection app also includes a UI, which can be used for debugging purposes to display the current value of each supported sensor. This UI is shown in Figure 3.1, as displayed on a Pixel 6a. The cellular and WiFi signal strengths are not shown, as airplane mode is active and a change in WiFi strength must occur before the strength is reported.
We chose to use a separate collection app for two reasons: The first reason is to ease collection and formatting of data, as all of our desired data can be acquired through a single tool. Secondly, unlike DiNucci et al. [17], we obtain current measurements through the `BATTERY_PROPERTY_CURRENT_NOW` property instead of calculating it using the hardware states reported by `dumpsys batterystats`. We were unable to find a record of battery current measurements within the logs provided by the various Android subsystems, and believe that the battery current is not recorded due to the relatively high update rate of current measurements.

[6]`https://developer.android.com/reference/android/os/BatteryManager`
[7]`https://www.msoon.com/specifications`


11:29

Stop running logs

Battery current: 487500

Battery Voltage: 4429

Display 0 state on

de:d1:f5:b9:d0:40

Display brightness : 49

Figure 3.1: Collection App UI

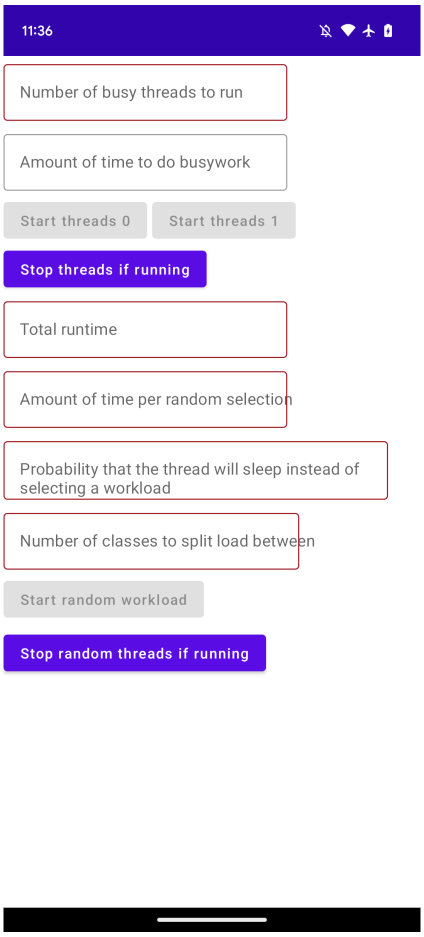

11:36
Number of busy threads to run
Amount of time to do busywork
Start threads 0
Start threads 1
Stop threads if running
Total runtime
Amount of time per random selection
Probability that the thread will sleep instead of selecting a workload
Number of classes to split load between
Start random workload
Stop random threads if running

Figure 3.2: Worker App UI

3.3.3. Perfetto

Beginning in Android 14, the prior two methods of data collection will be made obsolete due to improvements made to the *Perfetto*[8] tool. These improvements will allow *Perfetto* to collect atoms exposed by *statsd*[9], which include each of the power and environmental data points that we currently collect. As *Perfetto* also offers integration with *Simpleperf*, using these new capabilities would simplify our tool, and allow it to run with less overhead by removing the need for a companion app. *Perfetto* comes with one primary drawback: due to its integration with the native Android Runtime (ART), newer versions cannot be sideloaded onto older versions of Android, unlike *simpleperf*. Once Android 14 is widely adopted, we expect *Perfetto* to replace our companion app.

3.4. Test App

In order to implement our tests defined in chapter 4, we developed a separate app. This app is controlled in a similar way to the data collection app in Section 3.3.2. Its primary method of control is through intents[10], which can be sent either manually using the `am` utility available in the Android Debug Bridge (ADB) shell, or automatically as part of a test script. There is also a basic UI, shown in Figure 3.2, that contains controls for two of its workloads: a “busywork” workload that performs busy work on one or more threads, and the random test defined in Section 4.3. We used this UI during early stages of development, but exclusively used intents in our evaluation. The app is designed such that new workloads can be added with minimal changes, and can be extended with new tests in the future.

3.5. Data Analysis

Once our collected data has been moved from the device under test to the host system, it is analyzed in a two-step process.

3.5.1. Intermediate Trace Representation

The first step in our data analysis is to convert the collected data to an intermediate representation. The purpose of this is to abstract the specifics of the collected data away from the further stages of our analysis pipeline, so that changes in the data collection do not necessitate changes to the rest of the process.

We abstract the collected data in the process shown in Figure 3.3 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 or more actively executing threads from the AUT, the instantaneous power draw of the whole device, the time until the following sample (or, its ‘period’), and the entry’s timestamp. We note that while we assume in our implementation that the timestamps of our different data sources match precisely, this is not a hard requirement. The exact method of matching timestamps between data sources can be considered an implementation detail handled by the process of converting collected data to the intermediate trace representation. This stage of the process is responsible for calculating the instantaneous power draw at the precise time of each callchain sample. As with matching timestamps, the precise method with which this is done can be considered an implementation detail, while keeping in mind that this is a very important step. A more precise measurement of instantaneous power will result in the end result – energy consumption attributed to methods – being more precise. We chose to implement this by creating a time-series of instantaneous power draw, updated each time either the voltage or current reported by the device changes. When determining the power draw at the time of a callchain sample, we find the most recent measurement in the series **prior** to the sample.

3.5.1.1. Our Implementation

We concretely implement this process shown in Figure 3.3 as follows: We collect callchain samples using *simpleperf* as explained in Section 3.2.1, and current/voltage measurements as described in Section 3.3.2. To implement steps 1 and 2, we walk through our sampler app’s logs in sequence, creating a list of Python objects representing the logged data. We then sort the list by time, to ensure that any out-of-order entries in the log are fixed. We then create a new list of `Power` objects by iterating

[8]`https://perfetto.dev/`

[9]`https://source.android.com/docs/core/ota/modular-system/statsd`

[10]`https://developer.android.com/reference/android/content/Intent`

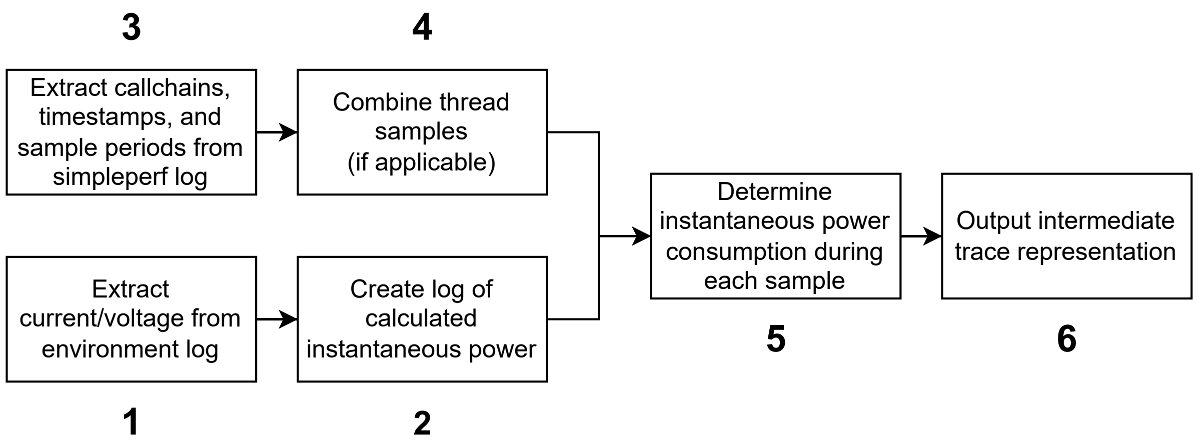

3
Extract callchains, timestamps, and sample periods from simpleperf log
4
Combine thread samples (if applicable)
Extract current/voltage from environment log
1
Create log of calculated instantaneous power
2
Determine instantaneous power consumption during each sample
5
Output intermediate trace representation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

Figure 3.3: Abstraction Process

over the voltage and current measurements, creating a new `Power` object by multiplying the most recent current and voltage measurements each time either is updated. Step 3 is implemented using the *simpleperf* tools provided as part of the Android NDK. The `perf.data` log output by *simpleperf* is iterated over, and the callchain, timestamp, and period of each sample are copied to a Python object, named `AppState`. We do not implement step 4, and explain this decision in Section 3.5.1.2. We then combine the two lists of objects in step 5: for each `AppState`, we find the `Power` object with the closest preceding timestamp, and add it to the `AppState`. This allows us to keep track of which `AppState` samples were taken between `Power` measurements. Finally, for step 6, we output the `AppState` list, along with a list of `Power` objects associated with one or more `AppState` objects.

3.5.1.2. Representation of Apps as Single-Threaded Programs

In our implementation, we have chosen to simplify the attribution process as follows: we assume that the AUT uses no more than one thread at a time, allowing us to assume that each sample reported by *simpleperf* was the only thread executing on the device CPU at the time of sampling. While this assumption reduces the accuracy of our approach when multiple threads are executing simultaneously, we were unable to find a reliable way to collect samples from multiple threads simultaneously. The simpleperf `--trace-offcpu` flag records when a thread is scheduled on and off the CPU, and this information could be used to determine which threads were running at the time of a sample. However, we were unable to find a method with which to synchronize samples across all active threads, and thus decided not to use it. Through manual analysis of app samples, we found that, outside of use-cases involving heavy background processing, Android apps do not typically actively run multiple threads for extended periods of time, which we believe reduces the impact of this assumption. We also purposefully ignore other processes running on the system, as we cannot collect information over processes external to the AUT without root access. We have also chosen to use the sample period reported by *simpleperf* directly, instead of computing the true time between two samples. In our configuration, *simpleperf* reports the period between samples on a single thread, with respect to that thread's CPU time, rather than app-global wall-clock time. This can cause two sequential samples to be closer or farther apart in wall-clock time than the given period would imply. We justify this choice for two reasons: First, using the provided period greatly simplifies our data collection, as – since Android apps frequently idle – we would need to account for idle time when calculating sample periods otherwise. Second, the given sample period guarantees that the sampled thread has run for the amount of time in the sample period, allowing us to correctly account for the total CPU time of each thread, which we assert will limit the impact of this decision.

3.5.2. Energy Attribution Approach

When attributing energy consumption to methods, we categorize it as either **local** and **non-local**. Local energy consumption is defined as energy consumed while code 'local' to the method was being executed – that is, code which is part of the method itself. Non-local energy consumption is defined

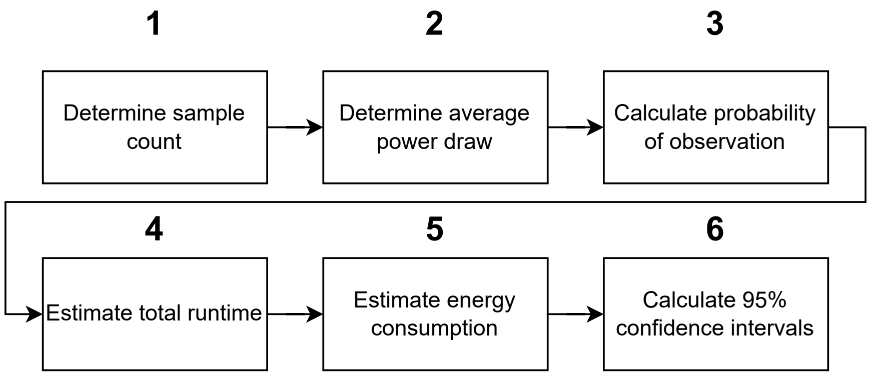

1
Determine sample count
2
Determine average power draw
3
Calculate probability of observation
4
Estimate total runtime
5
Estimate energy consumption
6
Calculate 95% confidence intervals

Figure 3.4: Energy Attribution

as energy consumed while the method was present in the sampled callchain, excluding local energy consumption.

A diagram of our energy attribution approach is shown in Figure 3.4. The steps shown in this figure are performed twice per method – once each for local and non-local energy. In step 1, the total number of either local or non-local samples is found from the intermediate trace representation. In step 2, the average power draw for the method is calculated by averaging instantaneous power measurements associated with it, as shown in Equation 3.7. Steps 3 and 4 are performed similarly to the method used by ALEA [34], but with some modifications, and are further explained in Section 3.5.2.1. With the estimated total runtime, we then estimate the energy consumption of the method, step 5, in Section 3.5.2.2. Finally, for each of the values calculated in steps 3 through 5, in step 6 we calculate 95% confidence intervals as defined in Section 3.5.3.3.

3.5.2.1. Probability calculation

We use the method defined by Mukhanov et al. [34] to estimate the probability of each method being sampled while actively executing, as well as 95% confidence intervals. We reproduce their methodology here, explaining differences where appropriate. The primary change made in our version 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} :

$$X_{method} = \begin{cases} 1 & \textit{method} \text{ is the sampled method} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

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:

$$p_{method} = P(X_{method} = 1) = \frac{C^1_{t_{method}}}{C^1_{t_{exec}}} = \frac{\sum_{j=1}^{k} latency^j_{method}}{t_{exec}} \tag{3.2}$$

$$\frac{\sum_{j=1}^{k} latency^j_{method}}{t_{exec}} = \frac{t_{method}}{t_{exec}} \tag{3.3}$$

We define t_{method} as the total CPU time of a given method. Approaches utilizing method/function tracing generally record t_{method} directly, by recording timestamps at method entry and exit. When using a sampling-based approach, the total run-time of the method must be approximated based on the recorded samples and the total run-time of the app under test. To calculate t_{method}, we multiply the probability of the method being observed, p_{method}, by the total CPU time of the app, as in Equation 3.4.

In contrast to Mukhanov et al [34], 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 3.2 thus represents the probability of sampling a method equaling the ratio of its CPU time to the total CPU time of the app.

$$t_{method} = p_{method} \cdot t_{exec} \tag{3.4}$$

In order to calculate the true runtime of a method, we need the true probability of it being observed in a sample. However, the true probability is not known, and we must estimate it based on the collected samples. Using the same process as Mukhanov et al. we find the maximum likelihood estimator $\hat{p}_{method}$ for $X_{method} = 1$ in Equation 3.5, where n is the total number of samples collected, and n_{method} is the number of samples where *method* was sampled.

$$\hat{p}_{method} = \frac{n_{method}}{n} \tag{3.5}$$

With $\hat{p}_{method}$, we are able to estimate the method's total runtime, $\hat{t}_{method}$, in Equation 3.6.

$$\hat{t}_{method} = \hat{p}_{method} \cdot t_{exec} = \frac{n_{method} \cdot t_{exec}}{n} \tag{3.6}$$

3.5.2.2. Energy model

We faced two major difficulties implementing Mukhanov et al.'s energy model [34]. First, while we used *simpleperf* to collect the callchain, it is not capable of sampling energy consumption. This prevents us from collecting energy samples in sync with callchain samples. Secondly, the update rate of energy sensors found on typical Android devices is far lower than the devices used in their validation. However, the low update rate enables us to use a different collection approach: we record every value reported by the on-device sensors, and assign each callchain sample to one of these values. We chose to assign each callchain the closest sample at an equal or earlier timestamp, however, other strategies are also possible, such as interpolating between samples, or assigning to future samples.

