EMPIRICALLY INVESTIGATING ENERGY IMPACTS OF SOFTWARE ENGINEERING DECISIONS

by

Cagri Sahin

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Summer 2017

EMPIRICALLY INVESTIGATING ENERGY IMPACTS

OF

SOFTWARE ENGINEERING DECISIONS

by

Cagri Sahin

Approved: _____

Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

James Clause, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Lori Pollock, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

John Cavazos, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Yu David Liu, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I would like to express my immense gratitude to my advisor, Dr. James Clause, for his invaluable guidance, constant support, enthusiasm, and encouragement throughout my PhD journey. He has always been kind and helpful to me, and willing to share his skills, knowledge, and expertise.

I would like to thank my co-advisor, Dr. Lori Pollock, for giving so generously of her time, and providing invaluable suggestions and advice that have greatly improved not only the quality of my research but also my scientific thinking.

I would also like to thank my committee members, Dr. John Cavazos and Dr. Yu David Liu, for all of their time, advice, and constructive comments.

I want to thank all of my friends in Delaware. We have shared so many memories together and I will always remember them with a big smile.

Last but not the least, I would like to thank my family for their endless love, support, understanding, and patience. I am grateful to have them in my life.

TABLE OF CONTENTS

Appendix

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Software energy efficiency has become an important objective in a broad range of environments where reducing energy consumption is a high-priority goal (e.g., embedded systems in devices, mobile phones and tablets, laptops, and large data centers). Historically, software engineers were unconcerned with energy efficiency; instead they focused on quality attributes such as correctness, performance, reliability, and maintainability. Although the task of improving energy efficiency was left for compiler writers, operating system designers, and hardware engineers, software developers can further reduce the energy usage of the applications that they write beyond what can be achieved at lower system levels. Unfortunately, lack of information about how software engineering decisions impact energy consumption of applications and incorrect assumptions about the underlying causes of energy impacts prevent software developers fulfilling their role in reducing energy consumption.

In addition to reducing the energy consumption of an application, it is also important to maintain the application's energy efficiency. Therefore, developers need to test their applications for energy consumption and energy issues while evolving them. However, the high costs of energy testing can adversely impact the planning process of application evolution since developers must anticipate performing energy testing in response to code changes.

The research in this dissertation aims to enable and support software engineers in developing and maintaining energy-efficient applications in two ways. First, we have conducted empirical studies that examine the software engineering decisions to improve developers' understanding of how the decisions they make potentially impact the energy consumption of their applications. Second, we have developed a technique that predicts energy testing requirements of proposed code changes to help developers

in making informed decisions and creating an effective timeline during the planning process of application evolution.

Chapter 1

INTRODUCTION

Computers are now used in a broad range of environments including embedded systems in devices; mobile phones and tablets; laptops and desktops used for everyday computing tasks; and large data centers for enterprise and cloud computing.

As the use of computers has expanded in these areas, so too has concern about the amount of energy that they consume. For embedded systems, where recharging can be difficult or impossible, and laptops, mobile phones and tablets, where battery life is an important selling point, extending the lifetime of a system has become a major design goal. Any computing platform in which heat or fan noise may be a factor also demands attention to energy usage. Data centers are limited in scalability as they struggle with soaring energy costs as many large companies rely on fast, reliable, and round-the-clock computing services. On large-scale computing clusters, like data centers, even a small reduction in energy consumption can have large effects. In short, across nearly all computing contexts, reducing energy consumption has become a major concern.

Although the computing environments described above are diverse, they share a common limitation: They would be more energy-efficient if the software they execute was designed and implemented with regard for energy consumption. Historically, software engineers have focused on quality attributes such as the correctness, performance, reliability, and maintainability of the software they create while concerns about energy were left for compiler writers, operating system designers, and hardware engineers. While this strategy has been successful—indeed, researchers have made advances in reducing energy consumption by designing computer architectures that are more energy efficient (e.g., [30, 33, 63, 55, 96, 112]), developing compiler optimizations focused

on energy usage (e.g., [49, 51, 50, 53, 60, 64, 99, 106]), creating operating systems that help manage energy usage (e.g., [37, 93, 94, 95, 119]), and designing power-aware hardware and batteries (e.g., [3, 23, 34])—even greater results can be achieved by enabling, encouraging, and supporting the participation of software developers in the process of reducing software energy usage.

A recent broad-based study revealed that software engineers have begun to care and think about energy consumption of their applications [76]. However, there are several obstacles to software engineers fulfilling their role in reducing energy consumption. First, software engineers do not understand how the choices and tradeoffs they make on a daily basis impact the energy consumption of their applications. Unlike for common optimization targets such as execution time and memory usage, where software engineers feel they have at least some understanding of the impacts of their decisions, they simply have no idea what the energy impact of a decision will be. Second, they often have incorrect assumptions about the underlying causes of energy impacts and how other nonfunctional aspects of their applications relate to energy usage. For example, one of the common assumptions is that energy consumption is directly associated with the CPU utilization. However, only measuring reduction on CPU utilization cannot be translated into actual energy savings. These obstacles can be overcomed with a better understanding of the implications of software engineering decisions with regard to energy consumption, and software engineers can play an important role in reducing the energy usage of the applications they write.

In addition to reducing the energy consumption of an application, it is also important to maintain its energy efficiency. Software engineers currently plan code changes to evolve their applications without knowing whether those changes impact energy consumption of the applications. This lack of information introduces the need to test applications for energy consumption and energy issues in response to code changes. Unfortunately, the high costs of energy testing can significantly increase the total testing cost and adversely impact the planning process of application evolution. For example, developers might be limited in the number of changes they can include

in a release because they must conservatively plan to conduct energy testing after each change. Predicting the amount of energy testing required for proposed changes can help software engineers to develop a realistic and effective application evolution timeline. Furthermore, they can make decisions on code changes such as ordering, postponing, or canceling them.

The overall goal of my research is to enable and support software engineers in developing and maintaining energy-efficient applications. My dissertation work addresses this goal first by gathering knowledge about how software engineering decisions impact the overall energy usage of an application and second by developing a technique for supporting the software engineering process.

The main contributions of this dissertation include (1) guidelines to design and conduct high-quality empirical studies on software engineering decisions with energy consideration; (2) data generated by four empirical studies of major software engineering decisions including design patterns, code refactorings, code obfuscations, and performance tips; (3) analyses of the generated data to determine how software engineering decisions impact energy usage; (4) a technique to predict energy testing requirements of proposed code changes; and (5) a prototype implementation of the technique for Android applications.

The following chapters are organized as follows. Chapter 2 provides background information on related work. Chapter 3 introduces energy measurement approaches and energy measurement infrastructures used in the empirical studies. Chapter 4 describes empirical studies of software engineering decisions. Chapter 5 describes an approach to predict energy testing requirements of proposed code changes with a prototype implementation of the approach. Finally, Chapter 6 summarizes contributions of the dissertation and discusses the potential future work.

Chapter 2

BACKGROUND AND STATE OF ART

In today's computing enviroments, energy consumption is an important topic. There have been successful strategies at the system and programming language levels for improving energy efficiency of software. In addition to these strategies, researchers have begun to investigate how design and implementation decisions made by software developers impact software energy consumption to combat the lack of knowledge available to developers and to optimize energy usage at the software level. This chapter discusses related work in these categories. Note that, the most closely related area of work to this dissertation is provided within Chapters 4 and 5 to simplify reader's task of comparing with this dissertation.

2.1 System-Level Strategies for Energy Efficiency

There is a significant amount of research on optimizing energy usage at the system-level including compiler, operating system, and hardware levels.

At the *compiler level*, work has focused on optimizing code to use fewer instructions or a more efficient ordering of instructions; controlling hibernation, dynamic frequency and voltage scaling; and performing remote task mapping (e.g., [49, 50, 53, 60, 64, 99, 106, 38, 112, 29, 78]).

At the *operating system level*, work has focused on the goals of allowing an operating system to manage energy in the same manner as other system resources (e.g., [119]) and optimizing the balance between power and performance via the automatic selection of power policies during application execution (e.g., [95, 37, 94]).

At the *hardware level*, there has been significant work on many topics including reducing excessive CPU cycles (e.g., [111]), capping RAM energy consumption

(e.g., [28]), and the addition of special cores to support common virtual machine (VM) operations (e.g., [19]). There is also significant work in the area of high performance computing: for example, assigning threads to a subset of the processors to enable power-gated sleep mode for unused processors while not degrading performance (e.g., [46, 80, 86, 33, 63, 55, 96]).

2.2 Programming Language Level Strategies for Energy Efficiency

There are several approaches to helping developers write more energy efficient software at the *programming language-level*. Such work includes new type systems (e.g., [25]), new programming models (e.g., [12, 73, 110, 75]), mechanisms for exposing energy-expensive architectural details (e.g., [72, 74] and manipulating quality-of-service [10] and the precision of the results of the computation at runtime [36, 105].

2.3 Investigating Software Level Impacts

Recent studies at the source code level (i.e., *software level*) have focused on identifying the underlying causes of energy consumption by investigating the impact of various software development decisions (i.e., the impact of developers' decisions). These investigations include empirical studies on the impacts of applying a method or pattern, and choosing among available components. For example, researchers, including us, have investigated the impacts of design patterns [71, 17, 88], code refactorings [26, 98, 100, 89], and performance tips [67, 114, 84] to support developers' decision making with regard to energy usage. We will discuss them in Section 4.7.

Other studies at the software level include investigating the impacts of sorting algorithms [16], web servers [77], advertisements [42], API usage [70], and lock-free data structures [54] within a single application in addition to investigating trends in an application's energy consumption among versions [47, 91] and among separate implementations of the same specification [4, 9]. The remainder of this section discusses these studies in chronological order.

Sorting algorithms are used to reorder elements of a list in a certain order. Efficient sorting is important for optimizing other algorithms which require sorted lists as input. Bunse et al. [16] compared energy usage impacts of choosing among different sorting algorithms executed on an embedded system. They found that the algorithms indeed use different amounts of energy. For instance, among insertion sort, bubble sort, heap sort, merge sort, quicksort, selection sort, shake sort, and shellsort, they found insertion sort to be the most energy efficient. Additionally, they demonstrated that there is no correlation between time complexity and energy usage of sorting algorithms. For instance, a sorting algorithm may consume less energy while it has a worse time complexity.

In multithreaded programs, accessing shared data should be synchronized correctly to ensure data consistency and integrity. The most common technique for maintaining data consistency is to use locks such as mutual execution and coarse-grained locks. Alternatively, lock-free data structures can be used although it is more difficult to implement for developers. Hunt et al. [54] analyzed performance and energy efficiency of lock-free and locking data structures. Based on their study, lock-free data structures improve not only performance but also energy efficiency.

Comparing the energy consumption impacts of selecting between different software systems that achieve the same purpose can help both developers and users make informed decisions. Amsel et al. [4] measured CPU energy usage of several popular Internet browsers including Internet Explorer, Mozilla Firefox, and Google Chrome. Their results showed that Internet Explorer was most energy efficient.

Stereo matching is an open problem and actively researched topic in computer vision. Arunagiri et al. [9] performed a comparative study to solve the global stereo matching problem in terms of performance and energy consumption. Their results suggest that stereo matching with the graph cut algorithm is a lot better than stereo matching with simulated annealing for both terms they consider.

Developers periodically evolve their software and provide new versions of the software. Hindle [47] investigated the effect of software change on power consumption

by using the Firefox web browser and the Azureus/Vuze BitTorrent client applications. He compared the power consumption of different Firefox branches and Azureus/Vuze revisions and tried relate object-oriented metrics with power consumption. The main findings of his work are that power consumption varies among different versions of the same application and there is no strong correlation between static object-oriented metrics and power consumption. Similarly, Pathak et al. [91] showed that two versions of the same app might have significantly different energy consumption behavior. They worked on popular smartphone apps to examine where the energy is spent inside the apps.

The impacts of choosing different web servers (Mongrel, Puma, Thin, and WEBrick) on the energy consumption of a web application were analyzed by Manotas et al. [77]. Their experimental results indicate that a web application's energy consumption depends on the web server used to handle its requests. Furthermore, energy efficiency of the web servers changes based on the executed web application's features.

Application program interfaces (APIs) help developers to build applications by providing routines, object classes, data structures, variables, etc. Linares-Vasquez et al. [70] investigated whether some API method calls in the source code of an application may cause high energy consumption than others and influence energy consumption in the application. They analyzed energy consumption of 55 Android applications and classified 131 out of 807 Android API methods as energy greedy. Based on their findings, most of the energy greedy API methods are related to GUI and image manipulation, and database.

In summary, these studies provide evidence that software engineers can play an effective role in reducing energy usage through their design and implementation decisions.

Chapter 3

ENERGY MEASUREMENT

This chapter begins with background an electrical power and energy definitions. It then describes the different energy measurement approaches for software and introduces the energy measurement infrastructures we built and used in our empirical studies.

3.1 Power & Energy

In this section, we provide definitions of electrical power and energy, and their formulas, as background.

Power: Power is the rate at which energy is transmitted by an electric circuit. Thus, power is the amount of energy consumed per unit time. It is measured in *watts* and real power consumption of an electrical device is calculated by multiplying electric potential(voltage) difference by electric current as shown in the formula:

$$P = V * I \tag{3.1}$$

where

(i) P is *Power*, measured in *watts*

(ii) V is *Voltage*, measured in *volts*

(iii) I is *Current*, measured in *amperes*

Energy: The energy consumed by an electrical device is the product of power and time. Energy is measured in units called *joules* which is equivalent to watt-seconds.

$$W = P * T \tag{3.2}$$

where

(i) W is *Energy*, measured in *joules*

(ii) P is *Power*, measured in *watts*

(iii) T is *Time*, measured in *seconds*

Thus, we need to measure power and time to calculate energy consumption of software during its execution.

3.2 Energy Measurement Approaches

The ability to measure the energy usage of a unit of software is a necessary prerequisite for optimizing its energy usage. Although measuring energy consumption is conceptually simple, this is an active research area. Work in the area of energy usage measurement has been conducted at various levels.

Hardware instrumentation-based approaches (e.g., [109, 117, 48]) use physical instrumentation (i.e., soldering wires to power leads) to measure the actual energy usage of a system. Such approaches have the benefit of being accurate since they measure actual energy usage; however, they are also expensive and difficult to use since they require specialized hardware.