While we use a different method for energy collection, the rest of the energy model remains the same. Equation 3.7 shows the calculation of the maximum likelihood estimator of the mean power draw of *method*, and Equation 3.8 shows the calculation of the maximum likelihood estimator of the total energy consumed by *method* over the course of the test.

$$p\hat{o}w_{method} = \frac{1}{n_{method}} \cdot \sum n_{method} i = 1 pow^{i}_{method} \tag{3.7}$$

$$\hat{e}_{method} = p\hat{o}w_{method} \cdot \hat{t}_{method} \tag{3.8}$$

3.5.3. Our Implementation

Our implementation of the approach described in 3.5.2 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 3.5.1. This phase corresponds with steps 1 and 2 in Figure 3.4. Once this phase is complete, we have sufficient information for each method with which to perform steps 3 through 6.

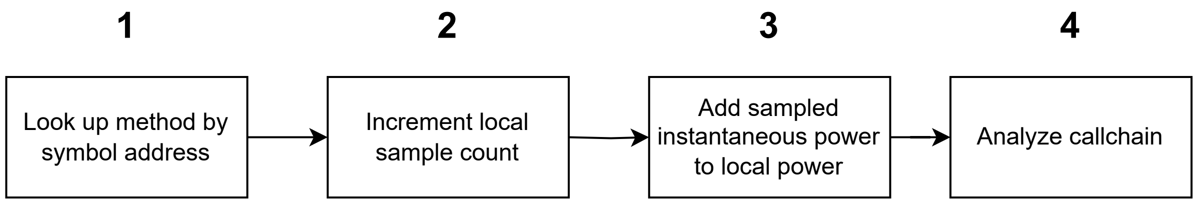
3.5.3.1. Method Extraction

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 3.5a, 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.

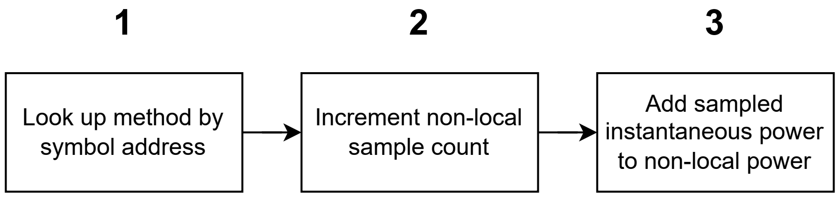
First, step 1 is performed using the symbol address of the method containing the current instruction at sample-time. Using this address, we check if a matching `Function` is present in our dictionary, and if so, retrieve it. If not, we create a new `Function` and add it to the dictionary.

Step 2 and 3 are performed by incrementing counters in the `Function` object with the appropriate values.

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 3.5b. This process is identical to the 'local' process described


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

Figure 3.5: Method Extraction Phase

above, but increments different counters that track ‘non-local’ values. We filter duplicate callchain entries to ensure that methods that appear multiple times in the callchain do have non-local run-time attributed to them more than once per sample,.

Once this process has been performed for each sampled app state, each method observed during sampling has been entered into our dictionary, and its average instantaneous power draw and sample count, both local and non-local, are known.

3.5.3.2. Post-Processing

The second step of our implementation is the ‘post-processing’ phase shown in Figure 3.6, which implements steps 3 through 6 of the process shown in Figure 3.4.

To perform this phase, we iterate over each method in the dictionary, and perform each of the four steps in sequence for local and non-local samples. In step 1, we generate the maximum likelihood estimator for the probability of observing the method as shown in Equation 3.5. In step 2, we use this probability to estimate the total run-time of the method, as defined in Equation 3.6. In step 3, we generate the estimated energy consumption of the method, as in Equation 3.8. As these generated values, $\hat{p}_{method}$, $\hat{t}_{method}$, and $\hat{e}_{method}$ are maximum likelihood estimations, we are able to generate confidence intervals, and EDATA provides 95% confidence intervals for all three.

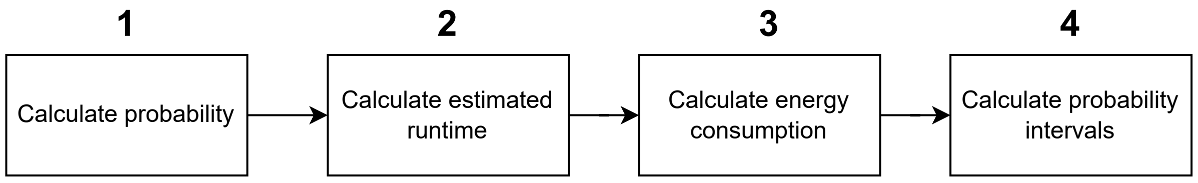

1
2
3
4
Calculate probability
Calculate estimated runtime
Calculate energy consumption
Calculate probability intervals

Figure 3.6: Post-Processing Phase

3.5.3.3. Confidence intervals

We calculate confidence intervals, step 4 of our post-processing phase in Figure 3.6, using the same method as Mukhanov et al. [34]. As we have not made modifications to their approach, we do not reproduce it here. We generate 95% confidence intervals for each of the three maximum likelihood estimators generated during post-processing. An example of the output for the observation probability, $\hat{p}_{method}$, is shown in table 3.1.

In order to calculate the confidence intervals for instantaneous power, and thus energy, we filter the samples to discard those with duplicate power samples, where the callchain samples were taken in between updates from the device’s current sensor. This approach means that the confidence intervals for energy consumption and power may not be centered on their displayed means, depending on the update rate of the sensor and the chosen sample rate. We have chosen this approach as we observed that not filtering duplicates could result in very small confidence intervals in some conditions, such as a short test run with a high sample rate. Since these confidence intervals are provided for use by users of EDATA, and our evaluation does not use them, we chose to discard duplicate power samples as we consider this to be a more accurate representation of the “true” confidence of our measurements. We note, however, that we have not fully validated this choice.

Local Probability	Non-local Probability	Local Probability Interval	Nonlocal Probability Interval
0.25	0.00	[0.16, 0.33]	[-1, -1]

Table 3.1: Sample data showing the probability of observing a function and the corresponding probability intervals

3.5.3.4. Interpretation of results

After completing this process, EDATA produces a list of methods, and reports the information provided in table 3.2 for each method. “Local” metrics are calculated based on code executed within the body of the listed method, excluding any function calls – in this case, the method listed was at the end of the callchain. “Non-local” metrics are calculated based on code executed when the listed method appeared in the callchain, but not at the end.

Output Data			
Data	Local	Non-local	Confidence Interval
Sample Count	X	X	
Probability	X	X	X
Runtime	X	X	
Energy	X	X	X
Average Power	X	X	X

Table 3.2: Data provided in output of analysis step

3.6. Test Orchestrator

The final component of EDATA is the *test orchestrator*, which controls *simpleperf* as described in Section 3.2, our data collection app as described in Section 3.3, and a test workload that can be defined by the user. The *test orchestrator* handles all parts of the testing process, with the exception of any one-time setup actions that are difficult or impossible to automate (such as performing a factory reset).

The *test orchestrator* splits the test process into a series of phases, which enables us to use a modular design in which components implementing desired phases can be easily added to the test process. Central to the test orchestrator is the primary test workload, which controls the app under test, or AUT. This workload implements a separate interface to the rest of the components, as it has a different set of responsibilities. There is also a sleep workload, which can be used for manually performed tests, or for exploratory testing where no automated tests are available.

3.6.1. Test Process

The test process performed by the *test orchestrator* consists of four phases, including the primary test loop. The primary test loop itself consists of five phases, and will be described in depth in 3.6.2. The four phases are executed in the order shown in Figure 3.7, and each component may optionally define a function to be executed in each phase. In this section, we will describe each of the phases carried out by the *test orchestrator* and their general purpose. Section 3.7 will describe the specific actions we implemented for each of these phases, if any.

3.6.1.1. Pre-test setup

This phase is performed before executing a ‘test loop’ which is a loop where a single test, with a single configuration, is executed a fixed number of times. This step contains actions that need to be performed between different tests in order to ensure a consistent test environment, but are not executed between runs of the same test.

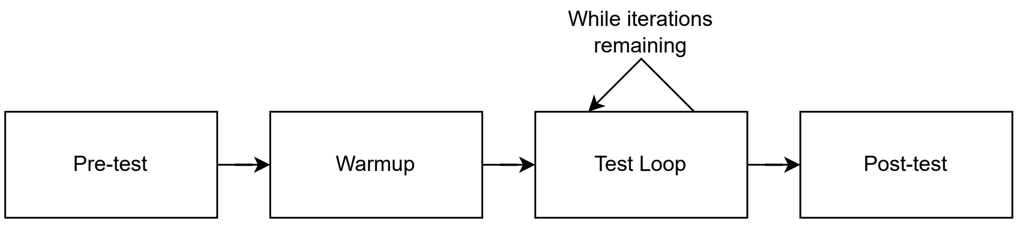


Figure 3.7: Test Orchestrator Process

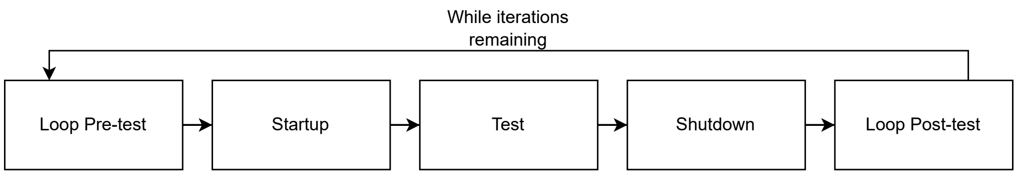


Figure 3.8: Test Orchestrator Loop

3.6.1.2. Warmup

This phase performs a shortened version of the primary test workload to allow the ART to perform profile-guided JIT compilation. We have observed that the stack traces generated by EDATA can vary over the first minutes of execution, thus a separate warm-up phase is crucial to obtaining accurate data.

3.6.1.3. Post-test teardown

This phase is performed after the test loop has finished, and is the last phase in the test process. It is primarily used for uninstalling the AUT from the device under test, but can be used for any actions that should be performed after the test loop.

3.6.2. Test Loop

The test loop consists of five phases, and each phase is executed once for each **iteration** of a test, as shown in Figure 3.8. As with each phase described in 3.6.1, components may optionally provide a function to execute per-phase, with exception of the ‘test’ phase, which is exclusively managed by the test workload.

3.6.2.1. Loop Pre-Test

The loop pre-test phase is intended for setup actions that may potentially run for long periods of time, such as deletion of files or pre-processing of data.

3.6.2.2. Startup

The startup phase is intended for short-running actions that take place immediately prior to the primary test workload. This phase is the final phase accessible to test components prior to the test workload, so any components that must be activated on-device are started in this phase.

3.6.2.3. Test

This phase is exclusively accessible to the primary test workload. Three functions are provided as part of the workload interface: `start_test()`, `wait_for_test()`, and `stop_test()`. These functions are intended to perform last-second setup, block the test process until the workload has completed, and immediate tear-down, respectively. As test workloads have access to each of the other test phases, and data collection will be started prior to `start_test()` and ended after `stop_test()`, only short-running, essential actions should be performed in this phase.

3.6.2.4. Shutdown

The shutdown phase is intended for short-running actions to be performed immediately following the primary test workload, such as shutdown of each data collection component. As data collection components will still be running in this phase until their shutdown function has been called, long-running operations should not be placed in this phase.

3.6.2.5. Loop post-test

This is the final phase in the test loop, and contains potentially long-running operations such as copying data to the host machine.

3.7. Empirical Evaluation Methodology

3.7.1. Testbench Setup

This step is taken prior to beginning a test or series of tests, and contains actions that do not need to be repeated once completed. As these steps generally consist of manual actions, they are not part of the scripts included in the test orchestrator.

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. Linares-vasquez, et al. defined a methodology in which these factors are mitigated by enabling airplane mode, disabling all other processes and services to the extent possible, and preventing the phone from moving to reduce sensor power draw. Similar strategies are used by diNucci et al. [17] and Bouaffar et al. [6]. Jagroep et al. identify similar concerns when measuring the energy use of applications running on a dedicated server, and in addition to disabling unnecessary services, recommend allowing the system to enter a 'steady-state' after rebooting to avoid interference from background operating system services. [24]

Following the examples laid out above, the test devices used for evaluation will be set up as follows: We will remove all third party apps to the extent possible, either through regular uninstallation or by performing a factory reset. The cellular modem will be disabled by use of airplane mode. Bluetooth will be disabled through the system toggle, with the exception of test cases that make use of Bluetooth hardware. The device's display will be off, except for tests involving Bluetooth scanning, as the display must be switched on to scan for devices.[11]

3.7.2. Pre-Loop

To prepare for a test loop, we first install the app under test using the command:

```
adb install -g /path/to/package.apk
```

The installed app is then AOT compiled using the command:

```
adb shell cmd package compile -m speed -f name.of.package
```

By performing AOT compilation on the installed app prior to testing, we intend to reduce the variation in energy consumption caused by execution of unoptimized code, as well as simulate a more 'production-accurate' profiling environment, as some devices will automatically perform AOT compilation during downtime.[12]

While the AOT compilation process will optimize the app, Android versions later than 7 also include a JIT compiler that performs optimizations in cooperation with the AOT compiler.[13] To reduce the confounding effects of this compiler, we then run a shortened version of the current experiment for one minute, followed by a two minute wait period to allow the system to enter a steady-state. As all of the test cases used in validation contain little code, we expect this relatively short warmup period to be sufficient for the JIT compiler to profile and compile the hot code in the test cases.

3.7.3. Primary Test Loop

The primary test loop is responsible for running a single iteration of a test. This loop consists of three primary steps: setup, execution, and teardown. The setup and teardown steps are further split into two, with each containing an "immediate" and "non-immediate" phase, such that short-running time-sensitive actions can be performed immediately prior to or following the execution step. For brevity, we do not explicitly describe these sub-phases.

First, in the **setup** step, the testing environment is set up. This includes preparatory steps taken on both the test device and the host machine, and may differ between workloads. For example, the Bluetooth test must ensure that the device display is active, and will perform this check in the loop setup step.

[11] `https://developer.android.com/reference/android/bluetooth/le/BluetoothLeScanner#startScan(android.bluetooth.le.ScanCallback)`

[12] `https://source.android.com/docs/core/runtime/configure`

[13] `https://source.android.com/docs/core/runtime/jit-compiler`

Next, the **execution** step begins. In this step, *simpleperf* and the sampler service are both started, followed by the test workload. Then, if desired, the test orchestrator will disconnect from wireless ADB, to reduce the effect of ADB on the test results. The test workload component of the test orchestrator uses a slightly different mechanism than the other components - it will sleep for the duration of the execution phase, blocking the thread until the test is complete. It will then ensure that the on-device test workload has exited before allowing the orchestrator to continue, which will exit the **execution** phase.

Finally, in the **teardown** step, *simpleperf* and the sampler service will be shut down, and their recorded data will be copied from the device to the host machine. Test workloads may also implement functionality for this step, such as the random workload, which retrieves test information from the device's `logcat`.

3.7.4. 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.

3.8. Case Study Methodology

3.8.1. Prior Work

Pereira et al. [40] evaluate their SPELL toolkit by providing developers with a simple programs known to contain energy bugs, and assigning them to use either the SPELL toolkit, an existing profiler, or neither of the two to modify them to reduce energy consumption. They then evaluated the modified programs to ensure no bugs were introduced, and measured their energy consumption and execution time. They then evaluated the behavior of the test participants, observing which methods they chose to modify and how long they took to make their modifications.

3.8.2. Our Approach

Our case study focuses on the use case where a developer uses 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, instead of release mode.

4 Evaluation

The goal of EDATA is twofold; We want to provide information on the relative energy consumption of individual methods for a given program trace, as well as compare two traces to each other to determine whether there is a meaningful difference between them. In order to validate our approach, we have defined the following 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?

This chapter introduces each of these research question, relates them to the energy debugging and energy testing use cases of EDATA, and defines the methodology with which we will answer each of these questions.

4.1. Test Devices

We performed our evaluation on two test devices, an Adyen AMS1 and a Google Pixel 6a. 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. We performed our empirical evaluation using 20 iterations per test on the AMS1, and 30 on the Pixel 6a. 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 commodity hardware publicly available, and can be used by others to reproduce our results. It also allows us to compare our results across different devices, as the AMS1 and Pixel 6a have very different hardware and software characteristics.

4.1.1. 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. The relevant specifications of the Pixel 6a are as follows:

- **CPU:** Google Tensor – 2x Cortex X1, 2x Cortex A76, 4x Cortex A55
- **RAM:** 6GB LPDDR5

- **Display:** 1080 x 2400 OLED
- **WiFi:** WiFi 6e
- **Bluetooth standard:** 5.2
- **Battery:** Li-Po 4410 mAh

4.2. RQ1: Energy Testing

The goal of energy testing is to identify differences in energy consumption between versions of the same software, which may indicate the presence of an "energy bug". Energy testing can be performed in a number of settings. For example, a developer might want to locally test a set of changes they made to determine whether or not they will negatively affect battery life. Energy testing may also be made part of a continuous integration pipeline, so that an organization can test release candidates automatically to find energy bugs.

In order to validate our approach with respect to energy testing, and answer RQ1, we need to compare two or more versions of an app with known differences in energy consumption. We have chosen to create our own test cases, identifying five scenarios in which we expect to see a measurable difference in energy consumption. The first three of these are a set of code smells, which we have chosen due to prior work which found that methods containing these smells consume more energy than when those smells are removed [38]. The second type of workload does not change its behavior on the CPU between versions, but instead activates various hardware modules commonly found and used on mobile devices.

4.2.1. Code smells

We chose to compare differences between "code smells" that are known to cause an increase in energy consumption, as these are well-defined and can be easily implemented in isolation. For ease of implementation and portability, we restricted our selection of code smells as follows: Firstly, due to limitations of our tool, increased energy consumption at times when no thread of the app under test is being executed will not be detected. Energy bugs such as forgotten wakelocks will, therefore, not be detected. To avoid this scenario, we limit our selection of code smells to those involving active execution of code. Secondly, we want to avoid having our validation process depend on something outside of the device under test, such as fetching data from the internet. We therefore exclude code smells that involve accessing a network. With this restrictions in mind, we identified the following three code smells from Palomba et al. [38] as being suitable to our validation process: *internal setter (IS)*, *slow for loop (FOR)*, and *member ignoring method (MIM)*. Each of these code smells has been observed to (drastically) increase the energy consumption of their containing method [38]. In the case of *MIM*, at least part of the energy consumption increase is caused by the use of dynamic binding instead of static binding[1], which increases the overhead involved in each function call. As this is a core feature of the Java language, we do not expect the differences between fixed and "smelly" versions of this code smell to have been optimized since it was observed.

In addition to validating our approach based on these code smells, we will investigate whether or not these code smells are still relevant on newer versions of Android. As Palomba et al. [38] used Android 5.1.1, some features of the Android Runtime (ART), such as just in time (JIT) compilation, were not yet implemented, and may have an effect on the energy consumption of code smells. We also want to compare the difference between debug and release builds, as the ability to run profilers on release-mode builds was not yet available in this Android version, and may also affect the energy efficiency of code, due to the reduced optimizations applied in debug mode.

4.2.2. Hardware-based workloads

In order to validate the effectiveness of our approach at detecting changes in the energy consumption of hardware aside from the CPU, we have added two workloads that are designed to induce energy consumption changes in other hardware modules.

The first of these workloads activates a Bluetooth low energy scan at defined intervals. By altering the duration and frequency of the scan, we expect to observe differences in the energy consumption of

[1]`https://www.geeksforgeeks.org/static-vs-dynamic-binding-in-java/`

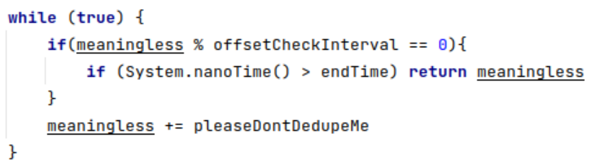
the device over a period of time. In order to make these differences visible to our tool, which requires actively running threads in order to measure energy consumption, we will run a basic cpu-based workload at regular intervals. Since this workload will be consistent across all configurations of the test, we do not expect it to influence our results.

The second of these workloads activates the accelerometer sensor on the device, registering a `SensorEventListener`[2] with the `SensorManager` to request updates from the sensor at a given frequency. To ensure that the values from the sensor are used, and that our tool is able to measure differences in energy consumption, we store the values received from the sensor in a volatile variable, and perform a number of addition operations using this data at a fixed rate. Aside from updating the stored values, no computation is done in the `SensorEventListener` callbacks, to avoid significantly increasing the amount of work done by our test when the update rate of the sensor is increased.

4.3. RQ2: Energy Debugging

Our approach to energy debugging provides the user with a list of methods, ranked by the total energy consumed during the recording. In order to test this, we first need a ground truth with which to compare our estimated ordering. As we were unable to find any recent works which performed a similar analysis, we chose to create our own ground truth using execution time as a proxy for energy consumption.

We generate our ground truth using random selection of six workload classes, each of which uses the same workload inlined from a common function, containing the loop shown in Figure 4.1.



```
while (true) {
    if(meaningless % offsetCheckInterval == 0){
        if (System.nanoTime() > endTime) return meaningless
    }
    meaningless += pleaseDontDedupeMe
}
```

Figure 4.1: Main loop of randomly selected workloads

At the beginning of the test, the random workload randomly generates a probability for each of the six classes to be chosen during each time step. 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. Algorithm 1 shows this selection methodology.

Algorithm 1 Random Test

```
procedure RandomTest($interval, total\_time, sleep\_prob$)
    while $elapsed\_time < total\_time$ do
        if randomFloat$(0, 1) < sleep\_prob$ then
            Sleep(interval)
        else
            $workload \leftarrow$ getWorkload()
            workload.workFor(interval)
        end if
    end while
end procedure
```

Each time a workload is executed, timestamps at the beginning and end of each timestep are taken and added to a total execution time log that is tracked for each workload, as well as for sleep. After the test has ended, these timestamps are printed to the system `logcat`, and collected by the *test orchestrator*. The *test orchestrator* then calculates and records the proportion of runtime taken by each of the classes, both with and without including time spent sleeping.

[2] `https://developer.android.com/reference/android/hardware/SensorEventListener`

4.3.1. Execution Time as Proxy for Energy Consumption

Relating execution time to energy consumption is controversial, and prior work can be found which both agrees [12] and disagrees [22, 8] with this relation. Corral et al. [12] 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. [22] 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 [43]. 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, 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 [8].

Our validation process most closely resembles that of Corral et al. [12]: We use a simple loop that performs only a few operations, which are expected to heavily load the CPU and no other system components. We make one call to `System.nanoTime()` within this loop, but only make this call every million loops to reduce its impact on the energy consumption of the benchmark. Further, we have inlined the same function into each of our test classes, ensuring that the workload is exactly the same regardless of which class's `work()` method is executing. We therefore have ruled out the confounding effects of network requests, display state, and processor power states, and consider execution time to be an effective proxy for energy consumption for this scenario.

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.

4.4. RQ3: Adyen POS Case Study

In addition to our empirical evaluation, in which we evaluate EDATA's performance against known software in a controlled environment, we performed a case study at Adyen, where we used EDATA to investigate an energy bug on the point of sale (POS) software shipped with our test device, the AMS1[3]. In addition to answering RQ3, we performed this case study in order to validate EDATA in a real-world scenario, to complement our empirical evaluation which used synthetic tests. In addition to the primary focus of RQ3, we will also investigate how our information is used by the developer, to determine what is most useful to the development process, and what can be improved.

In this case study, we cooperate with a developer tasked with solving an energy bug causing excessive idle battery drain, known to appear only when the Adyen POS software is installed on the device. We take an iterative approach to our case study: We analyze the Adyen POS software with EDATA, and deliver the ordered list of methods to the developer. We also assist with interpretation and contextualization of the results, if necessary. Once a change has been made, we evaluate the software and repeat the process. By comparing the debugging progress made before and after the start of our case study, we will answer RQ3: Does providing developers with an ordered list of methods ranked by energy consumption aid in identifying and fixing energy bugs?

[3]`https://www.adyen.com/pos-payments/terminals/ams1`

5 Results

In the following chapter, we evaluate each of our research questions, with specific focus on the evaluation laid out in chapter 4. Findings outside of these will be briefly noted, and discussed further in chapter 6. We will primarily focus on results gathered from the Google Pixel 6a, as this device is more relevant to the current state of the art, using a modern CPU and up-to-date Android version – Google's *Tensor* CPU and Android 13, where the AMS1 uses Android 10. Further, our results are mostly consistent between this device and our AMS1 test device. Differences between the two will be noted, and discussed where relevant.

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

In this section, we evaluate the effectiveness of EDATA in energy testing, by running multiple versions of a workload with changes introduced between the versions that either alter the characteristics of the workload or the hardware utilization of the device in a way that is known to affect energy consumption. It is important to note that these workloads are not intended to be directly compared to each other, as different amounts of work have been performed in each workload in an attempt to roughly align their release-mode runtimes. We perform similar adjustments between devices, and so cannot directly compare the energy consumption of the AMS1 to the Pixel 6a, except where noted.

5.1.1. Code Smell Tests

5.1.1.1. Internal Setter

EDATA was able to identify significant differences between the unfixed internal setter (IS) smell, in which our test case used getter and setter methods with `public` visibility, and a fixed version of the workload in both release and debug modes on both the AMS1 and Pixel 6a, with $p \ll 0.01$ in all cases. When using a `private` visibility setter, we found that, in release mode, there was no significant difference between a fixed and unfixed version of the workload on either device. In debug mode, however, the energy consumption of the test was significantly increased, by over 3.5 times on the Pixel, and just under 3x on the AMS1. This indicates that `private` setters, at least the trivial setters used by our workload, are fully optimized away in release mode, but not in debug mode.

5.1.1.2. Member Ignoring Method

As with the IS code smell, EDATA was able to identify differences between the workload containing a `public` visibility member ignoring method, and one where this method was declared as static. We have chosen not to test using a `private` visibility method, as the results are similar to those of the IS smell. On both devices, the differences were statistically significant with $p \ll 0.01$. However, we note a significant difference in the effect of the unfixed member ignoring method (MIM) smell when using release mode between the two devices. On the Pixel 6a, we find an increase in median energy consumption of about 12.6%, shown in Figure 5.1c. On the AMS1, however, we find an increase of about 94.4%, shown in Figures 5.2c and 5.2d. Additionally, there is little increase in energy consumption when using debug mode in combination with the unfixed variant, where on the Pixel there is a similar increase for both variants. This discrepancy could be caused by optimizations present in Android 13 that are not present in Android 10, but conclusively attributing a source to this phenomenon is out of scope for this thesis.

5.1.1.3. Slow For Loop

Unlike the other two code smells tested, we did not observe a significant difference between fixed and unfixed variants of the slow for loop (FOR) smell in release mode on either the Pixel or AMS1 ($p \gg 0.05$). In debug mode, however, we observed a difference between fixed and unfixed variants on both devices ($p \ll 0.01$), but this difference was in opposite directions. On the Pixel, we found a median reduction in energy usage of about 26 Joules when fixing the code smell, or about a 2.6% reduction. On the AMS1, we observed a 2.2 Joule increase - or 1.39%. We further observed the surprising result that, on the AMS1, debug mode appears to add no overhead, with the energy consumption median energy consumption being slightly lower.

When running our first iterations of the FOR test on the Pixel 6a, we noticed that the test finished much faster than expected when using the same parameters as the AMS1. Where the 'fixed' test in release mode ran for about two minutes with 3000 iterations of its outer loop, running for two minutes on the Pixel 6a required 40000 iterations - over ten times as many. We did not observe a similar speedup for other code smells, and thus interpret these results to indicate that the AMS1 is bottlenecked by something other than raw CPU speed, such as memory bandwidth, cache, etc. This could also explain the lack of overhead observed when using debug mode, as the CPU may be spending enough time waiting for some operation to complete that the extra overhead generated by debug mode does not appear in the energy consumption. We therefore are hesitant to draw strong conclusions on our primary research question from this code smell on the AMS1, but use it as a lesson learned - hardware may have unexpected effects on the energy consumption of code, and general assumptions may not hold true on all devices.