Simulation-based approaches (e.g., [43, 15, 83] use a cycle-accurate simulator to replicate the actions of a processor at the architecture level and estimate energy consumption of each executed cycle. Like hardware instrumentation-based approaches, simulation-based approaches can be accurate, but they are also difficult to use.

Finally, *estimation-based approaches* (e.g., [113, 107, 108, 44, 32, 87, 5, 47] build models of energy-influencing features and use such models to estimate energy usage. For example, Hao et al. [44] and Seo et al. [107, 108] construct energy models of Java bytecode and then use the models to estimate the energy usage of a given method or execution path. Estimation-based approaches are frequently less accurate than hardware instrumentation-based or simulation-based approaches, but they have the benefits of being easier to use and more widely applicable.

3.3 Energy Measurement Infrastructure

In our empirical studies, we used *hardware instrumentation-based approaches* because of their higher accuracy. To measure energy consumption of software, we currently have three different *hardware systems*. Each of our hardware systems is designed for a specific computing environment that includes embedded, desktop, and mobile systems. These systems were used in empirical studies that we conducted based on their availability and compatibility with the considered software engineering decision. For instance, we only had an embedded system during the design pattern study. While our embedded system can measure energy consumption of a small piece of code, our desktop system can measure energy consumption of real applications such as applications we used in the refactoring study. Similarly, our mobile system was used when we conducted an empirical study that investigated energy usage of mobile applications.

3.3.1 Embedded System

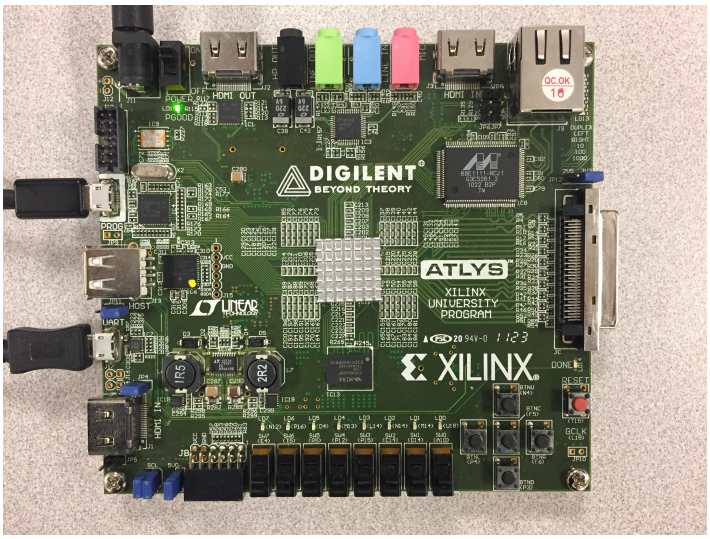


Figure 3.1: XILINX Atlys Spartan-6 FPGA

To measure the amount of energy consumed by executing an application on the embedded system, we developed a Field-Programmable Gate Array (FPGA)-based platform. Our FPGA platform uses the XILINX Atlys Spartan-6 FPGA development

board as an analogue for a standard desktop computer. We chose this board because it is a complete, single-board computer system with a collection of high-performance on-board peripherals including Gbit ethernet, HDMI video, DDR2 memory, audio and USB ports—essentially everything we need to mimic a conventional desktop computer. Moreover, it includes multiple, integrated, real time, power monitors, and the sampling rate of the Spartan-6's power monitors is ≈ 1 ms. In our current configuration of the Spartan-6 board, the 1.2V supply primarily powers the CPU and the Ethernet core, which was disabled in our experiments, the 1.8V supply primarily powers the DDR2 memory, and the 3.3 V supply powers the FPGA I/O, video (HDMI), USB ports, and audio. Note that the Spartan-6 has an additional 2.5 V that we disabled so it does not contribute to overall power consumption. Having multiple monitors allows for easily monitoring the power consumption of individual components, and the fact that they are integrated with the system means that they bypass the current smoothing infrastructure, which allows for more accurate power consumption measurements, and they do not impose any overhead on executing code. In addition, because the Spartan-6 is an embedded system, there is no operating system or other processes that can influence the energy usage of the system (i.e., only the code that we are interested in consumes the energy).

3.3.2 Desktop System

To measure the amount of energy consumed by executing an application on a desktop system, we developed the Low Power Energy Aware Processing (LEAP) platform. Our LEAP platform uses an x86 platform based on an Intel Atom motherboard (D945GCLF2) [109]. It is currently configured with 1 GB of DDR2 memory, a 320 GB 7200 RPM SATA disk drive (WD3200 BEKT), and runs XUbuntu 12.04. Each component in the LEAP system (e.g., CPU, disk drives, memory, etc.) is connected to an analog-to-digital data acquisition (DAQ) card (National Instruments USB-6215) that continually samples the amount of power consumed by the component at a rate of 10 kHz ($\approx$ 10 000 samples per second). The LEAP platform also provides running

applications with the ability to trigger a synchronization signal. This allows for synchronizing the power samples with the portions of the execution that are of interest.

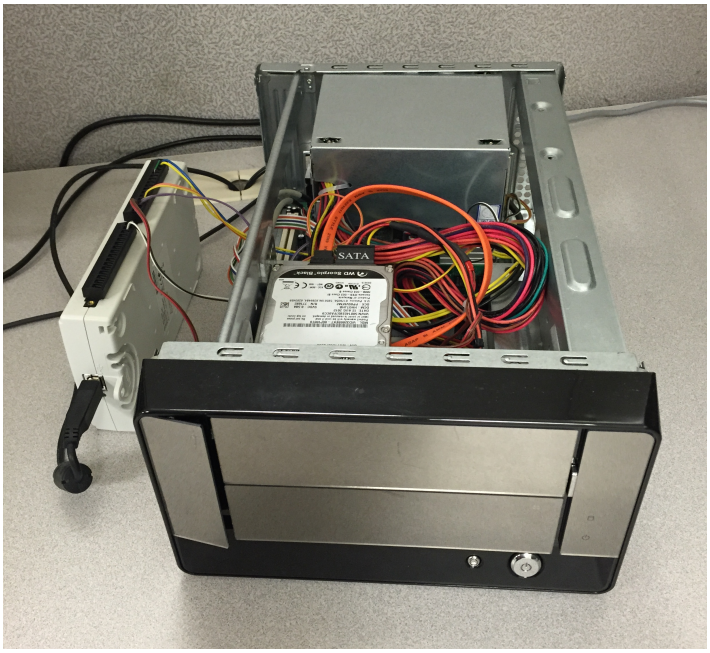


Figure 3.2: Leap Energy Measurement Platform

Note that while the original LEAP specification calls for using the same computer to both run an application of interest and collect power samples, we have modified the design to use dedicated hardware for each of these roles. Using separate machines prevents the introduction of any unwanted measurement overheads. The only remaining source of unwanted overhead is the collection of synchronization information. Because power samples are collected by hardware instrumentation, it is necessary to synchronize them with the application execution to identify, in terms of the application, when a specific power sample was taken. It is possible to account for costs of collecting synchronization information by profiling the energy cost of recording such information and subtracting it from the reported energy numbers. However, because we are concerned with energy consumption relative to a base line rather than absolute numbers and because the energy cost of recording synchronization information is essentially constant, we have removed this step.

3.3.3 Mobile System

To measure the amount of energy consumed by executing an application on a mobile system, we developed two custom-built Energy Measurement Platforms (EMPs) with a sampling rate of around 1000 Hz. Each EMP is based on a commercial Android smart-phone platform: the first EMP is based on a Nexus 4 with 8 GB of storage running Android version 4.3 (Jelly Bean), and the second EMP is based on a Samsung Galaxy S5 with 16 GB of storage running Android version 4.4 (Kit Kat). Figure 3.3 shows a picture of the Galaxy S5-based EMP that we built. The Nexus 4-based EMP is identical except that a Nexus 4 phone is used in place of the Galaxy S5.

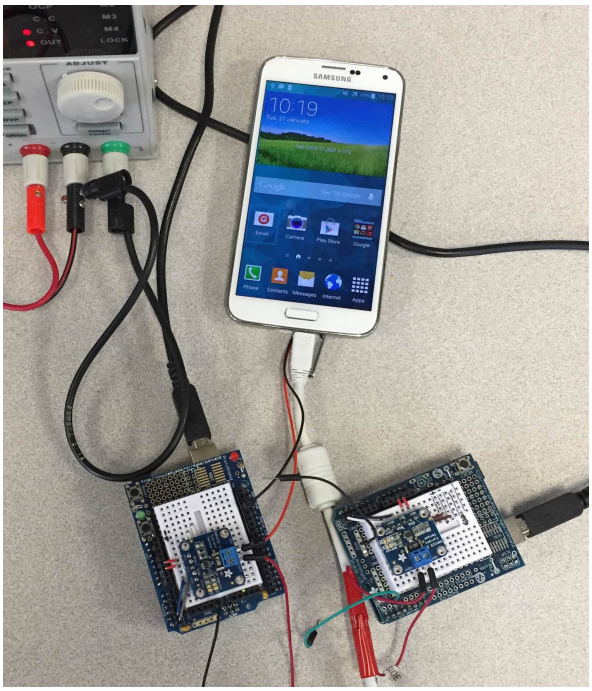


Figure 3.3: Design of the EMPs for the Nexus 4/Galaxy S5.

We chose to use these specific phone models because their hardware specifications are good representatives of the current and penultimate generations of Android mobile phones. They also have contrasting features that allow us to assess the impacts of the performance tips in various phone environments. For example, the Nexus 4 uses a Qualcomm Snapdragon S4 Pro system on chip (SoC) with a 1.5 GHz quad-core Krait central processing unit (CPU) and an Adreno 320 GPU while the Galaxy S5 uses

an Exynos 5 Octa 5422 SoC with two CPUs, a 1.9 GHz quad-core Cortex-A15 and a 1.3 GHz quad-core Cortex-A7, and an ARM Mali T628MP6 GPU.

Instead of using the phone's battery, the EMPs use a 30 V, 5 A DC power supply (KORAD KA3005D). Using an external power source ensures that the phone's battery monitor observes a constant charge level and allows us to compare results across executions without having to worry about variations in the physical battery's performance, age, or temperature, or the phone's power-saving infrastructure.

To sample the voltage and current draw of the phone, EMPs use two Arduino Unos, each equipped with an Adafruit INA219 High Side DC Current Sensor board. One Arduino is used to sense the voltage and current drawn from the DC power supply and the other is used to sense the voltage and current drawn over the phone's USB port. The EMPs report voltage measurements in volts (V) and current measurements in milliamps (mA).

Because our EMPs measure power consumption via hardware that is external to the phones, they do not introduce any measurement overhead to the application. This is ideal, since it means that we do not have to factor out the amount of energy consumed by the monitoring infrastructure itself. However, it also means that the EMPs and the phones do not share a single clock that can be used to identify which samples occurred during an execution of interest. A desktop computer can solve this problem by providing the global clock necessary for performing synchronization. By having the desktop computer start the execution of interest over the Android Debug Bridge (ADB), it is possible to discard power samples recorded before the start of the execution. Similarly, because the duration of the recorded scenarios are known, it is possible to identify samples that were recorded after the end of the execution.

Chapter 4

EMPIRICAL STUDIES ON SOFTWARE ENGINEERING DECISIONS

Our research activities are designed to gain knowledge about how the decisions that software developers make during the course of their daily activities impact the energy usage of the software that they design and implement. Because the scope of decisions that developers make is essentially unbounded, we have decided to focus on several of the most common types of decisions developers make. This chapter presents our investigations on software engineering decisions including empirical studies of the impacts of applying design patterns, code refactorings, code obfuscations, and performance tips [101, 102, 104, 103].

This chapter is organized as follows: Section 4.1 describes the general methodology that we used for conducting the empirical studies; Section 4.2 addresses potential threats to validity of our studies; Sections 4.3 through 4.6 present the empirical studies that we conducted and discuss their results; and Section 4.7 discusses the most closely related work.

4.1 General Methodology

Figure 4.1 shows the general methodology that we follow in our empirical studies. The overall procedure consists of four steps: subject creation, data collection, post processing, and data analysis. In the *subject creation* step, the experimental subjects are created by applying the software engineering decisions to the considered applications. In the *data collection* step, both the applications and the experimental subjects are executed on a suitable hardware-based energy measurement platform to collect power profiles. In addition, other execution data that can be easily gathered is collected (e.g., execution time). Finally, in the *post processing* and *data analysis* steps,

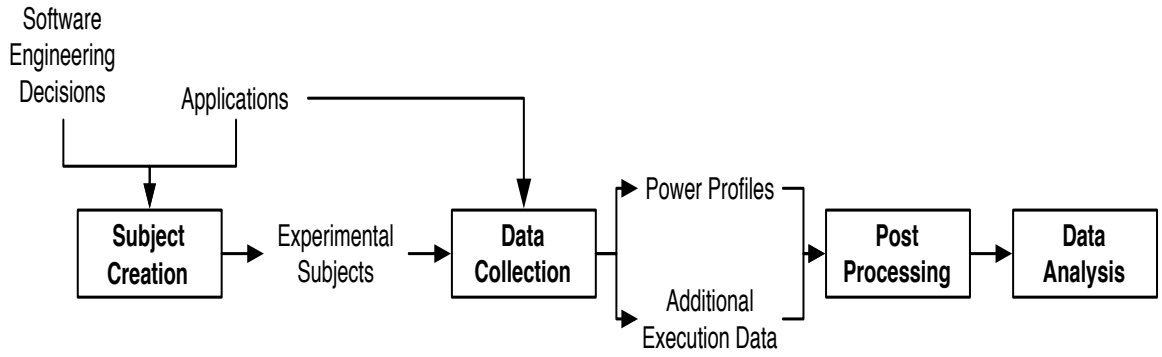


Figure 4.1: General methodology for investigating the energy impacts of software engineering decisions.

the data is post-processed to transform the power profiles into energy usage data, and a statistical analysis of the results is performed. In each step of the methodology, we use proper procedures and techniques and control for the effects of extraneous variables carefully.

4.2 Potential Threats to Validity

One of the most significant threats to the validity of our results is the possibility of energy usage measurement errors either due to imprecise measurements or failing to control for potential sources of noise. To minimize this threat, we rely on hardware-based energy measurement tools that are designed for specific systems. During our experimental runs, we disable all unessential services and programs to minimize potential sources of noise.

Another threat to validity is that selected applications and instances of considered decisions may prevent our conclusions from generalizing to all potential contexts and enviroments where these decisions can be applied. However, we make our selections carefully, explain those selections in detail, and use variety of applications and instances of considered decision in each study. The results of the studies provide information about whether design patterns, code refactorings, code obfuscations, and performance tips have potential to impact the energy usage. Therefore, we believe that our results can give direction to future research.

Finally, the source code of the application may need to be examined and modified manually to apply the considered software engineering decision due to lack of automated tool support. These might cause possible different implementations of our applications when the study is replicated. To prevent this case, we make all versions of our applications publicly available.

4.3 Studies of Design Patterns

Design patterns are commonly used to accomplish high-level goals such as readability, efficiency, and reuse in software systems [39]. In particular, design patterns are solutions to commonly recurring problems in code. They provide a template or description for how to solve the problem, and can be transformed into code by software developers.

In this empirical study, we compared the energy usage of software that uses design patterns against software that does not use design patterns as a way to explore how high-level design decisions can impact an application's energy usage.

4.3.1 Experiment-Specific Methodology

This section describes the details of our study design, including our independent and dependent variables, studied design patterns, and experimental procedure.

4.3.1.1 Experimental Variables

In this study, we considered one dependent variable, the amount of energy consumed by the execution of an application, and one independent variable: whether or not the design pattern is applied to the application.

To isolate the impacts of changing our independent variable (applying a design pattern) on our dependent variable (energy consumption), it is necessary to control for inconsistencies in driving an application. Therefore, we used the same driver code to execute the different versions of each subject application.

Table 4.1: Studied design patterns.

Design Pattern	Description
Creational	
Abstract factory	Provide an interface for creating families of related or dependent objects.
Builder	Separate the construction of a complex objects from its representation.
Factory method	Define an interface for creating an object, but let subclasses decide which class o instantiate.
Prototype	Specify the kinds of objects to create using a prototypical instance.
Singleton	Ensure a class only has one instance, and provide a global point of access to it.
Structural	
Bridge	Decouple an abstraction from its implementation.
Composite	Compose objects into tree structures to represent.
Decorator	Attach additional responsibilities to an object dynamically.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiency.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Behavioral	
Command	Encapsulate a request as an object.
Mediator	Define an object that encapsulates how a set of objects interact.
Observer	Define a one-to-many dependency between objects.
Strategy	Define a family algorithm, encapsulate each one, and make them interchangeable.
Visitor	Represent an operation to be performed on the elements of an object structure.

4.3.1.2 Studied Design Patterns

To investigate the impact of design patterns on energy usage, we selected 15 design patterns, five in each of the categories proposed by Gamma et al. [39] in: *creational*, *structural*, and *behavioral*. Design pattern categories comprise Class instantiation, Class and Object composition, and Class's objects communication, respectively. Table 4.1 lists the specific design patterns that we studied in each category with brief descriptions. Note that, these descriptions are taken from Gamma's book [39].

We chose specific patterns based on the availability of sample code showing an application before and after applying design patterns.

4.3.1.3 Experimental Procedure

Figure 4.2 shows, at a high-level, the procedure we followed in this study, divided into three main steps: *Subject Collection*, *Data Collection*, and *Post Processing*. The remainder of this section describes these three steps in detail.

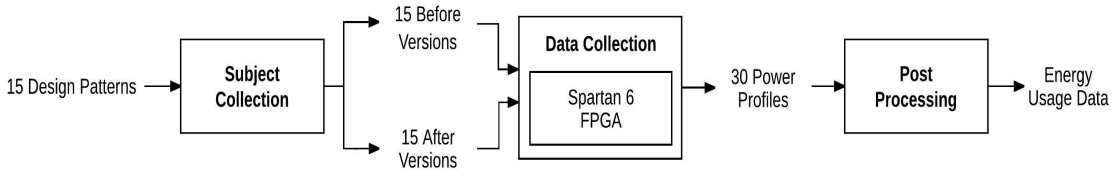


Figure 4.2: High-Level Experimental Procedure of Design Patterns.

Subject Collection

The first step in our procedure is to collect the applications of design patterns. As we mentioned before, selected design patterns have sample code showing an application before and after applying design patterns. For each design pattern, we obtained the before and after versions of an application (i.e., consists of a set of classes) mainly from SourceMaking.com.

Figure 4.3 shows an abbreviated example of an application (a) before and (b) after applying the proxy design pattern. The driver code (c) is the same for both

```
1. class Image {
2.   public:
3.     Image(FILE *f) { ... }
4.     ~Image() { ... }
5.     void draw() { ... }
6. };
```

(a)

```
1. class RealImage {
2.   public:
3.   RealImage(FILE *f) { ... }
4.   ~RealImage() { ... }
5.   void draw() { ... }
6. };
7. class Image {
8.   RealImage *realImage;
9.   FILE *file;
10.  public:
11.  Image(FILE *f)) {
12.    realImage = 0;
       file = f;
13.  }
14.  ~Image() { ... }
15.  void draw() {
16.    if (!realImage)
17.      realImage =
           new RealImage(file);
18.    realImage->draw();
19.  }
20. };
```

(b)

```
1. int main(int argc, char** argv) {
2.   for( ... ) {
3.     Image i1 = new Image("...")
4.     Image i2 = new Image("...")
5.     Image i3 = new Image("...")
6.     i1.draw();
7.     i2.draw();
8.     i3.draw();
9    }
8. }
```

(c)

Figure 4.3: Example code showing (a) before and (b) after applying the proxy design pattern and (c) the application driver code.

the before and after versions. The before version uses direct coupling to draw the images. It creates and initializes all of the image objects before they are actually needed. However, creating all image objects at once might not be desired. The after version uses another object, an image proxy, to instantiate the real image object only when it is requested. Then, the image proxy object forwards all subsequent requests directly to the real image object.

Data Collection

To collect power usage data, we executed the before and after versions of the applications for each design pattern on our FPGA-based platform. During each execution, we recorded power consumption of the FPGA I/O, DDR2 memory, and CPU.

Note that because these applications are small, we modified the driver code to execute multiple times, which allows us to easily collect more samples. The number of iterations for each pattern was chosen so that the total number of samples was as close to 40 000 as possible. 40 000 samples is the maximum number of samples that FPGA-based platform can collect at one time.

Post Processing

To obtain the total energy usage of the executions, we converted the Wattage measurements for the FPGA I/O, DDR2 memory, and CPU to Joules. We then added together the energy usage for FPGA I/O, DDR2 memory, and CPU.

4.3.2 Data Analysis and Discussion

We refined our overall question of whether or not applying a design pattern can impact the energy usage of an application into the following specific research questions:

- *RQ1: Impact* — How does applying a design pattern impact energy usage?
- *RQ2: Consistency* — Do all design patterns within a category (i.e., *creational*, *structural*, and *behavioral*) impact energy usage in the same manner?
- *RQ3: Predictability* — Is it possible to predict the impact on energy usage of applying a design pattern by examining design-level artifacts?

The remainder of this section discusses the results of our study in terms of these research questions.

RQ1: Impact

Table 4.2 shows the experimental data that we collected. The first column, *Design pattern*, shows the grouping of the 15 design patterns. The second column, *# Iterations*, shows the number of times we executed the driver code. The next two columns, *Before* and *After*, show the total energy consumption in Joules of the application before and after applying each design pattern. The fifth column, *Difference*,

shows the difference in total energy usage between the before and after versions of applying the design patterns. Positive numbers indicate that applying the design pattern increased energy usage, and negative numbers indicate that applying the design pattern reduced energy usage. The next column in the table, *Difference per iteration*, shows the difference in total energy usage per iteration (i.e., the difference in the total energy usage divided by the number of iterations). Again, positive numbers indicate that applying the design pattern increased the energy usage, and negative numbers indicate that applying the design pattern reduced energy usage. The seventh column in the table, *% Change* shows the percentage change in total energy usage between the before and after version for each design pattern.

As the data in Table 4.2 shows, the impacts of applying design patterns can vary greatly. For some design patterns (e.g., factory method, prototype, bridge, and strategy), the impact of applying the pattern is relatively small (less than 1 %). While for other patterns, the impact is moderate (e.g., abstract factory, flyweight, decorator, observer) or even substantial (e.g., decorator).

Note that while in absolute terms, the difference in energy usage per iteration is small (0.0002 J to 0.8672 J), there are several points to keep in mind. First, our FPGA platform is designed to be an extremely low-power system. A typical desktop or server computer will consume significantly more energy. Second, our application of the design patterns was minimal. We used the smallest number of classes and the simplest actions possible. Finally, the amount of energy used is shown per iteration (e.g., this is the difference in the amount of energy used by a single dynamic execution of a section of code changed by the application of the pattern). In a typical application, code associated with the implementation of a design pattern may be executed millions or billions of times, especially in the case of long-running server applications. Even though the cost of a single iteration can be small, in aggregate, the difference in energy usage caused by implementing a design pattern can be large.

Table 4.2: Energy usage measurement obtained by running the before and after versions of the design patterns.

Design pattern	# Iterations	Energy usage					# Objects		# Messages	
		Before (J)	After (J)	Difference (J)	Difference per iteration (J)	% Change	Before	After	Before	After
Creational										
Abstract factory	500	87.78	106.69	18.91	0.0378	21.55	11	13	7	12
Builder	750	111.75	113.08	1.33	0.0018	1.19	3	6	18	32
Factory method	500	118.13	118.06	−0.08	−0.0002	−0.07	3	3	1	2
Prototype	750	99.60	98.68	−0.93	−0.0012	−0.93	7	7	3	9
Singleton	250	98.70	99.11	0.42	0.0017	0.42	2	2	7	12
Structural										
Bridge	35	99.78	99.54	−0.24	−0.0070	−0.24	6	11	3	6
Composite	175	97.15	102.14	4.99	0.0285	5.14	17	19	10	11
Decorator	115	13.92	113.13	99.21	0.8627	712.89	14	24	14	15
Flyweight	500	92.89	38.94	−53.95	−0.1079	−58.08	60	6	60	60
Proxy	500	104.33	66.28	−38.05	−0.0761	−36.47	5	2	3	6
Behavioral										
Command	750	98.32	96.53	−1.79	−0.0024	−1.82	7	7	29	29
Mediator	250	120.55	109.02	−11.53	−0.0461	−9.56	4	5	23	26
Observer	400	61.62	99.95	38.33	0.0958	62.20	3	7	4	8
Strategy	500	115.73	115.52	−0.21	−0.0004	−0.18	4	3	9	12
Visitor	90	104.89	97.04	−7.86	−0.0873	−7.49	10	14	16	26

RQ2: Consistency

At a high level, design patterns in a category share a common purpose: creational patterns are concerned with providing alternate ways of creating objects, rather than instantiating objects directly; structural patterns are concerned with class and object composition; and behavioral patterns are concerned with communication between objects. If all patterns in a category impact energy usage in a similar way, it would simplify application design and development by allowing developers to make decisions about whether an entire category of patterns is compatible with their goals with respect to energy usage.

However, as Table 4.2 shows, in our study, the design patterns within a category are not consistent in their impacts on energy usage; in each category, there are patterns that have a positive impact and patterns that have a negative impact. Moreover, not only does the sign of the impact vary (i.e., positive or negative), but the magnitude can be wildly different as well. For example, in the structural category, both the composite and decorator patterns increase energy usage, but the decorator pattern increases energy usage approximately by $\approx 700\,\%$ while the composite pattern increases energy usage by only $\approx 5\,\%$. Similarly, in the behavioral category, the mediator pattern decreases energy usage by $\approx 9\,\%$ while the strategy pattern causes a very small reduction.

RQ3: Predicability

To discover whether it is possible to predict power behavior from a design perspective, we created class diagrams, sequence diagrams, and object diagrams for each of the patterns. Figure 4.4 shows an example of the object diagrams and sequence diagrams for the before and after versions of the application of the proxy pattern. Noticeable characteristics of these diagrams include the number of objects instantiated by a program and the messages passed between objects.

The final four columns of Table 4.2, *# Objects* and *# Messages*, provide a count of the number of objects instantiated and the number of messages passed between the

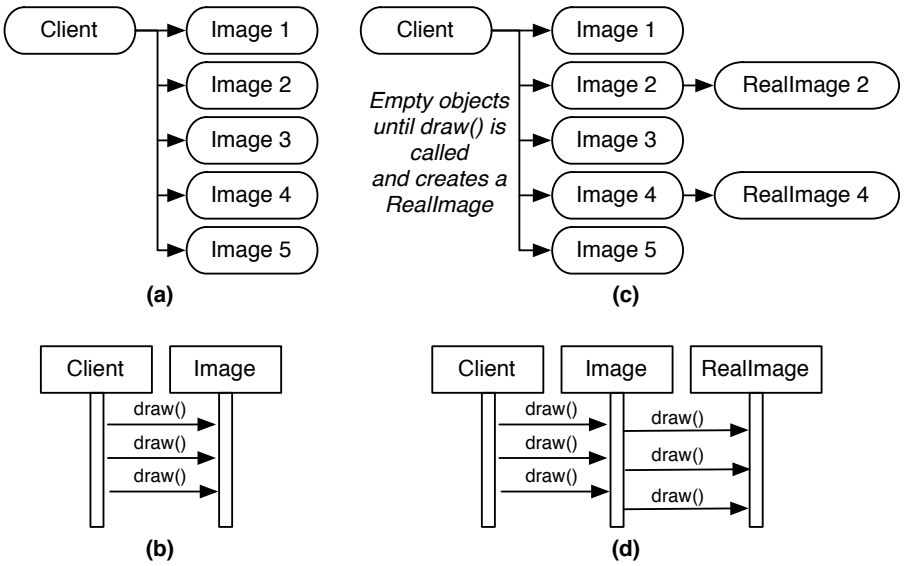


Figure 4.4: Example of design artifacts illustrating before and after applying the Proxy design pattern. (a) and (b) show the object diagram and sequence diagram of the code before applying the proxy pattern. (c) and (d) show the object diagram and sequence diagram of the code after applying the Proxy pattern.

objects for the before and after versions of the applications. In some of the cases that we considered, impact on energy usage appears to be connected with the change in number of objects and number of messages passed between objects. For example, applying the abstract factory pattern increases the number of objects from 11 to 13, the number of messages from 7 to 12, and the energy usage by $\approx 22\,\%$; applying the flyweight pattern decreases the number of objects from 60 to 6 and reduces energy usage by $\approx 58\,\%$; and applying the factory method pattern does not change the number of objects or messages and does not greatly impact the energy usage of the application.