5.1.2. Hardware-Based Tests

In contrast to the prior tests, our hardware-based tests use identical parameters on both the AMS1 and Pixel test devices. We will therefore compare the two devices directly in some of these tests, and show the difference in energy consumption from one device to another. While these differences do not directly answer any of our research question, they highlight that, given a particular workload, it is essential to perform energy validation when selecting appropriate hardware.

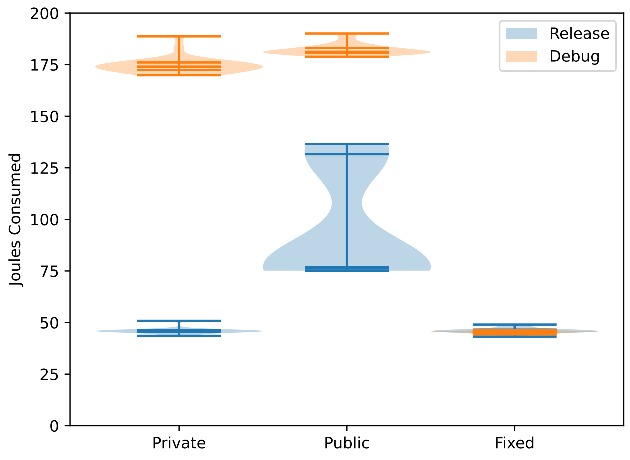
5.1.2.1. Accelerometer

We chose to compare three different update intervals in our accelerometer test: 2Hz, 20Hz, and 200Hz. EDATA was able to detect energy consumption differences between each of these tests, on both test devices, as shown in Figure 5.3. In all cases on both test devices, there was a statistically significant difference between each of the update intervals, with the closest in both cases being between 2Hz and 20Hz. On the Pixel, the median increased by about 4.4%, with a p-value of 0.0014. On the AMS1, each of our comparisons was significant with $p \ll 0.01$, and the difference in median energy consumption between 2Hz and 20Hz rates was 4.57%, very slightly higher than that of the Pixel, though with a larger absolute difference due to the overall higher energy consumption. On both test devices, the difference between 20Hz and 200Hz update rates was much more significant, with p-values $\ll 0.001$.

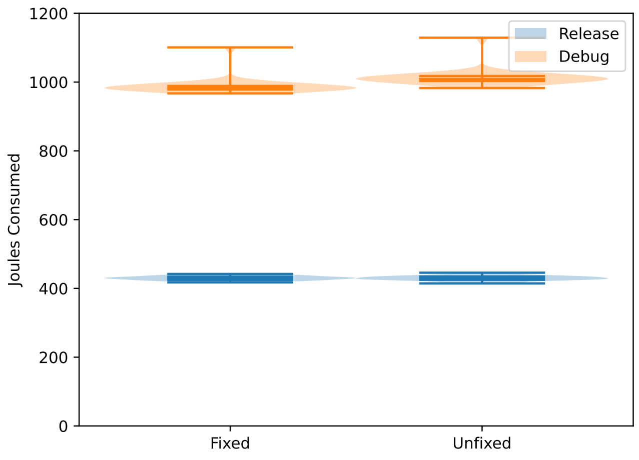
5.1.2.2. Bluetooth

We chose to compare two parameters for our Bluetooth tests: scans lasting 1000ms and 200ms, both repeating every 10 seconds. Initially, we chose to use the Random workload with a single class active and a 60% chance to sleep. This was intended to simulate a real-world app, that will not actively use the CPU 100% of the time.

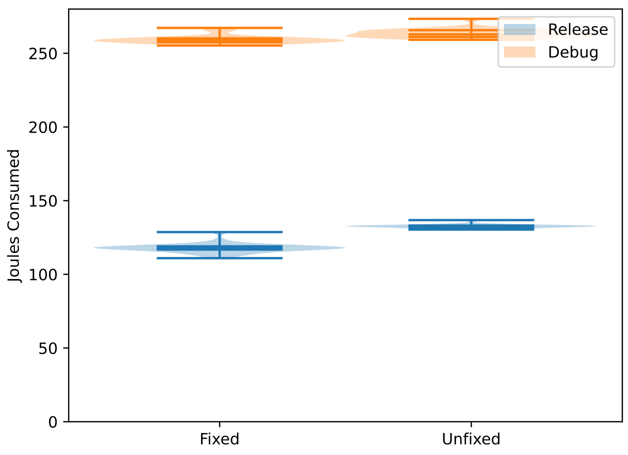
We found that, using 60% sleep, we were unable to detect the difference between the two scan lengths with statistical significance. On the Pixel, the medians of the two sets of tests differed by about two Joules, but this difference was not statistically significant (p-value: 0.455). The AMS1's results were similar, with a difference in median consumption of 2.76J, also not statistically significant (p-value: 0.337). In light of this, we decided to re-run our tests with a sleep probability of 0 to rule out



(a) Internal Setter

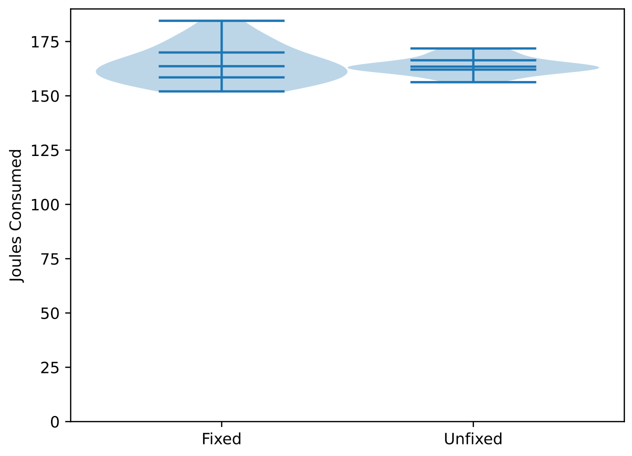


(b) Slow For Loop

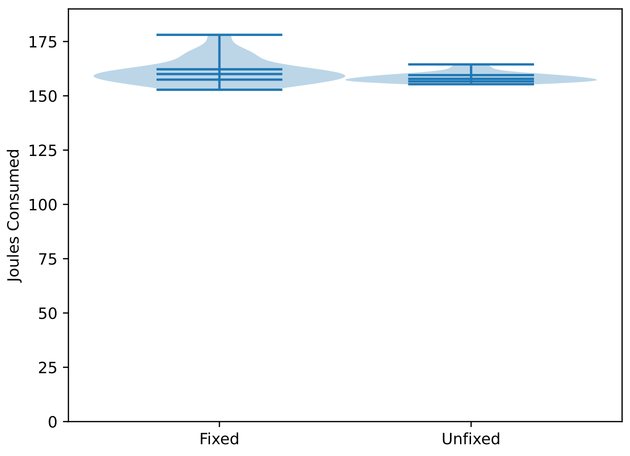


(c) Member Ignoring Method

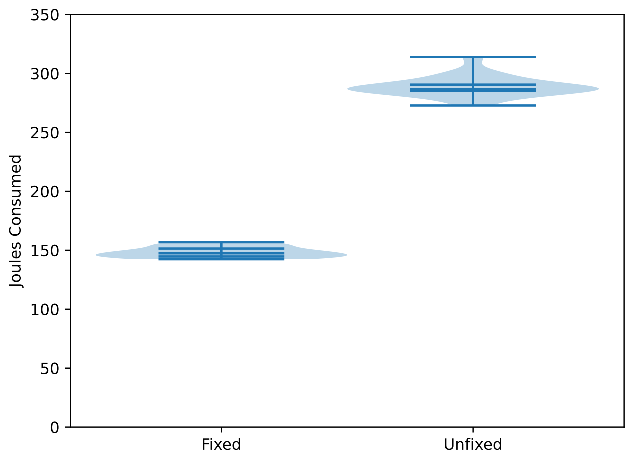
Figure 5.1: Code Smell Results - Google Pixel 6a



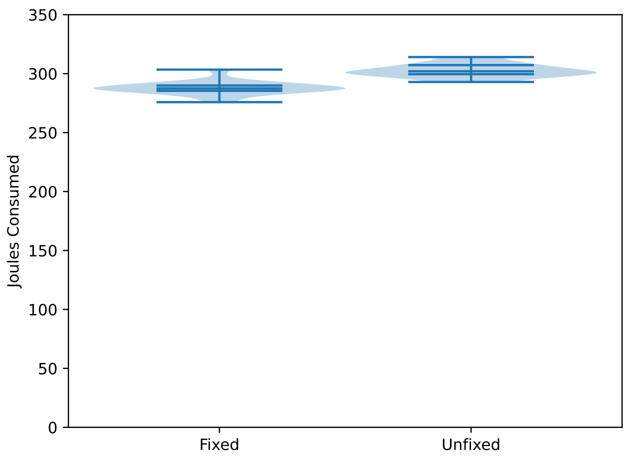
(a) Slow For Loop - Release Mode



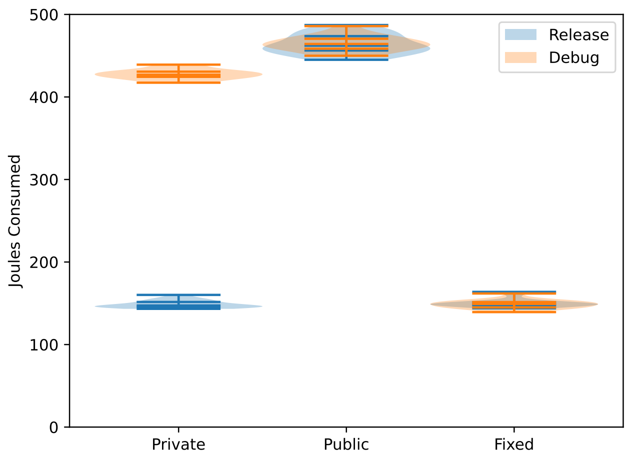
(b) Slow For Loop - Debug Mode



(c) Member Ignoring Method - Release Mode

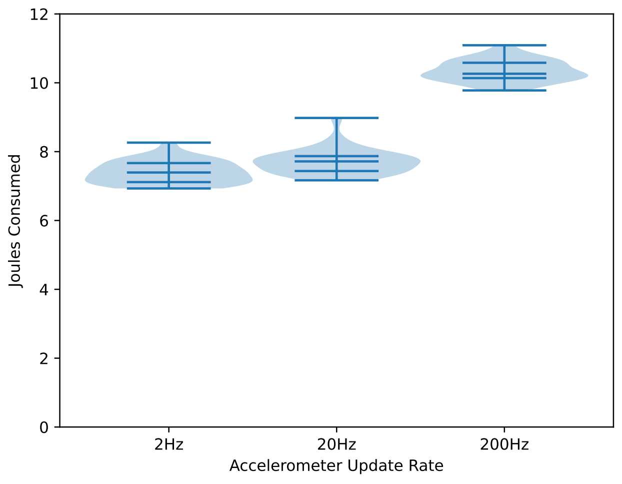


(d) Member Ignoring Method - Debug Mode

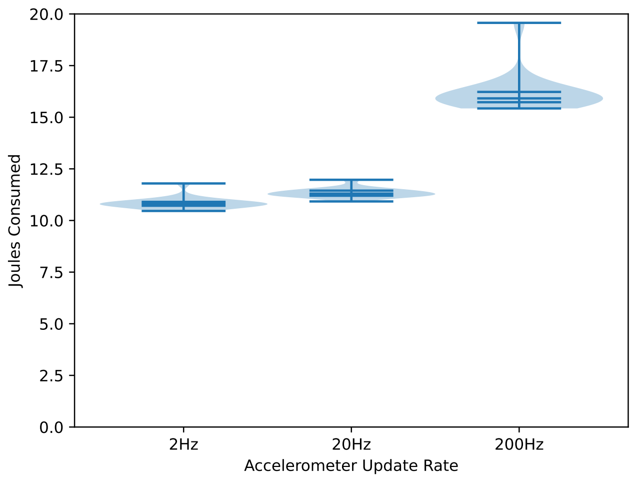


(e) Internal Setter

Figure 5.2: Code Smell Results - Adyen AMS1



(a) Pixel 6a



(b) AMS1

Figure 5.3: Accelerometer Test Energy Consumption

the possibility that the results were influenced by the device randomly sleeping during Bluetooth scans, and thus not recording the extra energy consumption. In these tests, we found that the median energy consumption of the test **decreased** by nearly 8 Joules when using a longer scan period on the Pixel, with noticeably smaller IQR and range. The result was also statistically significant, with a p-value of 0.001. The cause of this result is unclear, but could be attributed to optimizations performed by the Android system or differences in the environment, particularly in the number of Bluetooth LE devices being detected during the scans. On the AMS1, we did not repeat this result. The median energy consumption increased by 4.7J, with a p-value of 0.047. Plots of our Pixel results are shown in Figure 5.4, and our AMS1 results are shown in Figure 5.5

5.1.3. Conclusion

In the majority of our tests, EDATA is able to clearly distinguish between the energy consumption of different workloads, showing trends that align with our expectations. While the results of our evaluation of the FOR code smell and Bluetooth LE scan do not (fully) show the expected increase in energy consumption, we do not believe that these provide evidence against the efficacy of EDATA, and accurately represent the true energy consumption of our tests. We therefore 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.

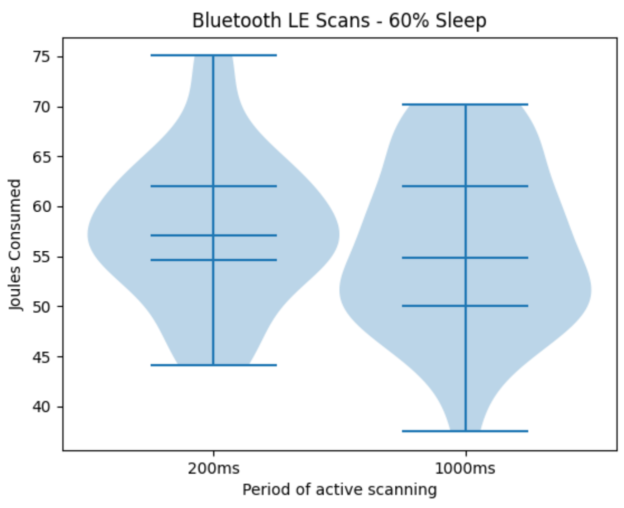
5.2. RQ2: Can we rank methods within Android apps by their energy consumption using a callstack-sampling approach?

In this section, we evaluate RQ2 using the methods described in Section 4.3. While we are primarily concerned with the ordering of the methods, and less with the precise energy consumption estimates, we use the mean magnitude of relative error (MMRE) as our metric, as a low MMRE implies that the ranked ordering of results is accurate. MMRE is used by similar works [17, 18, 34], and is considered to be a standard metric for effort estimation models, with an accepted upper threshold of 0.25 [7].