There are however several exceptions to this general trend (e.g., the builder, bridge, command, mediator, and visitor patterns). For example, one of the most expensive patterns in our experiments is the decorator pattern. The decorator pattern does instantiate more objects than its before version, but it does not explain a $700\,\%$ increase in energy usage. Investigating the decorator pattern more closely, we observed that, unlike the other patterns, decorator dynamically creates complex objects without

the use of inheritance. We believe that creating the same complex objects through inheritance requires less energy, as the structure is completely determined at compile time, whereas the flexibility that the decorator pattern offers requires more work at runtime which results in higher energy usage.

Consequently, for these subjects, a reliable prediction about the impact of applying a design pattern can not be made by considering only high-level design artifacts. Further investigation of the role of design and the interplay between design and implementation decisions is needed to better understand how to predict the impact of applying specific design patterns.

4.3.3 Summary

In this section, we have presented an empirical study that investigates the impacts on energy usage of applying design patterns. We considered 15 design patterns, five in each of the creational, structural, and behavioral categories. The results of this study demonstrate that:

(1) Applying design patterns can both increase and decrease the amount of energy used by an application.

(2) Design patterns within a category do not impact energy usage in similar ways.

(3) It is unlikely that impacts on energy usage can be precisely estimated by only considering design-level artifacts.

4.4 Studies of Code Refactorings

One of the most commonly used features in integrated development environments (IDEs) such as Eclipse is the automatic refactoring support that they provide developers [85, 62, 118]. For example, developers can use built-in refactorings to automate common tasks such as extracting code to methods, automatically generating boilerplate code, and introducing indirection. Refactorings typically alter an application to improve its quality in terms of nonfunctional attributes such as readability,

understandability, maintainability, etc. (the same properties that developers have historically been focused on).

While such changes are often beneficial, they may also have detrimental impacts on the application's energy consumption. Since concerns about energy efficiency are rapidly becoming a high priority concern in many environments, the decision to apply transformations must take into account the impacts of the code refactorings on energy consumption. However, developers are not able to make informed choices, primarily due to the lack of information available to them.

To address the lack of information available to developers, we investigated the energy impacts of applying six of the most commonly used code refactorings by creating a total of 197 refactored versions of nine applications.

4.4.1 Experiment-Specific Methodology

This section describes the details of our study design, including our independent and dependent variables, considered applications, studied code refactorings, considered platforms, and experimental procedure.

4.4.1.1 Experimental Variables

In this study, we considered one dependent variable, the amount of energy consumed by the execution of an application, and two independent variables: (1) the choice of whether or not to apply a refactoring, and (2) the platform where the application executes.

To isolate the impacts of changing our independent variables (applying a refactoring and execution platform) on our dependent variable (energy consumption), it is necessary to control for the effects of several extraneous variables (e.g., unnecessary changes in the considered application's code and the inputs that are used to drive the application). The remainder of this section describes how we controlled for such extraneous variables.

Controlling for Extraneous Changes in an Application's Code

In many cases, refactorings are not formally specified. Because of this, different people, or even different tools, may use the same name to refer to different sequences of code changes. This flexibility in nomenclature can be a potential source of bias and a potential source of confusion in interpreting the results of the study. If we compared the impact of refactorings that were inconsistently applied, we would essentially be comparing different refactorings. Similarly, if a developer would apply a substantially different set of code edits that happen to share the same name as one of the refactorings that we studied, the results that they observe could be drastically different than what we observed.

To avoid these potential problems, we must ensure that all refactorings are applied in a consistent, repeatable, and well documented manner. To accomplish this, we relied on the automated refactoring support provided by the Eclipse IDE version 3.7.2 (Indigo). By using the tools provided by Eclipse, we ensured that the changes we made to our considered applications are the same changes that a developer would apply if they applied the same refactoring using the same tool.

Controlling for Inconsistencies in Driving an Application

In general, applications are interactive. They accept input, perform some computation, and generate a response. In our experiments, this interactive nature can introduce a potential source of bias as it is difficult to manually reproduce a given execution exactly. For example, a user can often repeatedly perform the same sequence of actions (e.g., enter text into a form or click a button), but can not maintain the same timing between the actions. Although such differences may seem inconsequential, they may lead to observed differences in energy consumption that are not due to changing our independent variables, but rather differences in how the application is driven. To prevent such bias, it is necessary to be able to deterministically reproduce a given sequence of actions with great fidelity; unit testing frameworks provide this capability.

Unit testing frameworks (e.g., JUnit [59]) are commonly used as part of the software development process. They allow developers to encode how an application should respond when given certain inputs. Such descriptions are then executed and checked by an automated driver component. Because the testing framework is performing the actions instead of a user, the variability in the amount of time that lapses between performing actions is much less. Hence, any observed variations in energy consumption are more likely to be the result of changing an independent variable rather than inconsistencies in driving the application.

4.4.1.2 Considered Applications

We investigated the impacts of applying refactorings on nine Java applications. The specific applications we selected are described in Table 4.3. The first two columns, *Name* and *Version*, indicate the name and version number of each application, respectively. A blank version number indicates that the corresponding application has only a single version. The third and fourth columns, (*# Classes* and *# Methods*), provide the number of classes and the number of methods in each application, respectively. The number of lines of code is reported in the fifth column, *LoC*, and the number of JUnit tests provided with each application is shown in the sixth column, *# Tests*. The final column, *% Coverage*, is the percentage of statements covered by the test suite. For example, version 1.2 of Commons CLI consists of 21 classes, 192 methods, 4739 lines of code, and comes with 187 tests that cover 96 % of the application's statements.

We chose these applications for several reasons. First, they represent a variety of application domains. For example, Commons CLI is a library for processing command-line options, Commons IO is a library for performing various input/output operations, and Joda-Time is a library for handling dates and times. By selecting applications from varied domains, we can improve the generalizability of our results. Second, the applications vary in size. For example, Commons Math has over 100 000 lines of code, while Sudoku only has 497 lines of code. Refactorings are not only applied to large, well established projects. They are also used in the context of new or relatively small

Table 4.3: Java applications.

Name	Version	# Classes	# Methods	LoC	# Tests	% Coverage
Commons Beanutils	1.8.3	118	1199	31 538	1514	63
Commons CLI	1.2	21	192	4739	187	96
Commons Collections	3.2.1	412	3796	63 852	39 143	81
Commons IO	2.4	108	1069	25 663	966	89
Commons Lang	3.1	147	2219	55 626	2047	94
Commons Math	3.0	666	4974	135 796	3451	83
Joda-Convert	1.2	10	65	1317	105	93
Joda-Time	2.1	226	3731	67 590	11 663	88
Sudoku	—	4	57	497	25	81

projects. Again, selecting applications of various sizes can improve the generalizability of our results. Finally, they come with extensive test suites. As we mentioned in Section 4.4.1.1, we are using JUnit tests to drive the applications. We believe that extensive tests are more likely to cover large portions of the application's functionality and to drive the applications in ways that match their expected behavior. In addition to fulfilling our requirements for driving the applications, the unit tests also helped guide the choice of where to apply refactorings in each application (see Section 4.4.1.5).

4.4.1.3 Studied Code Refactorings

To select the refactorings that we studied in our study, we first examined all of the refactorings provided by the Eclipse IDE. We filtered this initial list based on two criteria: (1) the refactorings we select should be commonly used, and (2) applying the refactorings should make some structural change to the application.

To determine how often a specific refactoring is applied by developers, we examined the publicly available data gathered by the Eclipse Usage Data Collector (UDC) [35]. From this data, we identified the most commonly used editing commands (excluding navigation commands and formatting, organizing, and boilerplate generation actions). We then filtered the remaining refactorings and eliminated ones that make no structural changes. For example, although Rename Variable and Rename Method are among the most commonly used commands, the changes that they

make are not evident in the application's compiled bytecode. As such, they have no possibility of altering the amount of energy consumed by the application.

Finally, we sorted the remaining refactorings by how often they can be applied to our Java applications. Some refactorings (e.g., Convert Anonymous Inner Class to Nested Class) can only be applied in very specific circumstances. Since we were interested in identifying general trends about how refactorings impact energy consumption, refactorings that can only provide a single data point are not very useful. To estimate the number of times a refactoring could be applied, we manually examined the code of each application and searched for locations that satisfy the necessary conditions for each refactoring.

As our final set of refactorings to apply, we selected the following six refactorings (listed in alphabetical order):

* Convert Local Variable to Field: Creates a new field by turning a local variable into a field.

* Extract Local Variable: Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.

* Extract Method: Creates a new method containing the currently selected statement or expression and replaces the selection with a reference to the new method.

* Introduce Indirection: Creates a static method that can be used to indirectly delegate to the selected method

* Inline Method: Copies the body of a callee method into the body of a caller method.

* Introduce Parameter Object: Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduced parameter.

These refactorings all fulfill our requirements: they are commonly used and they cause structural changes that are reflected in an application's compiled bytecode.

4.4.1.4 Considered Platforms

We executed the original and refactored versions of each application on versions 7u25 (JVM 7) and 6b27 (JVM 6) of the OpenJDK Java Runtime Environment (JRE). We chose these versions because they are the versions most commonly used in practice.

Although, from a programmer's perspective, there may not appear to be many changes between JVM 6 and JVM 7, there are indeed a significant number of differences. For example, the performance of JVM 7 was improved by techniques such as Tiered Compilation, Compressed Oops (ordinary object pointers), Zero-Based Compressed Oops, and Advanced escape analysis. In addition to improving performance, JVM 7 changes also affected how internal strings are stored (they moved from being part of the permanent generation of the Java heap to the main part of the Java heap), the verifier, and the default garbage collector. All of these changes have the potential to interact with the modifications made by refactorings. Thus, investigating how the the refactorings applied on different underlying platforms impact energy consumption can give valuable information to developers depending on where their application will be deployed.

4.4.1.5 Experimental Procedure

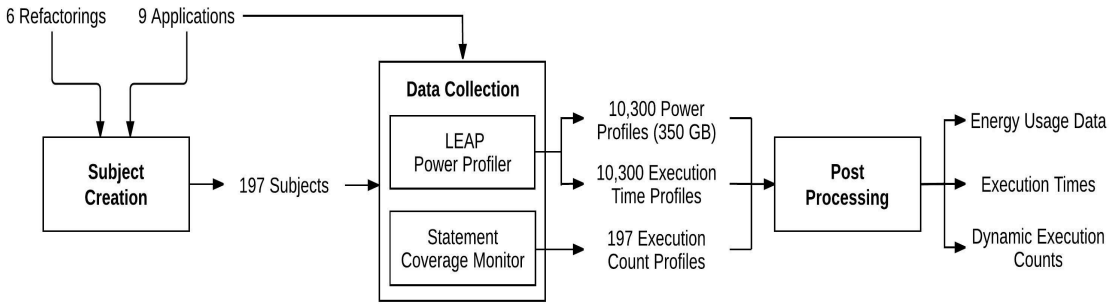


Figure 4.5: High-Level Experimental Procedure of Code Refactorings.

Figure 4.5 shows, at a high-level, the procedure that we followed in this study, divided into three main steps: *Subject Creation*, *Data Collection*, and *Post Processing*. The remainder of this section describes these 3 steps in detail.

Subject Creation

The first step in our procedure is to create our set of experimental subjects. Because we are interested in the impacts of applying a refactoring to an application, our experimental subjects are versions of our considered applications with a refactoring applied. To create the necessary refactored versions, we carried out the following sequence of actions.

First, for each considered application, we used Atlassian's Clover coverage tool (version 3.1.11) to identify the portions of the application that are covered by its test suite [24]. This coverage information serves as a filter to prevent applying a refactoring in a segment of the application that is not executed by the test suite. The impacts of refactorings in such areas would be unobservable because the code would not be executed.

Next, we identified a set of suitable locations where each refactoring could be performed. For each refactoring, we manually examined the covered portions of each application and searched for locations where the preconditions necessary for applying a refactoring are satisfied. We then attempted to apply the refactorings at the selected locations to create four different refactored versions of each application. Four is the maximum number of locations where refactoring could be applied for the smallest application. Basically, a refactored version was created by applying a refactoring at a selected location. Sometimes less than four versions could be created because there exists no possible locations for that refactoring.

To actually apply the refactorings, as mentioned previously, we used Eclipse's built-in refactoring tools. Figure 4.6 shows, for a code excerpt from Sudoku, (a) the original and (b) refactored versions of the code when applying Inline Method to the

```
public List<Box> getPeers(Puzzle puzzle) {
  ArrayList<Box> peers = new ArrayList<Box>();
  addUnique(peers, getPeersInSameRow(puzzle));
  addUnique(peers, getPeersInSameColumn(puzzle));
  addUnique(peers, getPeersInSameSubSquare(puzzle));
  return peers;
}

private void addUnique(ArrayList<Box> peers,
                       List<Box> peersInSameRow) {
  for (Box peer : peersInSameRow)
    if (!peers.contains(peer))
      peers.add(peer);
}
```

(a) Original

```
public List<Box> getPeers(Puzzle puzzle) {
  ArrayList<Box> peers = new ArrayList<Box>();
  for (Box peer : getPeersInSameRow(puzzle))
    if (!peers.contains(peer))
      peers.add(peer);
  for (Box peer : getPeersInSameColumn(puzzle))
    if (!peers.contains(peer))
      peers.add(peer);
  for (Box peer : getPeersInSameSubSquare(puzzle))
    if (!peers.contains(peer))
      peers.add(peer);
  return peers;
}
```

(b) Refactored

Figure 4.6: Applying the Inline Method refactoring to `addUnique`.

`getPeers` method. Note that not every refactoring attempt was successful; in several cases, Eclipse was unable to perform a refactoring due to an internal error.

When applying the refactorings, Eclipse provides configuration options for all of our studied refactorings. The options and the parameter values for those options that we used for each refactoring are listed below:

* Convert Local Variable to Field: The new field created by the refactoring can be made "public", "protected", or "private". We chose to make it public. Also, we chose to initialize the new field at its declaration location instead of in the current method, when it was possible.

* Extract Local Variable: All occurrences of the selected expression can be replaced by a reference to the newly created variable, or only the selected expression can be replaced. We chose to replace all occurrences.

* Extract Method: The extracted method can be created with "public", "protected", or "private" protection. We chose to make the extracted method public.