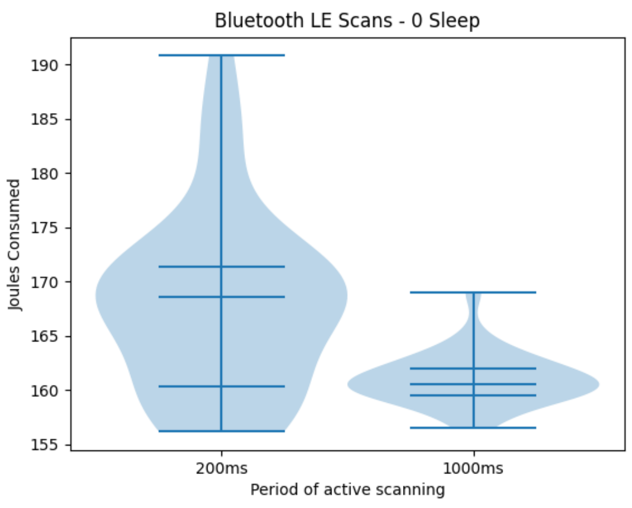
5.2.1. Selection of test parameters

We evaluate the effect of the following parameters on EDATA: Runtime, sample rate, and workload selection interval.

In order to evaluate the effect of test runtime on the accuracy of our estimates and method ordering, we selected two test lengths: two minutes and ten minutes. Due to time constraints, we chose not to evaluate all of our parameters at the ten minute length, leaving out 100ms workload selection intervals

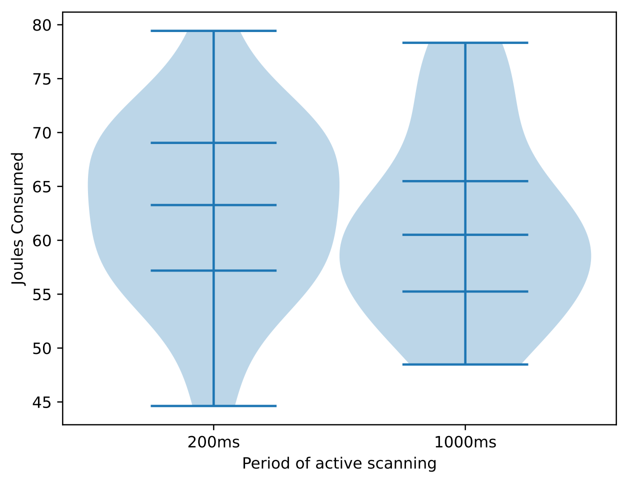


(a) 60% Sleep Chance

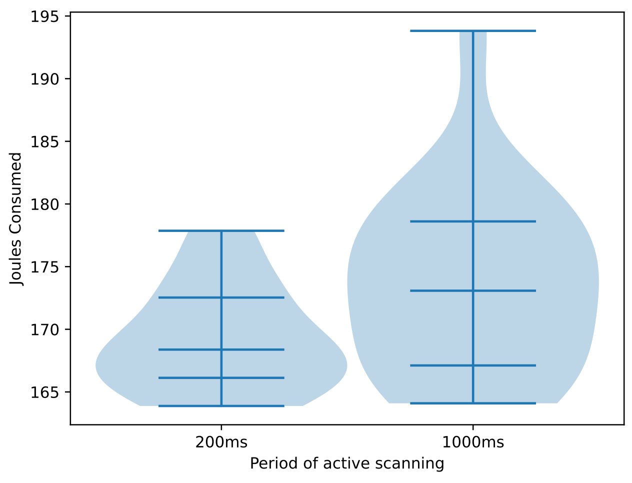


(b) 0% Sleep Chance

Figure 5.4: Bluetooth - Pixel 6a



(a) 60% Sleep Chance



(b) 0% Sleep Chance

Figure 5.5: Bluetooth - AMS1

for all sample rates except 2Hz. We do not consider this to weaken our results, as due to the high accuracy observed when using 100ms intervals with sample rates above 10Hz, there is little improvement to be found when extending the test duration.

We selected three sample rates for our evaluation: 2Hz, 10Hz, and 100Hz. We selected these sample rates for two reasons: Firstly, higher sample rates increase the amount of data collected and the overhead of collection. As EDATA is designed to be used in a production or near-production environment, we want to use sample rates that we expect would have low enough overhead to be usable in such an environment. Additionally, while increasing sample rate will make our probablistic estimates of method runtime more accurate, they will not increase the granularity of our power draw measurements. As the maximum update rate we observed on our test devices is 5Hz, even the "middle" sample rate of 10Hz will exceed the update rate of power draw. Thus, we expect diminishing returns by further increasing the sample rate. Finally, as our test uses fixed workload selection intervals, increasing the sample rate such that multiple samples occur during a single selection will have little to no effect on the results. We therefore chose to limit our test to three sample rates, all of which are relevant to the workload selection intervals chosen.

We selected two workload selection intervals: 10ms and 100ms. The workload selection intervals determine how long each workload class will run before being stopped and a new class (or sleep) is selected. We chose these two intervals as they allow us to test the following scenarios:

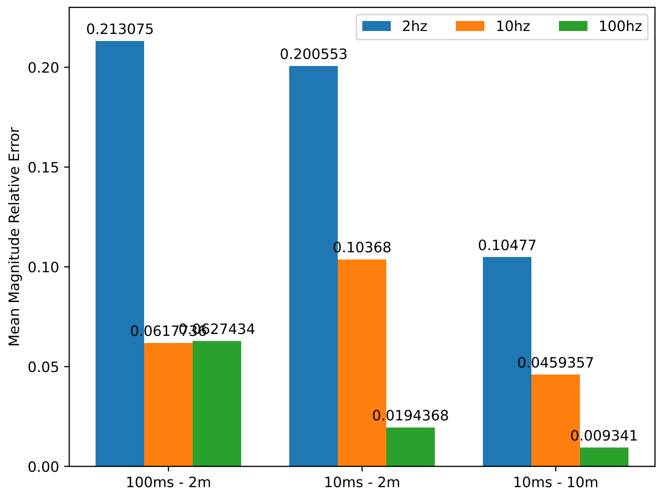
- Selection interval much faster than sample rate, with varied runtime (10ms, 2Hz)
- Selection interval matching sample rate vs selection interval shorter than sample rate, with varied runtime (100ms, 10Hz vs 10ms, 10Hz)
- Selection interval always equal or slower than sample rate, with varied runtime (100Hz sample rate)

We consider these cases to be sufficient to evaluate the effectiveness of EDATA in ordering methods.

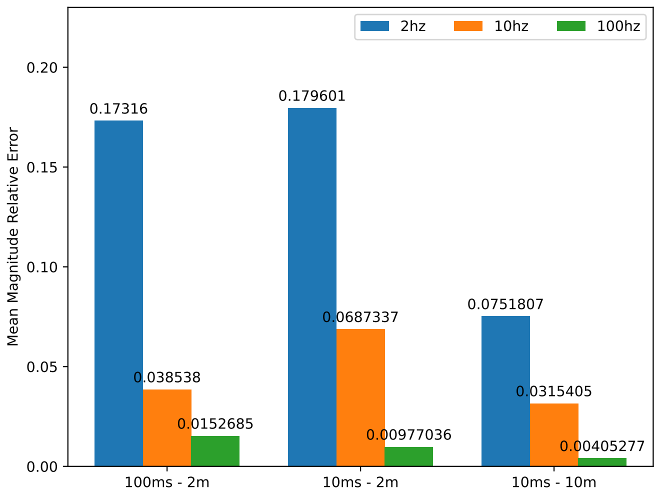
5.2.2. Results

With the Pixel test device, we found that EDATA was able to effectively order the six methods. In all of our test cases, the MMRE was low, with the highest observed difference (corresponding to the worst-case scenario tested) being about 0.203 on the Pixel test device. This remains below the upper threshold of 0.25, and as such can be considered to be sufficiently accurate. As shown in Figure 5.6a, the MMRE is highest when using a 2Hz sample rate. This is an expected result, as the lower sample rate limits the ability of EDATA to estimate the likelihood of a method being executed, and thus the energy consumption. Unexpectedly, we see that using a 100Hz sample rate with a 100ms selection interval increases the MMRE slightly compared to 10Hz. We believe this to be caused by random error, such as interference from background tasks, and in both cases the difference is small enough as to be irrelevant for practical use. By comparing the 10ms selection interval between two and ten minute runtimes, we see that the increased run-time greatly improves the performance of our method orderings, with the 2Hz sample rate over ten minutes outperforming a 10Hz sample rate over two minutes. As the total number of samples should remain (roughly) the same, with some variance due to the random chance of sleeping, these results indicate, but do not conclusively prove, that increasing test duration may be more effective than increasing sample rate, but that both options perform well.

On our AMS1 test device, our tests produced similar results to those on the Pixel, and are shown in Figure 5.6b. We note two surprising differences to our Pixel results: First, the maximum MMRE observed is 0.1796, on the 10ms-2m test. This is a reduction of about 0.02 from the MMRE observed on the same test on the Pixel device, which we expected to have a lower or equal MMRE due to its shorter current sampling interval. In addition to this, the MMRE of the 10Hz and 100Hz sample rate tests on the 100ms-2m test were significantly lower on the AMS1 than on the Pixel, with the largest difference being observed on the 100Hz sample rate, with the MMRE being only a quarter of that on the Pixel. This effect could be caused by the lower current reporting rate on the AMS1 reducing the effect of random variations in energy consumption on our measurements compared to that of the Pixel. Since we use run-time as a proxy for energy consumption in this test, random variations in power draw will likely result in an increase in error, and the lower current sample rate of the AMS1 means that a



(a) Pixel 6a MMRE



(b) AMS1 MMRE

random spike in energy consumption will probably be attributed to multiple methods in our test, reducing its effect.

5.2.3. Conclusion

Our results show that, even when using a low sample rate combined with a short run-time, the MMRE of our estimations remains under a maximum threshold of 0.25, commonly considered to be an adequate upper bound for estimation accuracy [17, 7]. We therefore 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.

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

5.3.1. Case Study Progression

Our case study focused on solving an issue identified in the Adyen software bundled with the AMS1: excessive idle energy drain causing the device to deplete its battery in under 24 hours without being used. The developer responsible for solving this issue, henceforth 'Developer A', had confirmed that the bug was caused by the Adyen software by uninstalling the app, which caused an immediate increase in battery life. Developer A had, before our case study, attempted to use Android Studio's built-in energy profiling tools, but was not able to get any meaningful data from them.

We first evaluated the Adyen software using EDATA by defining a suitable test case - in this case to leave the device idle with the display off - and performing a single hour-long energy analysis, using a 100Hz sample rate. We then filtered this list to methods in an `adyen` package to reduce our search space to those that were part of the Adyen codebase. Finally, we identified one method whose callchain consumed over 20% of the recorded energy consumption. We noted that the energy consumption recorded seemed extremely small for the amount of battery being used, at slightly over 22 Joules, where the energy consumption over an hour should have exceeded 2000 Joules. We gave this information to Developer A, who agreed that this method should not be consuming much energy during idle states, and investigated further.

After some time, we received a new build of the software from Developer A, who had removed the functionality we identified. We performed a new test using EDATA, using both a 10 minute and 1 hour runtime with a 10Hz sample rate instead of 100Hz. We found that very few samples were collected, even over the course of an hour, which indicated that the energy bug had been fixed. However, Developer A stated that this was not the case, and that, by observing the battery level over time, they found only a small improvement in battery life. After some investigation, we were able to identify the problem: using a slow sample rate caused EDATA to miss nearly all of the Adyen app's energy consumption. After performing another test using a 500Hz sample rate, we found that the energy consumption over 10 minutes was more than 100 Joules, which is much more in line with our expectations. While this does not fully account for the battery drain we observed, we expected this to be the case as EDATA will not record energy consumed by any software outside of the application under test (AUT). We further ran 10 tests of 10 minutes each at 500Hz, and found that the median energy consumption attributable to the Adyen software was 126 Joules, or approximately 756 Joules per hour. Since this is less than a third of the device's total energy consumption, and it is unlikely that we would be able to reduce this to a level that would resolve the energy bug without compromising the function of the software, we concluded that, while there were opportunities for energy efficiency improvements within the code of the Adyen software, we should focus on potential causes aside from code execution.

Our first step was to investigate the Adyen software using the Android Studio debugger. We found that, due to poor UX design in combination with several UI bugs, it was not clear how to activate the full set of energy debugging tools, and some of these tools had been missed during Developer A's initial investigation. Once activated, we found that a partial wakelock was being constantly held by the Adyen software while the device's display was off. Partial wakelocks prevent the system from entering deep sleep states, and thus have a large effect on the battery life of the device. We tested a build of the Adyen software with this wakelock disabled, and found that the energy consumption was dramatically reduced compared to our prior fix, with a median energy consumption of 12.4 Joules over 10 minutes, a 90% reduction. Due to the limitations of EDATA, we're unable to verify a reduction in energy consumed by sources outside of the Adyen software, so we added an additional test: we collected the battery charge level in percent from Android Debug Bridge (ADB) at the beginning and end of each test. This allows us to compare the overall battery life of the device in a controlled test. We tested all three versions of the Adyen software; the current master branch, our first fix, and our wakelock removal, and found that the mean percentage decrease over ten minutes was 1.4%, 1.3%, and 0.3 respectively. We further observed that the device was capable of lasting much longer, over 24 hours, before the battery would be fully depleted. We thus concluded that our second fix successfully resolved the energy bug.

5.3.2. Bug Resolution

Developer A informed us that the root cause we identified was not unintended behavior, but rather inherent to the design of the Adyen software. The fixed version of the software we tested could therefore not be used as-is, and functional changes would need to be made to remove the wakelock. During discussion of the necessary changes, we spoke with Developer B, who had been advocating for changes to this behavior for some time, but was unable to find support for their proposed changes due to a lack of quantifiable evidence of the impact the current architecture had on battery life. Once our findings were presented by Developer A, the team responsible immediately supported changing their architecture such that a permanent wakelock would no longer be necessary. The shift in dynamic was immediate, in spite of the effort required to implement the change.

5.3.3. Developer Interview

After concluding our case study, we conducted an interview with Developer A. In this interview, we asked them about how they approached the debugging process before the beginning of our case study, and what difficulties they faced when using existing tools. We then asked which features of EDATA contributed to solving the energy bug, and how they used them in the debugging process. Finally, we asked if they would use method-level energy consumption data in their development process in the future, if it were provided (for example) as part of a continuous integration pipeline.

5.3.3.1. Challenges of existing tools

Developer A was not experienced in energy debugging, and was unsure of both how to profile the energy consumption of the Adyen software, and what to look for. As the software consists of both native and java virtual machine (JVM) code, it was mandatory that an energy profiler support both, or the information collected would be incomplete. Another added complexity is networking - since the primary function of the software is to act as a point of sale (POS) device, any active use of the software would involve network communication, which can have a significant effect on energy consumption.

Therefore, in a scenario where, for example, battery life was insufficient during active use, an energy debugger would need to correctly account for energy consumed by networking. In their initial investigation, Developer A used the Android Studio profiler to collect energy data. they found the output of the profiler to have insufficient detail to assist in debugging. While they observed some spikes in the time graph of energy consumption, they lacked the context to determine whether or not these spikes were contributing to the energy bug they were attempting to fix, or if they were reasonable for the work being performed. While the Android Studio profiler, when used with debug mode, is able to obtain sampled traces from *simpleperf* similarly to EDATA, it does not use these to estimate energy consumption, providing only a time-series of abstract estimates – either 'low', 'medium', or 'high' energy consumption. In addition, while Developer A was aware of the two available profiling modes - full and limited - they were not aware of the exact differences between the two, such as the ability to monitor wakelocks in 'full' mode. Furthermore, while they attempted to activate 'full' mode in their investigation, a bug which occurs when attaching the debugger to a running process prevented this, causing the profiler to run in 'limited' mode, even though they had enabled 'full' mode.

5.3.3.2. Our Contribution

Our contribution to the resolution of the energy bug can be split into two categories: the first is contribution of EDATA; that is, the output generated from running tests on the Adyen software. The second is the contribution of the authors' knowledge in energy debugging, consisting of background knowledge on analyzing the energy consumption of Android devices and help in contextualizing the results of EDATA. In this thesis, we primarily focus on EDATA's contributions, but also reflect on our own direct contributions in order to gain insight into ways that EDATA can be improved such that developers of all experience levels can more effectively use its output.

We asked Developer A a number of questions regarding the usefulness of EDATA's output compared to the Android Studio profiler, as well as which components of the output were most useful. they found the ranked list of methods to be very helpful, as it allowed them to see immediately which parts of the software were responsible for the most energy consumption. they also found the amount of Joules consumed by each method useful, as it allowed them to gain a more thorough understanding of the software's energy consumption. However, they mentioned that the interpretation of the results we provided was more helpful than the raw results output by EDATA. One of the primary reasons for this is the lack of context included with the output: while it is straightforward to compare the Joules consumed between different methods, they did not know what this meant in terms of battery capacity, which is a less precise, but more relatable method. Since a conversion from Joules to battery percentage is easy to calculate given the total capacity of the battery, future versions of EDATA will include the ability to show the percent of the battery consumed in its results, so that developers do not need to perform this conversion themselves. they also mentioned that while the ranked list of methods did not solve this particular energy bug, it highlighted some areas for improvement in the Adyen software, which will be investigated further in the future.

While the root cause of the energy bug - the permanent wakelock - was not something that fell under the scope of EDATA's output, the information provided by EDATA in combination with our interpretation significantly accelerated the debugging process, and allowed them to find the cause much faster than would have otherwise been the case. Further, the results provided by EDATA gave them quantitative metrics that could be used in making a business case for the required changes to the Adyen software, something that was not possible with the existing output of the Android Studio profiler.

Finally, we asked Developer A if they would use the results of EDATA if it were implemented in an automated testing system, such as the existing robots used to test Adyen software on physical devices. They thought that this would be useful in general, but also thought that there was room for improvement, particularly regarding the tracking of energy consumed by hardware aside from the CPU. This would better align with the current efforts at Adyen to improve energy efficiency focus on WiFi and Cellular energy consumption, which are not fully captured by EDATA's measurements.

5.3.4. Conclusion

While the energy bug identified in our case study was not covered by the scope of EDATA, EDATA substantially contributed to its identification by ruling out code execution within the Adyen software as the primary cause. In addition to this, several opportunities for energy efficiency improvement have been identified, and have been put on a roadmap for future investigation. We have also gained valuable insight into the importance of detailed energy consumption data to the software development process, and how stakeholders are willing to incorporate such data into their decision-making. We therefore answer **yes** to RQ3, concluding that providing developers with an ordered list of methods ranked by energy consumption significantly assists in identifying and fixing energy bugs. Our findings also provide support for the generality of our assessment of RQ2, as the Adyen POS software is a real-world app, with many features depending on hardware other than the CPU, such as networking and card reader hardware.

6 Discussion

In the following section, we compare the benefits and drawbacks of three commonly used methods of attributing energy consumption to source code, the rationale behind our choice, and what it means for the future development of EDATA. We also discuss challenges we faced during this thesis, findings that are not directly related to our research questions, and their relevance to future work. Finally, we suggest future work to be done in the field of Android energy consumption estimation, and discuss the limitations of EDATA and threats to the validity of our results.

6.1. Challenges

6.1.1. Android Platform

One of the primary challenges inherent to the Android platform is its complex runtime. Many existing energy profilers for desktop and server environments are designed for use with pre-compiled code of one language, such as Java or C++. Android apps are more complex, and may contain both java virtual machine (JVM) and native code, and JVM code is typically just in time (JIT) compiled on-device. Energy consumption tooling targeting Android must therefore take these characteristics into account, as significant changes to the runtime environment such as disabling JIT or requiring debug mode may cause unexpected changes in the energy consumption of the application under test (AUT), and thus reduce the accuracy of the tool in ways that are difficult to quantify.

We were able to leverage the existing Android platform tool *simpleperf*, which already has the ability to obtain stack traces from native and JVM code, and in the case of JVM code, can handle pre-compiled, interpreted, and JIT compiled code. This significantly eased development of EDATA by allowing us to obtain stack samples "automatically", instead of requiring us to manually implement techniques for sampling each type of code. However, there are still some limitations involved with the use of *simpleperf*: We observed that, depending on the test being run, the function or method name would sometimes be mangled in EDATA 's output, due to, for example, the need to merge callchains from JIT compiled and interpreted code together. While *simpleperf* is able to handle this, the output needs to be improved. JIT compiled code also does not contain debug information mapping instructions to lines, which was a factor in our decision to estimate energy at the method-level instead of line-level. Simpleperf also does not support line-level estimations of JVM code regardless of compilation method, although it does support this for native code.

Another challenge posed by the Android platform is the strict permissions model imposed by the system. As we developed EDATA to be usable by developers without the need to significantly change their development process, we wanted to avoid requiring root access, as few developers will have rooted devices available. This entails the following limitations:

- Android versions 10 and up are able to profile apps in release mode using the `profileable` flag[1], but versions <10 must use debug mode or root.

[1]`https://developer.android.com/guide/topics/manifest/profileable-element`

- Only one app at a time may be profiled by a single simpleperf process, even when all apps are `profileable`. This limitation is due to the way in which *simpleperf* profiles apps, using the `run-as` capability to run as the user account associated with an app, granting it the ability to profile the app's process. Since it cannot run as multiple users at once, it is thus unable to profile more than one app per instance, requiring the use of multiple *simpleperf* instances, which may bottleneck the profiling.

The strict permissions imposed by Android also affect our data collection. Initially, we wanted to control *simpleperf* by issuing commands from a companion app; This app could then be controlled remotely, removing the dependency on Android Debug Bridge (ADB). However, we found that even apps installed as 'system' apps, running under the `system` user, are unable to invoke *simpleperf* on other apps, even if the other app is considered profileable. Avoiding this limitation would require changes to the Android system image to grant this access to the system (or other) user, or a rooted device. Apps are able to invoke *simpleperf* on themselves, but only when previously enabled through ADB[2]. Faced with this, we chose to make our companion app solely collect environmental data. We investigated the logs produced by Android's `bugreport` and `batterystats`, but were unable to find a way to record the battery current, with only the voltage and capacity (if available) appearing in the logs.

Finally, we also observed that many devices do not correctly implement the `BatteryManager` API, returning current values that do not meet the specification. The specification requires that current be reported in μA, with a positive value indicating current flowing into the battery and a negative value indicating current flowing out of the battery [5]. Out of our tested devices, including several devices not included in the evaluation of EDATA, only the Google Pixel 6a correctly implemented this API, with each other device either reporting current using the wrong unit, flipping the sign, or a combination of the two. Due to these mistakes, it is critical that any tool making use of the `BatteryManager` API to obtain current measurements correct these values, using either an automatically detected or manually entered multiplier. We did not find any device that required a more complex multiplier than a positive or negative power of 10, but as our investigation was by no means exhaustive, a given device's current outputs should be verified before use. EDATA solved this problem by collecting data from the device as reported, and allows use of a divider during data analysis to convert the reported value into amps.

6.1.2. Sample Timing

Ideally, EDATA would be able to sample each active thread of the AUT simultaneously, using a fixed wall-clock time as the sample interval. We were unable to implement this behavior using *simpleperf*, due to limitations of the events available. While the `cpu-clock` event should perform as expected, we found that it does not function correctly, instead behaving nearly identically to `task-clock` [30]. As hardware-based events such as `cpu-cycles` do not have a 1:1 relationship with wall-clock time, we opted instead to use `task-clock`. This has the effect of preventing simultaneous samples of threads, as each thread's timer is tracked independently. Additionally, threads with runtime shorter than the sample interval over a given test will not be sampled at all, causing their consumed energy to be lost.

6.1.3. Core Scheduling

A challenge mostly specific to mobile platforms, but with the introduction of Intel's Alder Lake[3] also relevant on desktop and laptop computers, is the use of heterogeneous CPUs. Commonly referred to as `big.LITTLE`, though this is a specific architecture developed by ARM, heterogeneous CPUs in the context of consumer systems contain multiple types of cores, with the "big" cores typically having higher performance at the cost of higher energy consumption, with the "little" cores the opposite. Some CPUs, as is the case with the Pixel 6a's Tensor, contain more than two types of core, with the Tensor having three different types. On Android devices, apps typically do not control which CPU core they're placed on, but this can have significant effects on their energy consumption, and the most efficient core may vary by workload characteristics [26]. As discussed in Section 6.2.2, we observed effects potentially caused by the use of different CPU cores between tests, where running our evaluation on a core with a higher power draw led to a decrease in total energy consumption. We consider solving this problem to

[2]`https://android.googlesource.com/platform/systecm/extras/+/master/simpleperf/doc/android_application_profiling.md#Control-recording-in-application-code`

[3]`https://en.wikipedia.org/wiki/Alder_Lake`

be out of scope for this thesis, but future work should take into account the potential for core selections made by the CPU scheduler to affect energy consumption.

6.1.4. Baseline for Energy Consumption Estimates

During our evaluation, we were faced with the lack of a baseline with which to compare our method-level estimates to. While we were able to work around this, our workaround came at the cost of generality, as it depends on the use of run-time as a proxy for energy consumption, and may not transfer to different workloads. While there are some existing approaches capable of producing accurate method-level energy consumption estimates on Android, these approaches require expensive hardware power meters, are outdated, require the use of debug mode, or have been discontinued (in the case of *Trepn*). There is a clear need for a baseline with which researchers can evaluate novel approaches, and given our findings in 6.4, the process used to generate the baseline would ideally be easily repeatable on any Android device, such that researchers can generate a new baseline for their specific hardware/software configuration.

6.2. Empirical Evaluation

In addition to answering our primary research questions, we compared our two test devices, a Google Pixel 6a and an Adyen AMS1, as well as debug and release mode.

6.2.1. Build Mode

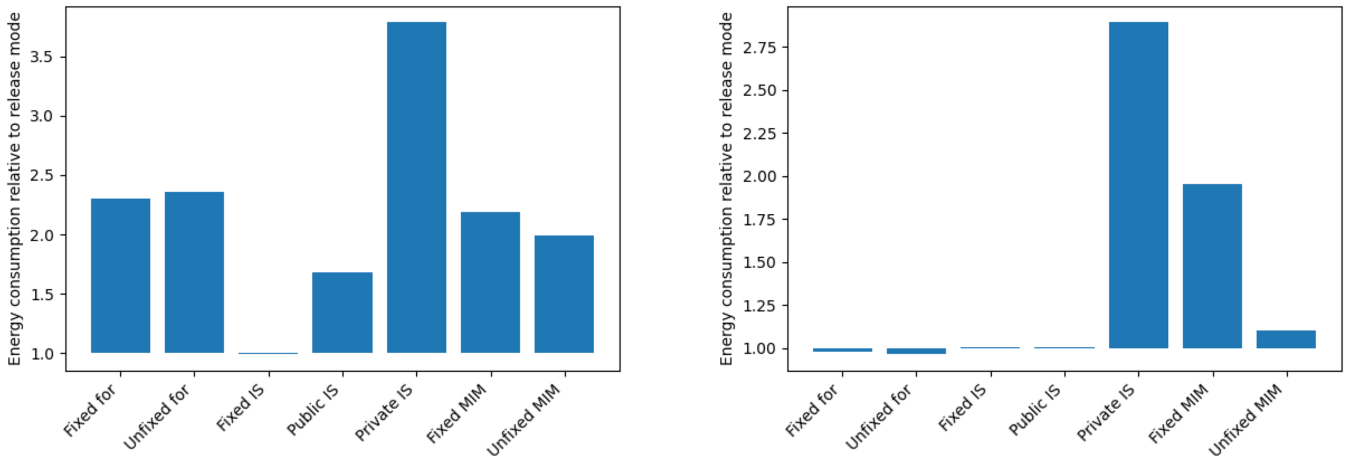
We observed that the overhead incurred by using debug mode is inconsistent not only between different code smells, but also between our two test devices. Figure 6.1a shows the relative energy consumption for each of our fixed and unfixed code smells when run on debug mode instead of release mode. The slow for loop (FOR) code smell incurs an overhead of about 2.3x for both fixed and unfixed, though with slightly higher overhead for unfixed. We observe similar results for the member ignoring method (MIM) smell, though in this case, the unfixed version has a slightly lower overhead. The most notable results are those of the internal setter (IS) smell, where we tested a version with a `private` getter/setter in addition to a fixed and unfixed (`public` visibility) version. We found that there was no difference in energy consumption for the fixed version, and a moderate increase for the public version. The largest difference is seen in the `private` version, where the energy consumption is increased by over 3.5 times. This is attributable to the same effect discussed in Section 5.1.1, which we attribute to a class of optimization being entirely disabled in debug mode.