* Introduce Indirection: Either all method invocations can be redirected to the newly created static method, or only the selected method invocation can be redirected. We chose to redirect all method invocations.

* Inline Method: The method to be inlined can be inlined into every caller method or only into the selected caller method. We chose to inline it into every caller method if it is applicable.

* Introduce Parameter Object: The new parameter object class can be a top-level class or nested within the current class. We chose to create the new class at the top level. In addition, the signature of the existing method can be changed, or it can be modified to be a proxy method (i.e., the method simply packages its arguments in an instance of the new parameter object class and passes along the new object.) We chose to modify the method rather than keep it as a delegate.

In total, we created 197 subjects. Five of the refactorings, Convert Local Variable to Field, Extract Method, Introduce Indirection, Inline Method, and Introduce Parameter Object, were successfully applied 36 times each, four times in each of our nine applications. The remaining refactoring, Extract Local Variable, was only successfully applied 17 times, 1 time in Commons Collections, 2 times in Commons IO, 3 times in Commons Lang and Joda-Time, and 4 times in Commons Beanutils and Commons Math.

Data Collection

We collected three different types of data: (1) power usage data, (2) execution times, and (3) dynamic execution counts. To collect this data, we first created a set of Apache Ant build files for executing each experimental subject using its test suite. Using an Ant file allows us to execute the subjects with a single command and from the command line. Both of these properties are important as they help reduce noise when executing the subjects.

Power Consumption: To collect power usage data, we executed each subject on the LEAP platform 25 times using JVM 6 and 25 times using JVM 7. Using multiple runs (i.e., 25) allows us to perform a statistical analysis on the impact of refactorings that takes into account the possibility of such fluctuations. To further reduce the possibility of noise, we disconnected the LEAP platform from the network, booted into single user mode, and terminated all unnecessary applications and processes. Although we eliminated many possible sources of noise by carefully configuring the LEAP platform, small fluctuations in energy consumption from execution to execution are still possible.

While each subject was executing, we sampled the power usage of the entire system. In total, we ran 10 300 executions—(197 subjects + 9 original applications) $\times$ 25 repetitions $\times$ 2 platforms—which took over 15 days worth of CPU time and resulted in over 350 gigabytes of raw power usage data.

Execution Time: To collect accurate execution times, we again used the LEAP platform as it also records synchronization information. This synchronization information includes timestamps that correspond to the start and end of the execution. By using this information, we can calculate the total execution time of each execution. Again, this process resulted in 10 300 data points.

Dynamic Execution Count: The final type of data that we collected was how many times each location where the refactorings were applied was executed by the test suite. To calculate this information, we again used Atlassian's Clover coverage

tool, but this time we recorded how many times each statement in each application was executed rather than only recording whether each statement was executed. Note that to collect this information, we only needed to consider one execution of the original, unmodified version of each application. The execution counts for each of the 197 subjects can be calculated from just this coverage information.

Post Processing

The final step in our procedure is to post process all of the collected data. Dynamic execution counts and execution times are usable in their current form, but the power consumption data needs to be synchronized, filtered, and converted to a useable form.

We post-processed the raw power usage data to calculate the total energy usage of each execution. Then, we grouped the collected data by application, applied refactoring, and platform used for the execution. Because of the large size of the power profiles, post processing this data took an additional 25 days worth of time.

4.4.2 Data Analysis and Discussion

We refined our overall question of whether or not applying a refactoring can impact the energy usage of an application into the following specific research questions:

- *RQ1: Impact* — Do refactorings impact the energy usage of an application? If so, how?
- *RQ2: Consistency* — Are the effects of applying a refactoring consistent across applications and across platforms?
- *RQ3: Predictability* — Is it possible to predict the impact on energy usage of applying a refactoring by examining data that is more easily accessible?

The remainder of this section discusses the results of our study in terms of these research questions.

RQ1: Impact

To gather the data necessary to answer our first research question, we performed a Mann-Whitney-Wilcoxon test to determine whether the difference between the amount of energy consumed by the original version and refactored version of each subject is statistically significant. We chose to use the Mann-Whitney-Wilcoxon test because we have one nominal variable (whether or not the a refactoring is applied), one measurement value (the amount of energy consumed), and we do not know whether our data are normally distributed. We chose an alpha (α) of 0.05 and used R version 2.14.1's implementation of the test (i.e., wilcox.test).

Of the 394 tests that we conducted (197 for each platform), 109 ($\approx 28\,\%$) indicated a statistically significant difference in energy usage between the original and refactored versions. This result demonstrates that, although refactorings do not always affect energy usage, it is possible for developers to impact the energy consumption of their applications by performing refactorings. Since refactorings are common, even if not every refactoring performed by a developer impacts energy consumption, developers are likely to perform at least a few refactorings that do indeed impact energy usage.

Table 4.4: Number of times each refactoring causes a statistically significant difference in energy usage.

		JVM 6			JVM 7		
Refactoring	# Subjects	Total	# NI	# PI	Total	# NI	# PI
Convert Local Variable to Field	36	13	5	8	12	3	9
Extract Local Variable	17	3	0	3	0	0	0
Extract Method	36	10	8	2	9	7	2
Inline Method	36	9	4	5	7	4	3
Introduce Indirection	36	12	9	3	9	8	1
Introduce Parameter Object	36	13	4	9	12	4	8
Total	197	60	30	30	49	26	23

To gain additional insight into how the refactorings impact energy usage, we investigated how many times each studied refactoring had a statistically significant

impact on energy usage. This information is shown in the first part of Table 4.4. In the table, the first column, *Refactoring*, lists each of our studied refactorings. The second column, *# Subjects*, shows the number of subjects that were created by applying the refactoring. The third and sixth columns, *Total*, show the number of times each refactoring caused a statistically significant difference in energy usage when the subject was executed using JVM 6 and JVM 7, respectively. As this data shows, the 109 cases where a difference occurs are split relatively equally over the 6 refactorings with Convert Local Variable to Field and Introduce Parameter Object making a difference most often (13 out of 36 for JVM 6 and 12 out of 36 for JVM 7) and Extract Local Variable making a difference least often (3 out of 17 for JVM 6 and 0 out of 17 for JVM 7). Most importantly, the data reveals that every refactoring has the potential to impact energy usage.

The next dimension that we investigated was how frequently each refactoring increased energy usage and how frequently each refactoring decreased energy usage. To answer this question, for the cases where there is a significant difference, we manually examined our data and determined whether the energy usage of the refactored version was more or less than the original version. The results of the investigation are also shown in Table 4.4. In the table, columns four and seven, *# NI*, show the number of times each refactoring had a negative impact (i.e., increased energy usage) for JVM 6 and JVM 7 and columns five and eight, *# PI* show the number of times each refactoring had a positive impact (i.e., decrease energy usage), again, for JVM 6 and JVM 7. For example, Extract Method increased energy usage 8 times and decreased energy usage 2 times on JVM 6. Similarly to how every refactoring has the potential to impact energy usage, each refactoring, with the exception of Extract Local Variable, both increased and decreased energy usage.

Finally, we investigated the magnitude of the differences caused by the refactorings. To determine the magnitude of the differences, we again focused on the cases where there is a significant difference. We calculated the percentage change in the means of the energy usages of the original and refactored versions. The results of these

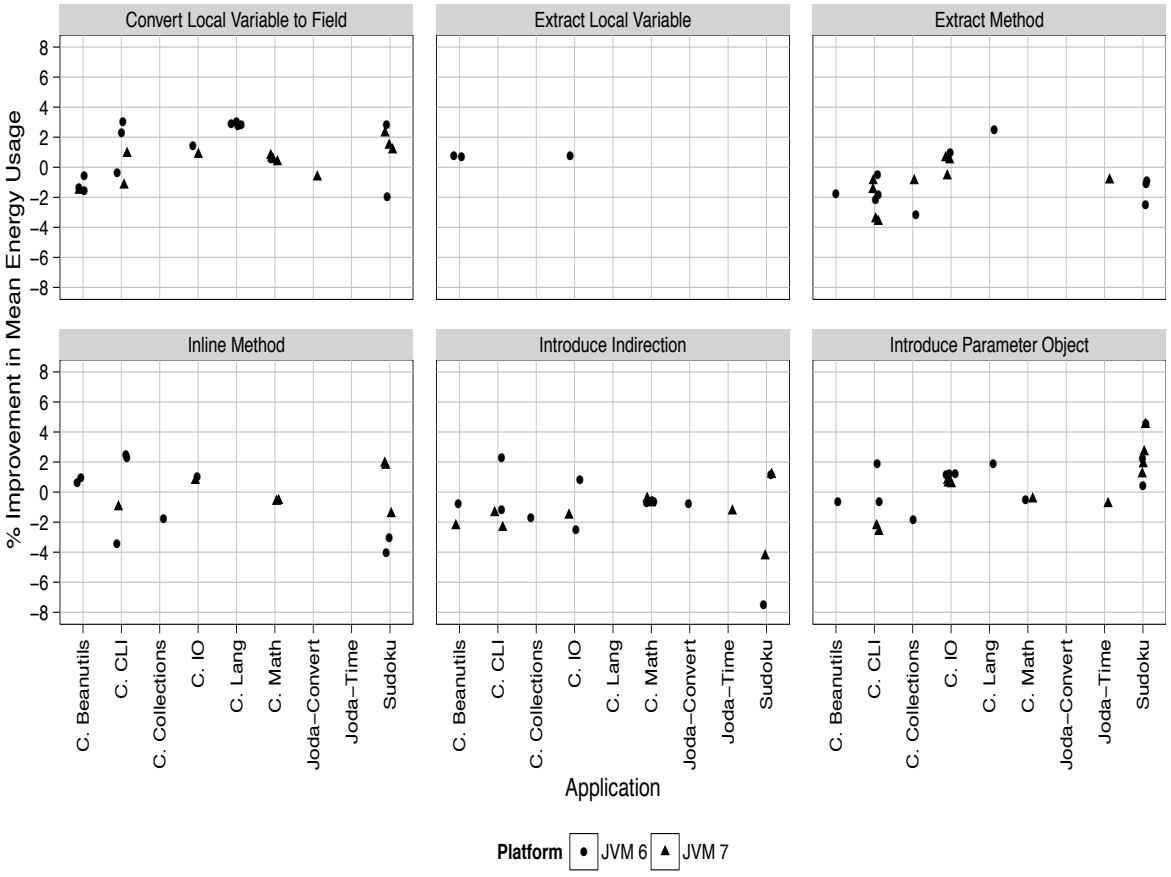


Figure 4.7: Impacts on energy usage of applying refactorings.

computations are shown in Figure 4.7. This figure is composed of 6 subfigures, one for each refactoring. In each subfigure, the x-axis shows each of our 9 applications, and the y-axis shows the percentage improvement in energy usage between the original and refactored versions. For example, 2 subjects of Inline Method in Commons BeanUtils resulted in a percentage change of $\approx 0.75\,\%$, whereas no subject of Extract Local Variable resulted in a significant change in Commons Lang. Note that in this figure, positive values on the y-axis indicate that energy usage improved (i.e., decreased) and negative values indicate that energy usage degraded (i.e., increased). Also note that the points have been "jittered" along the x-axis, to make it more obvious when several points overlap. Finally, the shape of each dot indicates the platform that was used to execute the subject: a • indicates that JVM 6 was used, and a ▲ indicates that JVM 7

was used. As Figure 4.7 shows, the percentage change in energy usage ranges from $-7.50\,\%$ to $4.54\,\%$.

Based on our investigations of the energy usage impacts of refactorings, we have found that:

(1) It is possible that applying a refactoring can significantly impact the energy usage of an application.

(2) All of our studied refactorings can both increase and decrease energy consumption, except Extract Local Variable.

(3) The likelihood of causing an increase or decrease is approximately the same.

(4) Both beneficial and negative impacts have similar maximum percentage change values.

RQ2: Consistency

The goal of our second research question is to determine whether refactorings are consistent in how they impact energy usage: (1) within an application, (2) across applications, (3) within a platform, and (4) across platforms.

To answer these questions, we again used the data presented in Table 4.4 and Figure 4.7. For all 4 questions, it appears that the refactorings are not consistent in their impacts. As Figure 4.7 indicates, the refactorings are not consistent within each application. With the exception of Extract Local Variable, for each refactoring, there is at least one subject of the refactoring that causes energy usage to increase and one subject that causes energy usage to decrease within an application. For example, two subjects of Convert Local Variable to Field cause the energy usage of Commons CLI to increase and three subjects cause it to decrease.

As Figure 4.7 shows, the impacts of the refactorings are not consistent across applications. In many cases, a refactoring that causes a significant difference several times in one application never causes a significant difference in another application. For example, Convert Local Variable to Field causes a significant decrease in the energy usage of Commons Math but does not cause a significant difference in Commons

Collections or Joda-Time. Moreover, a refactoring may decrease energy usage in one application but increase it in another application.

Similarly, refactorings are not consistent within platforms. There are cases where, when run on the same platform, refactorings both increase and decrease the energy usage of different applications. For example, when run on JVM 7, Inline Method decreases the energy usage of Commons IO but increases the energy usage of Commons Math.

Finally, the refactorings are not consistent across platforms. Again, there are cases where applying a refactoring will cause a significant change in energy usage when run on JVM 6 but not when run on JVM 7, and vice versa.

RQ3: Predictability

One of the most common questions that is asked about energy usage is whether or not it is strongly correlated with execution time. Intuitively, it makes sense that they would be strongly correlated; the longer a program runs, the more energy it consumes. However, this is not necessarily true [45]. It is possible for certain components such as disk drives or Wi-Fi radios to consume significant amounts of energy even during short executions. This is why our LEAP platform's ability to profile not only the CPU, but the disk and memory as well, is especially useful. With its capabilities, we can observe the energy costs of the additional components.

We computed a correlation of 0.81 between the execution times and energy usages of our subjects using Kendall's tau, with $\alpha = 0.05$. This indicates that there is a moderately strong positive correlation between execution time and energy usage. Although this result fits with the accepted view, it was surprising for us. Because our applications are CPU-bound and do not use the network or expensive sensors, we expected a much stronger correlation.