On the AMS1, however, we found that the overhead was, in most cases, much different to the overhead on the Pixel. As shown in Figure 6.1b we observed similar overhead in the `private` IS and fixed MIM code smells, but the rest of the smells had little to no overhead, with the unfixed FOR code smell showing a very small reduction in median energy consumption.

Identifying the root cause of these differences is out of the scope of this thesis, and these results should be considered to be strictly preliminary. Comparing our test devices was not the primary focus of this thesis, and as such, we have not designed our experiment with a comparison in mind. We further note that there are significant differences between the devices which we are unable to separate. One of the most apparent differences is the Android version, with the AMS1 running Android 10 and the Pixel running Android 13. Improvements to the Android Runtime (ART) between the versions could lead to, for example, improved release mode optimizations that are not carried over to debug mode. Another difference is the CPU found in each of the devices, as differences in hardware, such as cache size, memory, and lithography can alter the energy characteristics of the device.

6.2.2. Bimodal Distribution of IS Code Smell

In the IS tests performed on the Pixel 6a, we observed that the release mode test using a `public` internal setter had a bimodal distribution of energy consumption as shown in Figure 5.1a. While this can be caused by a number of factors, such as accidentally leaving the display on or poor WiFi strength, we also observed a significant decrease in run-time in the set of tests with a lower energy consumption, even using the same test parameters. Our initial assumption was that we had performed the wrong test by mistake, but the resulting list of functions showed that the correct test had indeed been performed. In addition, we could see that the average power draw over the course of the test had increased by about 0.4W, from approximately 0.8W to 1.2W. We believe that these differences were caused by



(a) Pixel 6a (b) AMS1

Figure 6.1: Overhead Incurred by Debug Mode

Android's CPU scheduler using a different CPU core from one test to the next - trading increased power draw for performance. In our test, this resulted in reduced energy consumption, as the extra performance significantly outweighed the extra power draw. Unfortunately, we did not collect CPU core affinity information in our tests, so we are unable to say conclusively that CPU core selection caused this result. However, we believe that this is the case, and that this result shows one of the pitfalls of performing energy consumption tests on mobile devices: a lack of control over CPU core affinity.

6.3. Case Study

As discussed in Section 5.3, EDATA was able to significantly ease the process of finding and fixing an energy bug in the Adyen point of sale (POS) software. This can be attributed in large part to the detailed information provided, which allowed us and the developer to immediately see which parts of the app consumed the most energy in our test scenario. A lack of specific information is a known barrier to adoption of developer assistance tools [25], and the information provided by EDATA gives it a key advantage over commonly used tools, such as the Android Studio profiler.

6.3.1. Sample Rate

We performed the initial exploratory tests in our case study using a sample rate of 100Hz – meaning that each thread would be sampled after every 10ms. The results we obtained from these tests showed a lower total energy consumption than expected, given the total battery usage, but were still reasonable. After the first fix had been created, we reduced our sample rate to 10Hz, or a period of 100ms, as 10Hz was used in most of the tests in our empirical evaluation. With this sample rate, we found that very few, if any, samples were taken during our test scenario. To verify this, we tested on an "unfixed" version of the app, which yielded the same result. We suspected that the cause of this was a too-low sample rate, and repeated the test with a rate of 500Hz. This appeared to solve the problem, and our output was similar to that found when using a 100Hz sample rate. As we use a per-thread CPU-time timer to trigger samples, we did not expect to see this effect when reducing our sample rate, as many methods had a total run-time (when combining local and non-local run-time) of multiple seconds, much more than the 100ms minimum time to be sampled when using a 10Hz sample rate.

One potential cause is that real-world apps generally interact with threads in a more complex way than the test workloads we created for our empirical evaluation. If, for example, a task is switched between many threads in a thread pool, no one thread may exceed the sample threshold if it is set too high. Alternatively, if an app frequently creates threads for a single short-lived task that are then destroyed, they may also not be sampled. This limitation is inherent to the `task-clock` event we chose to use with EDATA. Since this event is tracked separately for each thread, each new thread must run for a full sample period before being sampled, which can lead to threads being ignored entirely and reduce the randomness of our sampling approach. The use of an event tracked globally, and not per-thread, may alleviate this problem. Future work should investigate the suitability of other events, or whether modifications must be made to allow sampling on a wall-clock timer.

When using our methodology, care must be taken to ensure that the sample rate is high enough to capture sufficient data, particularly in scenarios where many short-lived threads are used. A given sample rate can be verified by comparing the measured total run-time of the same test scenario using different sample rates. If the lower sample rate has a significantly smaller run-time, it is likely that the sample rate was too low.

6.3.2. Presentation and Contextualization of Results

It is critical for developers to be able to contextualize the output of EDATA in order to understand the impact that a particular method or functionality has on the overall energy consumption of their app. While, from the perspective of an energy expert, listing the Joules consumed by a method is clear, a mobile developer will likely be unable to (without significant practice) be able to relate this to the percentage of battery consumed by that method. In our case study, we contextualized our results in this manner for Developer A to more easily understand them, but such a calculation is trivial to implement automatically as long as the capacity of the test device's battery is known. Contextualizing energy consumption to the battery of a device allows it to be compared with known battery drain, and can help developers make informed choices on whether changes to their software are worth the potential impact.

Another weakness identified by our case study is the lack of call-chain information presented in our output. While this did not significantly impact our case study, developers analyzing a function called in different locations will not be able to distinguish energy characteristics between those locations, and thus will lack information on which call-chain consumes the most energy. One method of displaying this information is the flame graph[4], which is commonly used, including in Google's *simpleperf* tooling, to display information relating to call-chains. Presenting our data in a flame graph would be familiar to many developers, and further help contextualize the energy consumption of their app.

6.3.3. Energy Efficiency in the Development Process

One of the common threads between our interview with Developer A and informal conversations with other mobile developers at Adyen was that, while energy efficiency (and thus battery life) is a concern among developers, there is a lack of knowledge available on how architectural decisions affect energy consumption. Thus, energy consumption, while important, is not considered as a first-class citizen in the decision-making process. Once made aware of the effect of their software on the energy consumption of their hardware, immediate action was taken to correct problems, even at the cost of making non-trivial architectural changes. Notably, the data we provided succeeded in convincing stakeholders that these changes were worth the effort, something Developer B was unable to do. Grosskop and Visser wrote in 2013 that many stakeholders have a low level of awareness of the importance of software to energy consumption [20]. Despite improvements in the field of energy profiling since then, we conclude that the tools currently used by Android developers do not provide sufficient information to elevate energy consumption to be a first-class concern in the software development process. However, we are encouraged by the willingness of stakeholders to adopt energy consumption information into their decision making process, which shows that a lack of information is a primary factor preventing the wider adoption of energy efficiency into the development process.

6.3.4. Beyond EDATA: The Future of Energy-Aware Development at Adyen

While our case study has lead to an immediate boost in developer awareness of effects of their code on the energy consumption of the devices it runs on, it is only the first step towards creating a culture of energy-aware development. Developers' receptiveness to incorporating energy consumption into their decision-making, given sufficient information and context, a potential next step could be to establish an internal energy advocate, responsible for making developers aware of the energy consumption of their code and how it effects the products they create. By doing so, energy efficiency is elevated to the same level as other code quality metrics, many of which are established in industry as critical to the

[4]`https://www.brendangregg.com/flamegraphs.html`

development process.

6.4. Implications

6.4.1. Measuring Energy Consumption

The results of our empirical evaluation and the difficulties we faced during development of EDATA have a number of implications for future work concerning the energy consumption of Android devices. To our knowledge, no prior work has explored the energy overhead caused by debug mode on Android, partially owing to the past requirement that apps be set to debug mode in order to be profiled. Our findings in Section 6.2.1 show that this overhead is inconsistent between both different code smells and different test devices. This implies that the relative energy consumption of different methods in debug mode may not be equal to release mode, and in the some cases could lead to wasted development time, as developers would be misled by the results of a tool using debug mode. Some tools, such as the Android Studio profiler, already warn users that timing information found using debug mode should not be trusted. Now that release-mode profiling is supported by Android, we argue that it should become the standard for energy consumption profiling, and tools should only use debug mode if absolutely necessary.

These findings also show that results from one device cannot be assumed to transfer to another device. We do not attempt to conclusively assign a cause to the differences observed between our two test devices, and the differences between them would make this a difficult task. Even so, the clear differences in the results of our code smell tests show the need for up-to-date information on energy consumption, to keep up with both advances in hardware and software. Developers and researchers profiling energy consumption should also be aware of the potential for unexpected differences between hardware, and in cases where hardware specifications are known at development time, it is essential to test on that hardware.

6.4.2. Test Environment Setup

Our findings have additional implications with regard to methodology used in preparation of the test environment, both in a research setting and in real-world use. In Section 6.2.2, we observed that one of our tests appears to be affected by differing CPU scheduling strategies, where the energy consumption of the test strongly depends on the CPU core it is executed on. In addition, with the introduction of heterogeneous CPUs to laptop and desktop platforms, the potential confounding effect of CPU scheduling is no longer exclusively a concern of mobile devices. Allowing the CPU scheduler to function as intended can lead to inconsistent results, yet is necessary to obtain a complete image of how the application under test will behave in a production environment. We suggest that future work should take into account the presence of heterogeneous CPUs, and make a conscious decision on whether or not to allow the system to determine which cores are used to execute an AUT. If the system is allowed control, data on which cores were used at which times should be recorded in order to understand the influence that different scheduling choices have on the energy consumption of the test. We further suggest that experiments intended to measure typical energy use of software, such as a developer profiling their app or an energy regression test, should allow the regular behavior of the system. Experiments designed to be repeatable and consistent, such as our validation process, may want to restrict this behavior.

6.4.3. Developer Awareness

Our case study has shown that, while there is a willingness among developers to incorporate energy consumption data into their development process, including for larger architectural decisions, there have been few breakthroughs over the last decade in providing this information. While the academic space has seen significant research into fine-grained energy profiling on mobile devices, tools used in practice are still restricted to coarse grained output, which is of limited use to developers. Future work should take developers' needs into account, recognizing that a trade-off between precision and ease of use may be beneficial. Approaches using static analysis or machine learning to provide immediate energy consumption estimates without the need for time consuming tests on real hardware may prove useful in this regard, although care must be taken to ensure that the output is valid between different operating system versions and devices. Our results indicate that energy characteristics of software may vary greatly between devices, and failing to take this into account may lead to inaccurate estimations.

6.5. Limitations

While there are many ways in which EDATA can be improved, there are some essential limitations of our chosen approach that must be kept in mind. At its core, 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 as discussed in Section 2.3.2 may be appropriate, as such an approach typically only requires access to source code, given access to a pre-trained model.

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 Sections 2.1.5 and 2.1.6, 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.

6.6. Future Work

Our comparison in Section 2.3 highlights some of the strengths and weaknesses of EDATA, and our decision to use callstack-sampling through *simpleperf* instead of the more common approach using instrumentation. We had two primary motivations for the use of statistical sampling: The first of these is to limit overhead to be both predictable and low, thereby reducing the influence of data collection on the energy consumption of the device. Second, one of our initial design goals when creating EDATA was to create a tool with sufficiently low overhead as to run on production devices, so that real-world usage data could be obtained. Statistical sampling is the best choice for this use-case, as the sampling rate can be lowered until both the overhead and size of collected data is sufficiently reduced. This goal also led us not to pursue a static analysis based approach, as collecting data from end-user devices would allow developers to have automatically up-to-date information on a large range of devices without requiring pre-training a model, and would be less likely to miss changes in energy consumption caused by outside influences, such as changes to the Android Runtime.

In light of these choices and design goals, we have identified several areas where future work could improve the user experience and energy consumption attribution quality of EDATA, and move it further towards becoming an SDK that could be integrated into production builds of Android apps, similarly to analytics services such as *Google Analytics*[5].

6.6.1. User Experience

There are several areas for improvement in the user experience of EDATA, involving both the testing process and the analysis and display of results.

First, moving away from the current command line + settings file configuration system towards a GUI, either standalone or IDE-integrated, would significantly increase the usability of EDATA and reduce the chance of incorrect test configurations. Data analysis can also be improved through the addition of data visualization methods, such as flame graphs. *Simpleperf* in the Android Native Development Kit (NDK) is bundled with tools to create flame graphs from its output, although these flame graphs do not include energy consumption estimations. Integration of such a visualization into EDATA could, for example, allow the visualization of the energy consumption of specific callchains, even if there is overlap between them. The current output format does not allow this, as metrics are reported only per-method, even though the necessary data is collected during the measurement phase.

6.6.2. Estimation Quality

6.6.2.1. Hardware Energy Accounting

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. While EDATA does not implement any specific accounting methodology for hardware components, such as WiFi, cellular modem, etc,

[5]`https://firebase.google.com/docs/analytics`

prior work has been done to attribute energy consumption to hardware [11]. Further, since the Pixel 6, Google's Pixel devices ship with sensors that report the power draw of individual hardware components [2], entirely 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. Android devices are required to ship with a `power_profile.xml` file[6] containing estimations of the amount of current consumed by different hardware components at various states, which can be combined with state information obtained during recording to approximate the energy consumption of different components on devices that do not support individual power rail measurements.

While accounting for energy spent on hardware is an important improvement, attributing this energy at the method level is critical to informing developers of how their choices impact the energy consumption of their app. Attribution of hardware energy consumption is not trivial, as 'tail energy' must be taken into account – the extra energy consumed by powering on a component after its primary task has been completed. As mentioned by Li et al. [28], this energy is often consumed long after the method which powered on the component has finished, and naively attributing this energy to the actively running method – as EDATA currently does – may be incorrect. Li et al [28], Cornet and Gopalan [11], and Pathak et al. [39] attribute hardware energy consumption to methods, though their approaches require high-overhead method tracing. Future work should develop an approach to perform this attribution without the high overhead costs of instrumentation, and make use of the new 'power rails' available on some devices. Hardware manufacturers should provide profileable power rails, so that developers can make use of these features regardless of the device they choose to test on.

6.6.2.2. Simpleperf Events

Simpleperf supports a number of different events that can be monitored to trigger a sample. We chose to use `task-clock`, but this event has several limitations, which we discuss in Sections 6.1.2 and 6.7.1. Future work should further investigate the suitability of this event, and whether another event, such as `cpu-cycles`, would be more effective in this use-case.

6.6.3. Clustering

Initially, we planned to implement a "clustering" technique, where EDATA would take the runtime environment, such as the signal strength, display brightness, and other energy-influencing factors into account when displaying data. This would allow data collected in uncontrolled scenarios, such as from end-user devices, to be more accurately compared so that differences in energy consumption can be identified. Such a technique could be used in combination with attribution of energy consumption to hardware components, but this is not necessarily required, as clustering would make individual traces more directly comparable, reducing the need for separating the energy consumption of hardware components. Clustering may also improve the reliability of energy consumption regression tests by detecting changes in the environment, such as WiFi signal strength or background tasks, that cause an increase in energy consumption. By reducing false positives, automated energy testing as part of a continuous integration pipeline will be more attractive to development teams.

6.6.4. Platform Knowledge

Our results show that there are significant, measurable differences between the effects of debug mode on different code smells, and across different devices. Much of the existing literature on the energy consumption of code smells on Android analyzes apps that have been compiled in debug mode, or does not define which mode was used. Our observations imply that the results found when using debug mode are not guaranteed to transfer to modern devices running apps in release mode, and reporting the build of the app under test is critical to the interpretation of results. Additional research is needed to confirm our observations, understand where these differences originate from, and determine whether the current understanding of how software metrics such as code smells relate to energy consumption transfers to release mode builds.

[6]`https://source.android.com/docs/core/power/values`

6.7. Threats to Validity

6.7.1. Internal Validity

We identified several potential threats to the internal validity of EDATA's results, as well as our evaluation.

The first of these is the potential confounding effects of software other than the AUT running on our test devices. While we attempted to reduce this influence on the Pixel test device by resetting it, there is no way to fully remove the possibility of outside influence on our results. We combat this effect by performing each test at least 20 times, so that we can identify outliers and take these into account.

Another potential threat is the sampling method we chose to use, with which we have identified two issues. First, as discussed in Sections 3.2.1.3 and 6.3.1, it requires a minimum threshold of run-time before a thread's activity is sampled. Threads with short lifespans or methods called at thread start may fall completely out of view. We observed something similar to the first of these problems in our case study, where a low sample rate resulted in very few samples being made during our tests, which we resolved by greatly increasing the sample rate. 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. Future work should investigate further, and evaluate which of *simpleperf*'s events provides the most accurate results.

6.7.2. 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. Future work should perform more real-world evaluations of EDATA, such that its effectiveness over a variety of apps and devices can be evaluated.

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. An example of this is Developer A's relative inexperience with energy debugging. A developer with significant experience may find the output of EDATA less useful in comparison, as they may have existing workflows that provide similar information, and domain knowledge with which to interpret and contextualize it.

7 Conclusion

Energy consumption on mobile devices is an important to users and developers alike, and significant research has been done into measuring the energy consumption of mobile software, as well as investigating how code can affect energy consumption. While there exist a number of tools for Android to analyze apps' energy consumption, much prior work was performed on older Android versions where use of debug mode was mandatory, and many used high-overhead methods of obtaining program trace information, such as method instrumentation. With this in mind, our goal was to find an approach with low enough overhead that it could be used on end-user devices, or as part of a continuous integration pipeline with little impact to existing tests.

To achieve this goal, we developed EDATA, or Energy Testing And Debugging for Android. EDATA uses statistical sampling to analyze the an Android app and provide method-level estimations of its energy consumption. In contrast to high-overhead methods such as method instrumentation, sampling can be performed on Android devices with low overhead, and does not require any modifications to an app's compilation process, working with native code, as well as interpreted, ahead of time (AOT)-compiled, and just in time (JIT)-compiled Java/Kotlin code. This allows developers to easily use EDATA as part of their development process, as only a flag must be set in the Android manifest of their app to allow analysis.

We validated our approach both empirically and with a case study on real-world Adyen point of sale (POS) software. In our empirical analysis, we found that by using statistical sampling in combination with the on-device current and voltage sensors, we were able to both accurately estimate the energy consumption of individual methods, and distinguish between different versions of a test case where changes had been made to alter its energy consumption.

As part of our validation process, we investigated several code smells known to increase energy consumption on Android: internal setter (IS), slow for loop (FOR), and member ignoring method (MIM). [38]. We compared their fixed and unfixed versions in both release and debug mode, as a preliminary investigation into the effect of debug mode on the energy consumption of code smells. We found that each of these code smells caused a statistically significant increase in energy consumption in debug mode, with IS and MIM similarly increasing energy consumption in release mode. Notably, we found that the overhead of debug mode is not consistent between code smells or devices, with different code smells showing different increases in energy consumption between debug and release mode, as well as private visibility member ignoring methods showing no energy consumption difference to "fixed" MIM code in release mode, but a large difference in debug mode. Additionally, some smells had no additional overhead when using debug mode on the AMS1, but did on the Pixel. These inconsistencies show a need for change in how energy profiling is approached on Android. Where virtually all prior work analyzes apps in debug mode, our preliminary results indicate that energy overhead caused by debug mode is inconsistent between different code smells and devices. These inconsistencies limit the generality of measurements performed in debug mode, and research investigating the energy cost of particular code structures, such as design patterns and code smells, must be performed using release mode to ensure that the optimizations used by modern Android devices are fully applied. Developers measuring the energy consumption of their apps, particularly in a debugging scenario, must also use release mode to ensure that their results are correct.

Finally, we performed a case study on real-world Adyen software, and found that EDATA had a significant influence on the speed with which a developer was able to identify and fix an energy bug. By providing detailed, contextualized information on the energy consumed per-method, we were able to quickly rule out suspected causes of the energy bug, and quantify the effects of the final fix once found. Further, we found that the information obtained through use of EDATA significantly eased the decision-making process, leading to a change that had previously been advocated for, but was considered not to be worth the effort. The results of our case study show the need to make energy consumption data available so that developers and stakeholders can make informed decisions on the energy efficiency of their software.

EDATA is a first step towards an energy consumption monitoring framework for Android, and our results thus far indicate that the information available on Android devices can be used to accurately order methods within an app by their energy consumption, as well as to detect changes between two versions of an app. We have also shown that EDATA fulfills a currently unmet need within the software development industry for detailed energy consumption metrics usable by software developers in their efforts to find and fix energy bugs, as well as stakeholders to make decisions on how to guide the further development of energy-efficient software. We have also identified difficulties faced by developers in our case study that are not yet addressed by EDATA, and provide suggestions on how this can be improved.

In the future, we hope to improve EDATA's user interface, to allow developers with less experience to easily use it as part of their development process. We also hope to improve the precision of its energy estimations through the addition of multi-thread-aware estimation and by further refining our sampling techniques. Finally, by adding accounting of hardware (tail-)energy consumption, we will provide developers with ability to measure the energy consumed by their apps' use of different hardware components. With these improvements, we hope to realize the goal of creating a simple to use framework with which developers can easily write energy efficient Android apps, thereby improving both the development and end-user experience and raising awareness in both developers and stakeholders of the importance of energy efficiency in modern software development.

Bibliography

[1] Accessed: 2023-06-06. url: https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc/README.md#comparing-dwarf-based-and-stack-frame-based-call-graphs.

[2] url: https://developer.android.com/studio/profile/power-profiler%5C#examples.

[3] Hamza Mustafa Alvi et al. *MLEE: Method Level Energy Estimation — A machine learning approach*. en. Dec. 2021. doi: 10.1016/j.suscom.2021.100594. url: http://dx.doi.org/10.1016/j.suscom.2021.100594.

[4] Thomas Ball and James R Larus. “Efficient path profiling”. In: *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE. 1996, pp. 46–57.

[5] *Battery Manager | Android Developers*. url: https://developer.android.com/reference/android/os/BatteryManager%5C#BATTERY_PROPERTY_CURRENT_NOW.

[6] Fares Bouaffar, Olivier Le Goaer, and Adel Noureddine. *PowDroid: Energy Profiling of Android Applications*. Nov. 2021. doi: 10.1109/asew52652.2021.00055. url: http://dx.doi.org/10.1109/ASEW52652.2021.00055.

[7] Lionel C. Briand and Isabella Wieczorek. *Resource Estimation in Software Engineering*. Jan. 2002. doi: 10.1002/0471028959.sof282. url: http://dx.doi.org/10.1002/0471028959.sof282.

[8] Shaiful Chowdhury et al. *GreenScaler: training software energy models with automatic test generation*. en. July 2018. doi: 10.1007/s10664-018-9640-7. url: http://dx.doi.org/10.1007/s10664-018-9640-7.

[9] Shaiful Alam Chowdhury and Abram Hindle. *GreenOracle*. May 2016. doi: 10.1145/2901739.2901763. url: http://dx.doi.org/10.1145/2901739.2901763.

[10] *clock_gettime(2)*. url: https://linux.die.net/man/2/clock_gettime.

[11] Alexandre Cornet and Anandha 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.

[12] Luis Corral et al. “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.

[13] Luis Cruz and Rui Abreu. “Catalog of energy patterns for mobile applications”. In: *Empirical Software Engineering* 24 (2019), pp. 2209–2235.

[14] Luis Cruz and Rui Abreu. *Improving Energy Efficiency Through Automatic Refactoring*. Aug. 2019. doi: 10.5753/jserd.2019.17. url: http://dx.doi.org/10.5753/jserd.2019.17.

[15] Luis Cruz and Rui Abreu. *Performance-Based Guidelines for Energy Efficient Mobile Applications*. May 2017. doi: 10.1109/mobilesoft.2017.19. url: http://dx.doi.org/10.1109/MOBILESoft.2017.19.

[16] Luis Cruz et al. “Do energy-oriented changes hinder maintainability?” In: *2019 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE. 2019, pp. 29–40.

[17] Dario Di Nucci et al. “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.

[18] Muhammad Umer Farooq, Saif Ur Rehman Khan, and Mirza Omer Beg. *MELTA: A Method Level Energy Estimation Technique for Android Development*. Nov. 2019. doi: `10.1109/icic48496.2019.8966712`. url: `http://dx.doi.org/10.1109/ICIC48496.2019.8966712`.

[19] Jason Flinn and Mahadev 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.

[20] Kay Grosskop and Joost Visser. "Identification of application-level energy optimizations". In: *Proceeding of ICT for Sustainability (ICT4S)* 4 (2013), pp. 101–107.

[21] Sarra Habchi, Romain Rouvoy, and Naouel Moha. *On the Survival of Android Code Smells in the Wild*. May 2019. doi: `10.1109/mobilesoft.2019.00022`. url: `http://dx.doi.org/10.1109/MOBILESoft.2019.00022`.

[22] Shuai Hao et al. "Estimating mobile application energy consumption using program analysis". In: *2013 35th international conference on software engineering (ICSE)*. IEEE. 2013, pp. 92–101.

[23] Abram Hindle et al. "GreenMiner: a hardware based mining software repositories software energy consumption framework". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, May 2014. doi: `10.1145/2597073.2597097`. url: `https://doi.org/10.1145/2597073.2597097`.

[24] Erik A. Jagroep et al. *Software energy profiling*. May 2016. doi: `10.1145/2889160.2889216`. url: `http://dx.doi.org/10.1145/2889160.2889216`.

[25] Brittany Johnson et al. *Why don't software developers use static analysis tools to find bugs?* May 2013. doi: `10.1109/icse.2013.6606613`. url: `http://dx.doi.org/10.1109/ICSE.2013.6606613`.

[26] Junha Kim, Yeomin Nam, and Moonju Park. "Energy-aware core switching for mobile devices with a heterogeneous multicore processor". In: *IEEE Consumer Electronics Magazine* 8.6 (2019), pp. 68–75.

[27] Ding Li et al. "An empirical study of the energy consumption of android applications". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 121–130.

[28] Ding Li et al. "Calculating source line level energy information for Android applications". In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, July 2013. doi: `10.1145/2483760.2483780`. url: `https://doi.org/10.1145/2483760.2483780`.

[29] Mario Linares-Vásquez et al. "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.

[30] *Linux perf events: CPU-clock and task-clock - what is the difference*. July 2019. url: `https://stackoverflow.com/questions/23965363/linux-perf-events-cpu-clock-and-task-clock-what-is-the-difference`.

[31] Rodrigo Morales et al. *EARMO: An Energy-Aware Refactoring Approach for Mobile Apps*. Dec. 2018. doi: `10.1109/tse.2017.2757486`. url: `http://dx.doi.org/10.1109/TSE.2017.2757486`.

[32] Irineu Moura et al. "Mining energy-aware commits". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE. 2015, pp. 56–67.

[33] Rui Mu et al. *Research on Customer Satisfaction Based on Multidimensional Analysis*. en. 2021. doi: `10.2991/ijcis.d.210114.001`. url: `http://dx.doi.org/10.2991/ijcis.d.210114.001`.

[34] Lev Mukhanov, Dimitrios S Nikolopoulos, and Bronis 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.

[35] Adel Noureddine et al. “Runtime monitoring of software energy hotspots”. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 2012, pp. 160–169.

[36] Zakaria Ournani et al. “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.

[37] Fabio Palomba et al. *Lightweight detection of Android-specific code smells: The aDoctor project*. Feb. 2017. doi: `10.1109/saner.2017.7884659`. url: `http://dx.doi.org/10.1109/SANER.2017.7884659`.

[38] Fabio Palomba et al. *On the impact of code smells on the energy consumption of mobile applications*. en. Jan. 2019. doi: `10.1016/j.infsof.2018.08.004`. url: `http://dx.doi.org/10.1016/j.infsof.2018.08.004`.

[39] Abhinav Pathak, Y Charlie Hu, and Ming 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.

[40] Rui Pereira et al. *SPELLing out energy leaks: Aiding developers locate energy inefficient code*. en. Mar. 2020. doi: `10.1016/j.jss.2019.110463`. url: `http://dx.doi.org/10.1016/j.jss.2019.110463`.

[41] Gustavo Pinto, Fernando Castor, and Yu David Liu. *Mining questions about software energy consumption*. May 2014. doi: `10.1145/2597073.2597110`. url: `http://dx.doi.org/10.1145/2597073.2597110`.

[42] Stephen Romansky. “Estimating Fine-Grained Mobile Application Energy Use based on Run-Time Software Measured Features”. In: (2020).

[43] Sanae Rosen et al. *Revisiting Network Energy Efficiency of Mobile Apps*. Oct. 2015. doi: `10.1145/2815675.2815713`. url: `http://dx.doi.org/10.1145/2815675.2815713`.

[44] *Simpleperf - Android NDK - Android developers*. url: `https://developer.android.com/ndk/guides/simpleperf`.

[45] *Simpleperf README*. url: `https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc/README.md`.

[46] Nathan R Tallent, John M Mellor-Crummey, and Michael W Fagan. “Binary analysis for measurement and attribution of program performance”. In: *ACM Sigplan Notices* 44.6 (2009), pp. 441–452.

[47] Benjamin Westfield and Anandha 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.