To gain some additional insight into whether changes in execution time can be used to predict changes in energy consumption, we identified the cases where applying a refactoring significantly changes execution time. To do this, we used a procedure

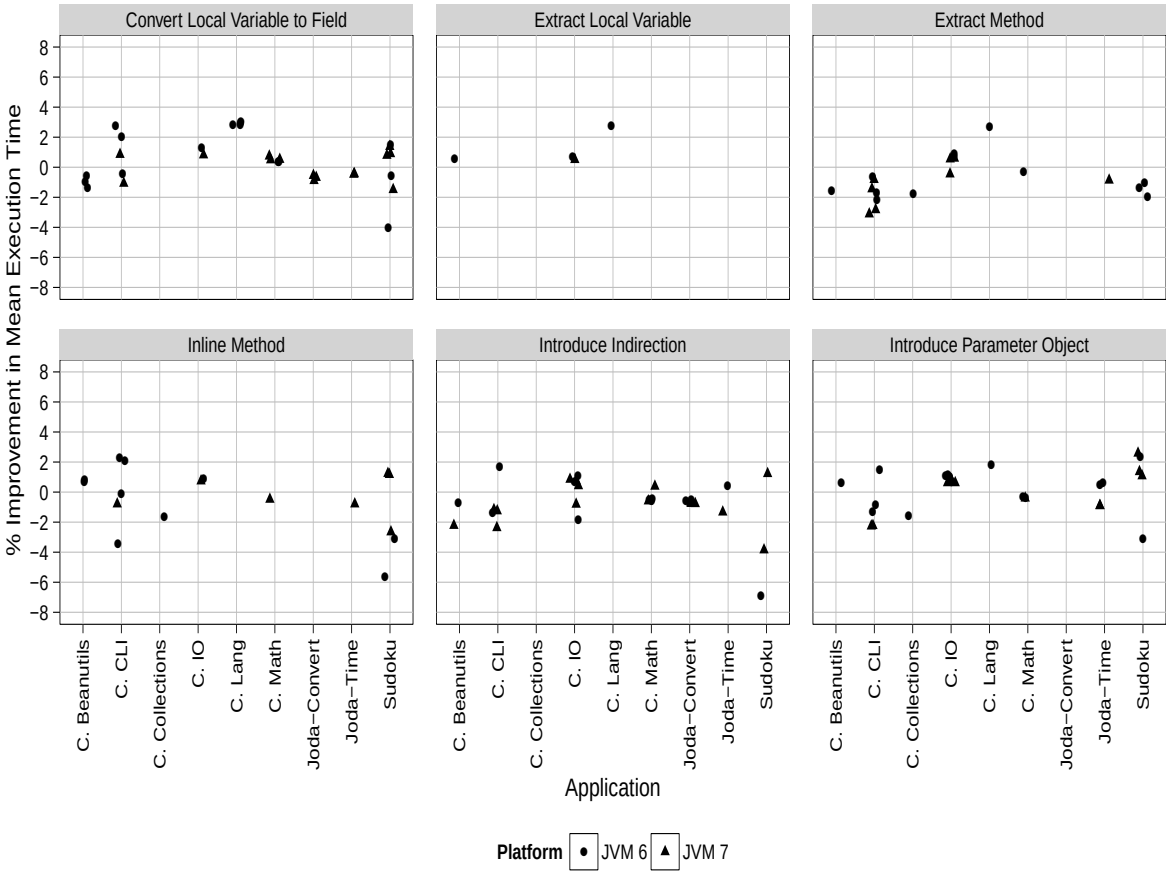


Figure 4.8: Impacts on execution time of applying refactorings.

similar to the one we used to determine when a refactoring significantly impacted energy usage. Again, we used the Mann-Whitney-Wilcoxon test with $\alpha = 0.05$ to identify the cases where execution time was significantly changed. We then computed the percentage difference between the means of execution times of the original and refactored versions in each subject.

Figure 4.8 presents the results of these calculations. Like for Figure 4.7, Figure 4.8 is composed of 6 subfigures, one for each refactoring. The x-axis shows our considered applications and the shape of the points shows whether the subject was executed using JVM 6 or JVM 7. The only difference is that the y-axis now shows the percentage improvement in execution time rather than the percentage improvement in energy usage.

Looking at the results of this comparison, we found that $\approx 11\,\%$ of the time (44 of the 394 cases), a significant change in one measure was not matched by a significant change in the other measure. For the subjects run on JVM 6, there are 76 cases where either energy usage or execution time was significantly impacted by applying a refactoring. For 6 of those cases, there was a significant change in energy usage but not a significant change in execution time; for 16 of those cases, there was a significant change in execution time, but not a significant change in energy usage. In the remaining 54 cases, there was both a significant change in energy usage and in execution time. Similarly, for the subjects run on JVM 7, there were 16 times when there was a significant change in execution time but not a significant change in energy usage, 6 times when there was a significant change in energy usage but not change in execution time, and 43 times when there was a change in both.

Consequently, we believe that, while energy usage and execution time are roughly correlated, execution time alone is unlikely to be an accurate enough predictor of energy usage. A more complex model is needed to account for the situation that execution time itself is unable to explain.

In addition to looking at overall execution times, we also considered whether the dynamic execution count of the locations where the refactorings were applied could predict changes in energy usage. Again, we computed Kendall's tau to check for a correlation. We computed a correlation score of -0.04 with $\alpha = 0.05$. This means that there is essentially no correlation between energy usage and the number of times the location where refactoring is made is executed. As such, we believe that execution counts are a poor predictor of energy usage impacts.

4.4.3 Summary

In this section, we have presented an empirical study that investigates the impacts of applying refactorings on energy usage. As subjects for the study, we used 197 instances of six commonly used refactorings to nine real Java programs of varying sizes

and characteristics. In total, we ran 10 300 executions across two separate platforms. The results of this study demonstrate that:

(1) All studied refactorings can statistically significantly impact the energy usage of an application.

(2) All studied refactorings have the potential to both increase and decrease energy usage, with the exception of Extract Local Variable which we only observed to decrease energy usage.

(3) The impacts of the refactorings do not appear to be consistent across or within applications, or across or within platforms.

(4) More commonly used and easily collectible information such as execution time and dynamic execution counts are unlikely to be able to accurately predict the energy impacts of applying a refactoring.

4.5 Studies of Code Obfuscations

Software piracy is an important concern for application developers. Such concerns are especially relevant in mobile application development, where piracy rates can be greater than 90 % [18]. The most commonly used approach by developers for preventing software piracy is code obfuscation. Basically, code obfuscation makes the code of applications more difficult for a human to understand by using different types of transformations such as renaming variables and methods; merging, splitting, and reordering code, etc. Both Microsoft and Google strongly recommend that developers obfuscate their applications [41, 79]. Google has even gone so far as to integrate obfuscation into the standard Android build system.

However, the decision to apply code obfuscation is performed without regard to their impacts on another area of increasing concern for mobile application developers, energy usage. As a result, an obfuscated application may consume an excessive amount of power, draining the battery and causing users to leave poor reviews or request refunds [7].

Because both software piracy and battery life are important concerns, mobile application developers must strike a balance between (1) protecting their applications and intellectual property, and (2) preserving the limited battery power of the devices where their applications will execute. A major obstacle to striking an appropriate balance between these concerns is a lack of information about how changes to an application impact its energy usage. As a result, developers must either make a poorly informed choice, or more commonly, use an obfuscation tool's default configuration. Unfortunately, these approaches often result in applications that either consume more energy than necessary or are not protected as effectively as they could be.

To address the lack of information available to developers, we investigated the energy impacts of applying 18 code obfuscations by creating a total of 198 obfuscated versions of 11 Android applications.

4.5.1 Experiment-Specific Methodology

This section describes the details of our study design, including our independent and dependent variables, considered applications and scenarios, obfuscation approaches, and experimental procedure.

4.5.1.1 Experimental Variables

In this study, we considered one dependent variable, the amount of energy consumed by the execution of an application, and one independent variable: the obfuscation applied to an application.

To isolate the impacts of changing our independent variable on our dependent variable, it is necessary to control for the effects of several extraneous variables (e.g., unnecessary changes in the considered application's code and the inputs used to drive the application). The remainder of this section describes how we controlled for such extraneous variables.

Controlling for extraneous changes in an application's code

In many cases, obfuscations are not formally specified. Because of this, different tools may use the same name to refer to different sequences of code changes. For example, many obfuscation tools provide a transformation called "string encryption". At a high level, all of these transformations perform the same operation: encrypting the constant strings in an application so that they cannot be easily understood. However, the specific encryption algorithm used can vary greatly. This flexibility in nomenclature can be a potential source of bias and a potential source of confusion in interpreting the results of the study. If we compared the impacts of obfuscations that were inconsistently applied, we would essentially be comparing different transformations. Similarly, if a developer would apply a substantially different set of code edits that happen to share the same name as one of the obfuscations that we studied, the results that they observe could be drastically different than what we observed.

To avoid these potential problems, we ensured that all obfuscations were applied in a consistent, repeatable, and well documented manner. To accomplish this, we relied on several commonly used obfuscation tools (see 4.5.1.4). By using preexisting, automated tools, we ensured that the changes we made to our considered applications are the same changes that a developer would apply if they applied the same obfuscations using the same tool.

Controlling for inconsistencies in executing an application

In general, mobile applications are interactive and event-driven. They accept input, either from a user or from a sensor, perform some computation, and generate a response. In our experiments, this interactive nature can introduce a potential source of bias as it is difficult to manually reproduce a given execution exactly. For example, a user can often repeatedly perform the same sequence of actions (e.g., enter text into a textbox or click a button) but cannot maintain the same timing between the actions. Although such differences may seem inconsequential, they may lead to observed differences in energy usage that are not due to changing our independent variable, but

rather to differences in how the application is driven. In order to prevent such bias, it is necessary to be able to reproduce deterministically a given sequence of actions with great fidelity. Capture/replay tools provide this functionality.

Capture/replay tools are designed to allow for the deterministic replay of a sequence of recorded events. Conceptually, this is accomplished by wrapping an application to insulate it from its environment. When capturing, the wrapper records all of the events that are passed to the application from the environment. When replaying, the wrapper replaces the environment and passes the recorded events to the application. Because precise timing information is recorded during the capture process, there is very little variability in when events are passed to the application during replay. Hence, when using a capture replay tool, any observed variations in energy usage are more likely to be the result of the obfuscations used rather than inconsistencies in driving the application.

We chose to use RERAN as our capture/replay tool, because it is designed to record and replay Android applications [40]. Also, RERAN has a lightweight implementation and its run-time overhead is low, close to 1 %.

4.5.1.2 Considered Applications

As the applications for our study, we used popular, easily accessible Android applications. We selected Android applications for several reasons. First, Android application developers typically care about both the security of their intellectual property and the energy efficiency of their applications. Second, there are many existing obfuscation tools that specifically target Android applications, or, more generally, operate on Java code, that we can use. Third, the source code of many Android applications is freely available, allowing us to easily create many different obfuscated versions. Finally, we have extensive infrastructure to run Android applications and measure their energy usage.

Table 4.5 lists the specific applications that we selected. The first two columns, *Application* and *Description*, list the name of each application and a brief description

Table 4.5: Considered applications.

Application	Description	LoC	Size (MB)
AnkiDroid	Flashcard application	44 913	2.4
Calculator	Default Android calculator	1427	2.6
Calendar	Default Android calendar	41 715	1.4
Clock	Default Android clock	13 477	1.0
DailyMoney	Daily financial tracker	8723	0.4
FrozenBubblePlus	Bubble popping puzzle game	7517	0.2
Nim	Mathematical strategy game	1475	0.8
OIFileManager	File manager	7200	0.7
OpenSudoku	Sudoku game	6079	0.2
SkyMap	Astronomy application	10 921	0.7
Tomdroid	Note taking application	7955	0.6

of its functionality. The third column, *LoC*, shows the application's number of lines of code and the final column, *Size*, shows the size of the application's compiled application package file (APK). The LoC measurement includes only the application itself, while the size measurement includes both the application and its necessary libraries. Because our studied obfuscation tools obfuscate both the application and its libraries, even when the source of such libraries is unavailable, we chose to report both measures to give a better understanding of the amount of code that is being obfuscated.

We chose these specific applications for several reasons. First, they are representative of a wide variety of common application types (e.g., games, study aids, productivity tools, etc.). Second, they are popular and widely used. For example, Calculator, Calendar, and Clock are part of the default Android installation. Finally, they are supported by RERAN. Although RERAN is generally effective at replaying user inputs, such as touch events, it does not support replaying network connections or other sensor readings (e.g., GPS). As such, we were unable to include applications that depend on these types of inputs.

Note that in order to experiment with these applications successfully, we needed to modify them slightly. Primarily, the modifications were made to their build systems so that we could automate the obfuscation processes, but in some cases, we also needed

to modify the application's source code to remove sources of randomness that are not handled by RERAN (e.g., we modified the random number generator to use a fixed seed).

4.5.1.3 Considered Usage Scenarios

Our considered applications are driven primarily by user input. To create the inputs necessary for driving the applications, we examined each application and created one or more usage scenarios. Our goal in creating these scenarios was to capture what we believe to be typical usage patterns for the application (i.e., actions that users are likely to perform). By focusing on typical scenarios rather than scenarios designed to maximize other metrics such as coverage, we were able to gain a better understanding of the impacts of obfuscations on a user's daily interactions with their mobile device.

Table 4.6: Considered usage scenarios.

Application	Name	Description
AnkiDroid	New Deck	Create a new slide deck containing 5 cards.
	Tutorial Deck	Review the 20 cards in the tutorial deck.
Calculator	Advance	Perform several advanced arithmetic calculations.
	Standard	Perform several basic arithmetic calculations.
Calendar	Add Event	Add a new event, search for it, delete it.
Clock	Interval	Create intervals while running the stopwatch.
	Stopwatch	Run the stopwatch for 10 seconds.
	Timer	Run a 10 second countdown timer.
DailyMoney	Add Detail	Enter two transactions.
	View Lists	View details and balances.
FrozenBubblePlus	Level 1	Play the first level.
Nim	Easy AI	Play three rounds with increasing difficulty levels.
OIFileManager	Create File	Create 2 folders, nest folders, delete folders.
	Play File	View 4 pictures and play a ringtone 3 times.
	View File	Open a file. Navigate directories.
OpenSudoku	Easy Level 1	Complete a single "easy" Sudoku grid.
	Hard Level 1	Complete a single "hard" Sudoku grid.
SkyMap	Find Mars	Set time to a fixed past date, searches for Mars.
	Move Zoom	Arbitrarily zoom in/out, moves along the map.
	Show Component	Show each component, toggle night mode.
Tomdroid	Notes	Create a note, search for text, open the note, delete the note.

Table 4.6 shows the specific usage scenarios that we created. The first two columns, *Application* and *Name*, show the application that is used in the scenario and a distinguishing name. For example, AnkiDroid has two scenarios, AnkiDroid: New Deck and AnkiDroid: Tutorial Deck. The third column, *Description*, provides a brief description of what user actions are performed during the scenario. For example, during the AnkiDroid: New Deck scenario, a new flash card deck is created and five flash cards are added to the newly created deck. In total, we created 21 scenarios for our applications: three for Clock, OIFileManager and SkyMap; two for AnkiDroid, Calculator, DailyMoney, and OpenSudoku; and one for Calendar, FrozenBubblePlus, Nim, and Tomdroid.

4.5.1.4 Studied Code Obfuscations

Obfuscation Tools

We had two requirements when choosing obfuscation tools. These were that the tools could (1) obfuscate Android applications, and (2) be easily integrated into the standard Android build system. Because we are repeatedly obfuscating multiple applications, manually applying obfuscations is infeasible. Unfortunately, these requirements eliminated the majority of the free or open source Java obfuscation tools. While such tools can work well for standard Java software, they either introduce changes that result in invalid Android applications when the obfuscated class files are converted to the dex format or they cannot be integrated into the Android build system. The only free obfuscation tool that we found that met our requirements was Proguard 4.10 [97], which is the obfuscation tool that is bundled with the Android Software Development Kit.

Because of the limited number of free tools that met our requirements, we also considered commercial obfuscation tools. Here, we found tools that were more likely to fulfill our requirements. However, their trial or evaluation versions are often limited in functionality (e.g., they only obfuscate parts of an application, or do not support the full suite of configuration options). As such, they are not suitable for our study. To

obtain full-featured versions, we emailed the tool developers and asked if they would be willing to donate a copy of their obfuscation tool. As the result of this process, we obtained copies of three commercial obfuscation tools: Allatori 4.7 [2], DashO 7.2 [27], and Zelix KlassMaster 6.1.3 (ZKM) [122].

Obfuscation Configurations

After reading the manuals of Allatori, DashO, Proguard, and ZKM, we identified several, common high-level configurations or obfuscation types:

* Control-flow (`cf`): Produces "spaghetti logic" that is difficult or impossible to decompile by inserting branching and conditional instructions into the body of a method.

* Rename (`rename`): Renames packages, classes, methods, and fields to short meaningless names (e.g., "a", "b", etc.) and, if possible, moves classes into a single package.

* Optimize (`opt`): Removes unused classes, fields, methods, and attributes; performs simple bytecode optimizations (e.g., peephole optimizations); removes dead code.

* String encryption (`se`): All constant strings in the application are replaced with an encrypted version; decryption methods are added so that strings can be decrypted at runtime.

* All (`all`): Combines all other configurations supported by an obfuscation tool.

Information about the effectiveness of these types of obfuscations can be found in related studies (e.g., [21, 22]). Note that, while the specific changes made by each tool for each configuration may vary (e.g., different string encryption algorithms may be used or branches may be inserted in different locations), from the point of view of an application developer, the results are essentially identical. In addition, not every configuration is supported by every tool.

Table 4.7: Studied obfuscations.

	Supported Configurations				
Obfuscation tool	`all`	`opt`	`rename`	`cf`	`se`
Allatori	✓	✓	✓	✓	✓
DashO	✓	✓	✓	✓	✓
Proguard	✓	✓	✓		
ZKM	✓	✓	✓	✓	✓

Table 4.7 shows which configurations are supported by which tools. The first column, *Obfuscation tool*, shows our studied obfuscation tools and the remaining five columns, `all`, `opt`, `rename`, `cf`, and `se`, show our studied configurations. A checkmark (✓) indicates that a configuration is supported by a tool and a blank space indicates that a configuration is not supported. As the table shows, there are 18 supported combinations. In the remainder of the sections, we will refer to a combination of an obfuscation tool and an obfuscation configuration as an obfuscation. To the best of our knowledge, the studied obfuscations are deterministic in that multiple applications of the obfuscation to the same application produce identical results.

4.5.1.5 Additional Energy Measurement Platforms (EMPs)

To investigate the impacts of code obfuscations in a wider range of platforms, we used two additional custom-built Energy Measurement Platforms (EMPs) that we could access. Similar to our EMPs, these EMPs are based on a commercial Android smart-phone platform. The first EMP is based on a Nexus 3 with 32 GB of storage running Android version 4.3 (Jelly Bean), and the second EMP is based on a Samsung Galaxy S II with 16 GB of storage running Android version 4.3 (Jelly Bean). Figure 4.9 shows a picture of the Galaxy S II-based EMP. The Nexus 3-based EMP is identical except that a Nexus 3 phone is used in place of the Galaxy S II.

These EMPs use a Monsoon Power Monitor from Monsoon Solutions Inc as an external source to power the devices [82]. The Monsoon Power Monitor also samples

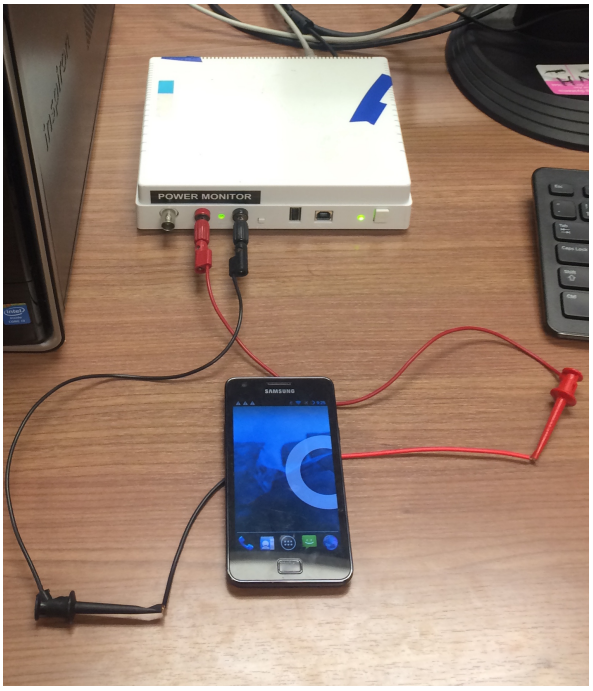


Figure 4.9: Design of the EMPs for the Nexus 3/Galaxy S II .

the voltage and current draw of the phone. It is equipped with a dual range, self-calibrating, integrating system. It has two current ranges with a 16-bit analogue-to-digial converter (ADC), one with a high-resolution range, and the other with a low-resolution range. Software continuously calibrates each of these and selects the proper range during measurement. It reports voltage measurements in volts (V) and current measurements in milliamps (mA).

4.5.1.6 Experimental Procedure

Figure 4.10 shows, at a high-level, the procedure we followed in this study, divided into four main steps: *Subject Creation*, *Replay-able Execution Creation*, *Data Collection*, and *Post Processing*. The remainder of this section describes these steps in detail.

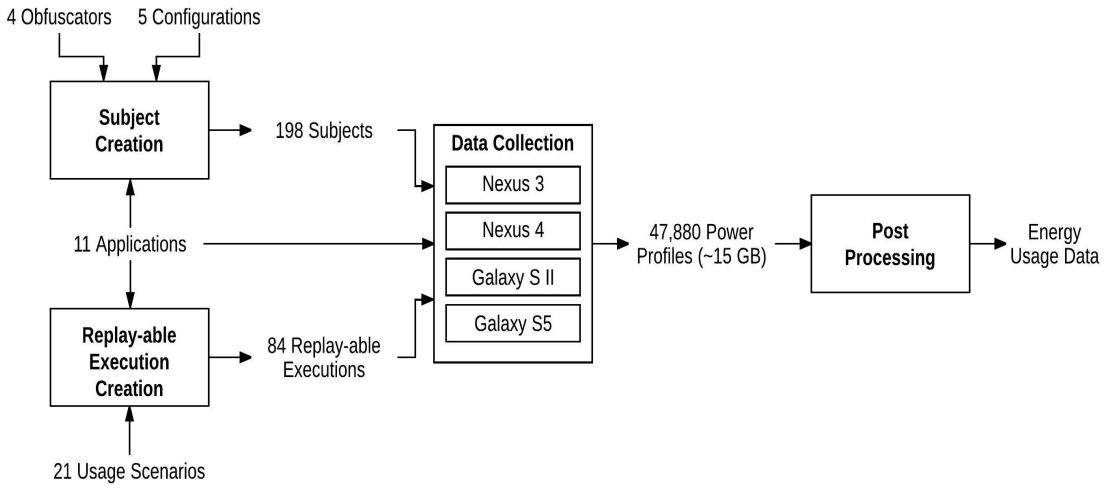


Figure 4.10: High-Level Experimental Procedure of Code Obfuscations.

Replay-able Execution Creation

The first step in our procedure is to create our set of replay-able executions. To create the replay-able executions, we manually performed the actions contained in each scenario while using RERAN's recording tool. Because the replays produced by RERAN are not portable across mobile phone platforms, we created four replay-able executions for each scenario, one for each of our considered EMP platforms. This resulted in a total of 84 replay-able executions (21 scenarios $\times$ 4 platforms). As a sanity check, we then verified that RERAN could accurately replay each execution by running RERAN with the replay-able execution as input and observing the replayed executions. Table 4.8 shows the durations of the replay-able executions for each scenario. The first two columns, *Application* and *Name* show the scenario from Table 4.6 and the remaining four columns, *Nexus 3* through *Galaxy S5*, show the duration of the replay-able execution for each platform in seconds (s).

Table 4.8: Recorded execution durations.

Application	Name	Duration (s)			
		Nexus 3	Nexus 4	Galaxy S II	Galaxy S5
AnkiDroid	New Deck	87	128	83	129
	Tutorial Deck	64	60	108	95
Calculator	Advance	51	79	60	74
	Standard	54	58	57	43
Calendar	Add Event	108	108	113	104
Clock	Interval	28	65	32	64
	Stopwatch	18	20	19	19
	Timer	21	19	23	20
DailyMoney	Add Detail	30	34	50	61
	View Lists	14	16	23	30
FrozenBubblePlus	Level 1	29	36	27	45
Nim	Easy AI	43	75	61	78
OIFileManager	Create File	46	60	80	58
	Play File	60	59	52	55
	View File	22	18	22	36
OpenSudoku	Easy Level 1	138	273	237	172
	Hard Level 1	145	135	223	129
SkyMap	Find Mars	49	42	56	58
	Move Zoom	21	65	19	63
	Show Component	90	100	68	101
Tomdroid	Notes	72	173	62	131

Subject Creation

The second step in our procedure is to create our set of obfuscated applications. To create the necessary obfuscated versions, we obfuscated each application (see Table 4.5) using each obfuscation (see Table 4.7). In total, we created 198 obfuscated applications: 11 applications, each with 18 obfuscated versions.

Data Collection

The third step in our procedure is to collect power usage data. To collect power usage data, we used RERAN to replay each replay-able execution (see Table 4.8) on the corresponding EMP, using both the unobfuscated and obfuscated versions of the scenario's application. For each EMP, each replay-able execution was executed on each version of the application (unobfuscated and obfuscated) 30 times as is suggested by well-known guidelines for empirical study design [8]. While each scenario was executing, we recorded the current and voltage measurements using the EMP.

While the EMP itself does not introduce measurement overhead, the replay infrastructure does—to replay a recorded execution, RERAN installs an application on the phone that injects events into the Android kernel's device drivers. However, because the RERAN process spends most of its time sleeping—it only wakes up to inject events—its overhead is negligible. In addition, because we are concerned with energy usage relative to a base line (i.e., before and after applying an obfuscation) rather than absolute numbers, and the energy costs are consistent across executions, factoring out this cost is not necessary.

To reduce the possibility of noise in the measurements, we terminated all unnecessary applications and processes and, when possible, enabled "airplane mode." Although we eliminated many possible sources of noise by carefully configuring the EMPs, small fluctuations in energy usage from execution to execution were still possible. For example, garbage collection or other operating-system level processes that could not be disabled may have been able to impact energy usage. Multiple runs (i.e., 30) allowed us to perform a statistical analysis on the impact of obfuscations that took into account the possibility of such fluctuations.

In total, we ran 47 880 executions—21 scenarios $\times$ (18 obfuscated versions + 1 unobfuscated version) $\times$ 30 repetitions $\times$ 4 EMPs—which took $\approx$ 924 hours (over 5 weeks) of continuous execution time and resulted in over 15 GB of raw power consumption data.

Post Processing

The final step in our procedure is to post-process the collected data by filtering it and converting it to a usable form. We first filtered the data to remove samples that occurred either before or after the execution. We then converted the current and voltage samples to power measurements in watts by multiplying them together and then dividing by 1000: watts (W) = volts (V) × milliamperes (mA) ÷ 1000. Finally, we converted the resulting power measurements to total energy usage in joules by summing the results of multiplying each power measurement by the length of time between itself and the following sample: joules (J) = watts (W) × seconds (s).

4.5.2 Data Analysis and Discussion

We refined our overall question of whether or not applying obfuscations can impact the energy usage of an application into the following specific research questions:

- *RQ1: Impact* — Do obfuscations impact the energy usage of an application? If so, how?
- *RQ2: Consistency* — Are there any significant differences in the impacts of the studied obfuscation tools or the studied obfuscation configurations?
- *RQ3: Importance* — Are the impacts of applying obfuscations likely to be meaningful or noticeable to a typical mobile application user?

The remainder of this section discusses the results of our study in terms of these research questions. Note that in answering these questions, we are analyzing the data for each platform separately. Because the replay-able executions are not identical (Section 4.5.1.6), it would be inappropriate to analyze the impacts of the obfuscations across platforms.

RQ1: Impact

To gather the data necessary to answer our first research question, we performed Mann-Whitney-Wilcoxon (wilcox) tests to determine whether the difference between

the amount of energy consumed by each scenario when run using the unobfuscated version of the application and each obfuscated version of the application is statistically significant. To check for statistical significance, we chose to use the Mann-Whitney-Wilcoxon test because we have one nominal variable (the obfuscation applied to the application), one measurement value (the amount of energy consumed by the execution), and the test does not require that the data be normally distributed. The resulting p values were adjusted using Benjamini & Hochberg's false discovery rate controlling method to account for performing multiple comparisons [13]. We chose an alpha (α) of 0.05 and used R version 3.0.3's implementation of the test (i.e., `wilcox.test`). Of the 1512 tests that we conducted, 378 (21 scenarios $\times$ 18 obfuscations) for each of our 4 platforms, 791 ($\approx 52\,\%$) indicated a statistically significant difference in the amount of energy consumed by the unobfuscated and obfuscated versions. For each platform, the number of statistically significant differences was 229 ($\approx 61\,\%$) for the Nexus 3, 282 ($\approx 75\,\%$) for the Nexus 4, 107 ($\approx 28\,\%$) for the Galaxy S II, and 173 ($\approx 46\,\%$) for the Galaxy S5.

For the cases where there is a statistically significant difference (i.e., $p \leq 0.05$), we computed Vargha and Delaney's $\hat{A}_{12}$ statistic to calculate the size of the effect of applying the obfuscation [115]. Vargha and Delaney's $\hat{A}_{12}$ statistic is a simple linear transformation of Cliff's δ: $\hat{A}_{12} = (\delta + 1)/2$. We prefer $\hat{A}_{12}$ because it is in the interval $[0, 1]$, while δ is in the interval $[-1, 1]$. Eliminating the negative sign makes Figures 4.11a, 4.11b, 4.11c, 4.11d more readable. In general, the $\hat{A}_{12}$ statistic ranges from 0 to 1 and indicates, on average, how often one technique outperforms another: when $\hat{A}_{12}$ is exactly 0.5, the two techniques achieve equal performance; when $\hat{A}_{12}$ is less than 0.5, the first technique performs worse; and when $\hat{A}_{12}$ is greater than 0.5, the second technique is worse. The closer $\hat{A}_{12}$ is to 0 or 1, larger the effect. For our data, $\hat{A}_{12}$ represents the probability that the *unobfuscated* version consumes *more* energy than the *obfuscated* version.

Figures 4.11a, 4.11b, 4.11c, 4.11d show the $\hat{A}_{12}$ statistics that we calculated. In the figures, the y-axis shows the considered scenarios and the x-axis shows each

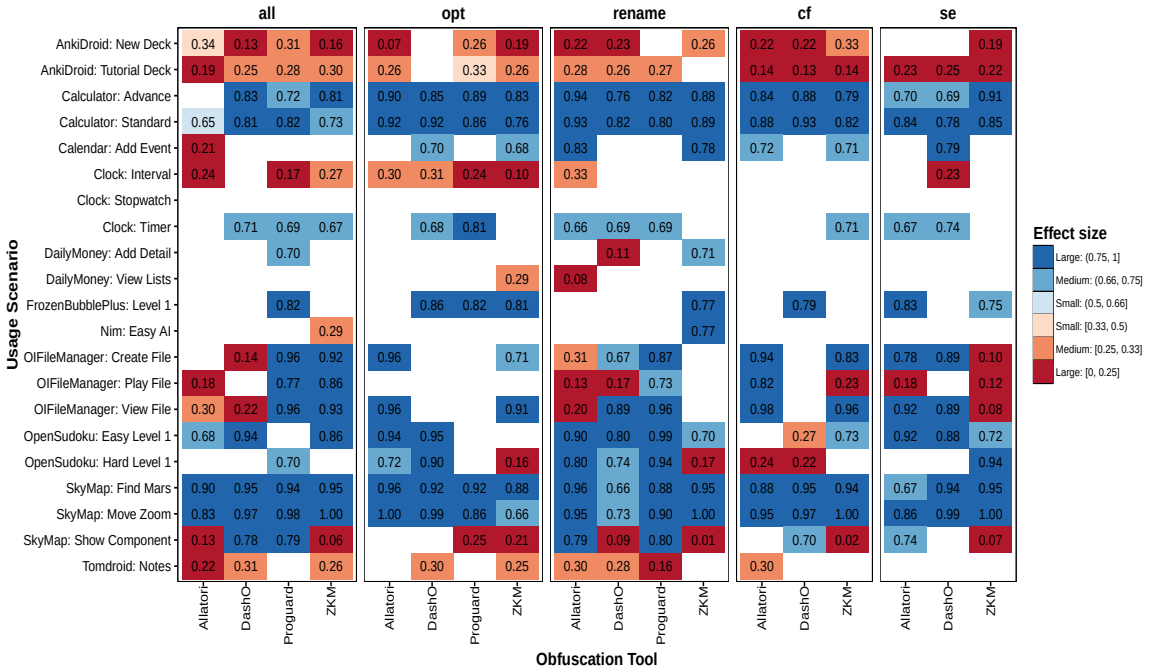


Figure 4.11a: Vargha and Delaney's $\hat{A}_{12}$—probability that an unobfuscated version consumes *more* energy than an obfuscated version when run on the *Nexus 3* platform.

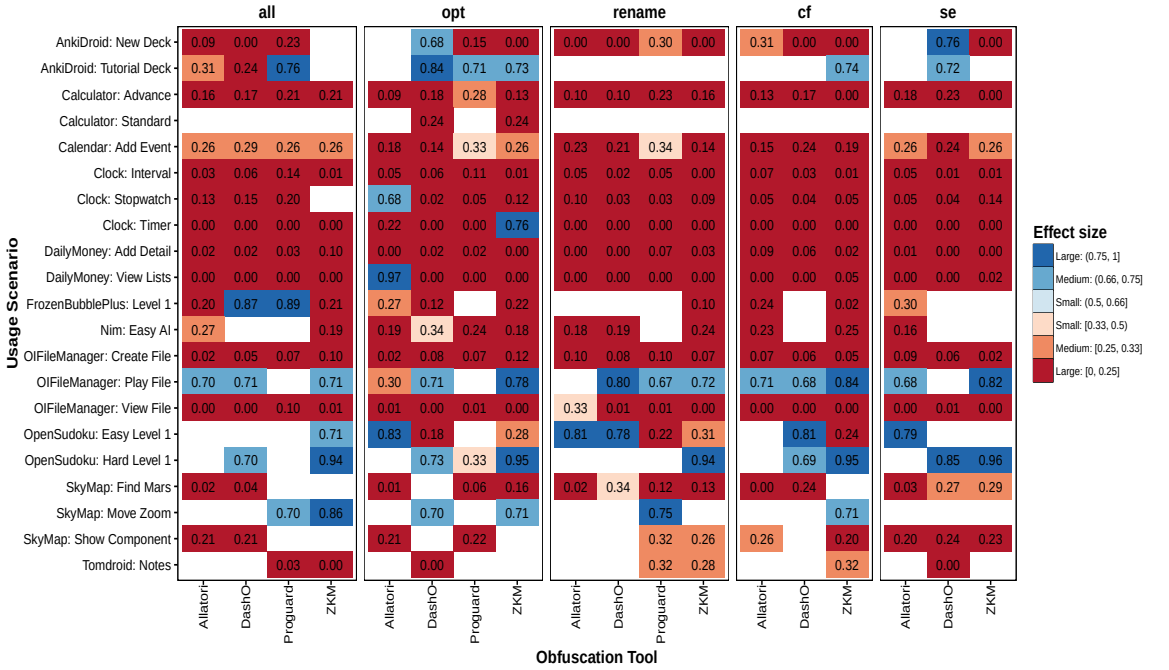


Figure 4.11b: Vargha and Delaney's $\hat{A}_{12}$—probability that an unobfuscated version consumes *more* energy than an obfuscated version when run on the *Nexus 4* platform.

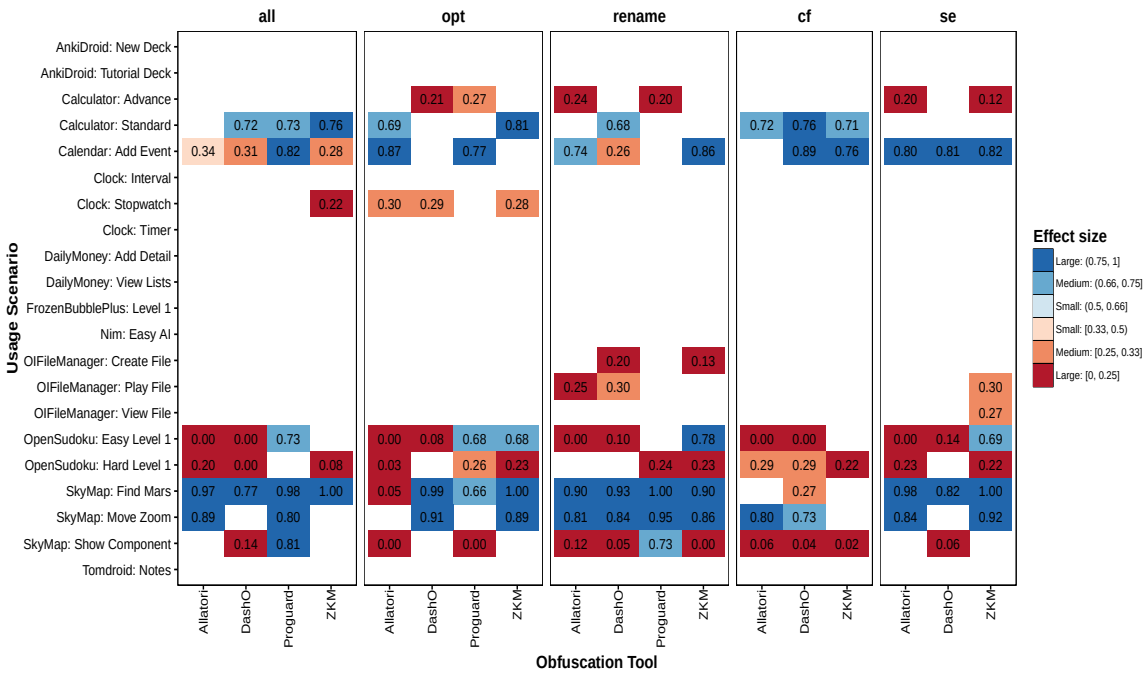


Figure 4.11c: Vargha and Delaney's $\hat{A}_{12}$—probability that an unobfuscated version consumes *more* energy than an obfuscated version when run on the *Galaxy S II* platform.

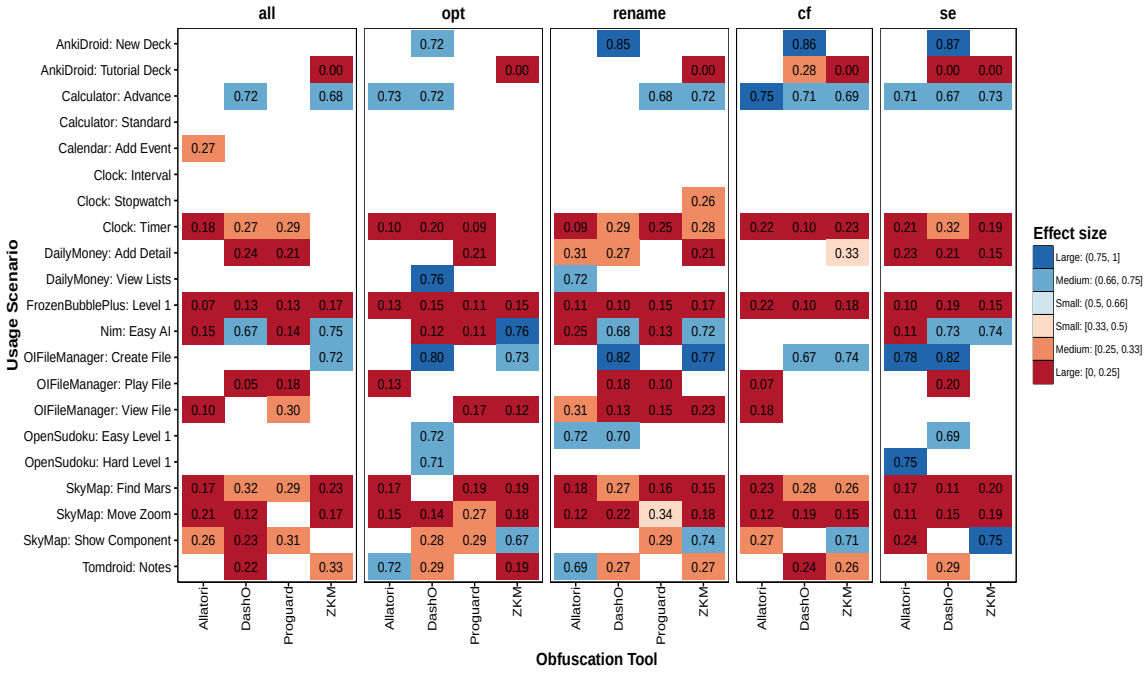


Figure 4.11d: Vargha and Delaney's $\hat{A}_{12}$—probability that an unobfuscated version consumes *more* energy than an obfuscated version when run on the *Galaxy S5* platform.

obfuscation (combination of obfuscation tool and obfuscation configuration). For example, the first grouping shows the $\hat{A}_{12}$ statistics computed between the unobfuscated version of each application and the obfuscated versions produced by each obfuscation tool when using the `all` configuration. The color of each cell indicates the size and direction of the effect. Cells colored blue indicate cases where the unobfuscated version is more likely to consume *more* energy than the obfuscated version (i.e., $\hat{A}_{12} > 0.5$) and cells that are colored red indicate cases where the unobfuscated version is more likely to consume *less* energy than the obfuscated version (i.e., $\hat{A}_{12} < 0.5$). In addition, the color's saturation indicates the size of the effect with the highest saturation indicating a "large" effect ($\hat{A}_{12}$ between 0.75 and 1.0 or between 0 and 0.25), a "medium" effect ($\hat{A}_{12}$ between 0.66 and 0.75 or between 0.25 and 0.33), or a "small" effect ($\hat{A}_{12}$ between 0.5 and 0.66 or between 0.33 and 0.5). Absent values indicate cases where there is not a statistically significant difference in energy usage between the versions.

From this data, we observe that, when all platforms are considered, obfuscations have a generally negative impact on energy usage (i.e., they increase energy usage). In 496 out of the 791 cases when there is a statistically significant difference in energy usage ($\approx 63\,\%$ of the time), the obfuscated version is more likely to consume *more* energy than the unobfuscated version. In the remaining 295 cases ($\approx 37\,\%$ of the time), the obfuscated version is more likely to consume *less* energy than the unobfuscated version. In addition, the size of the effect is most often "large": the effect size is "large" for 575 cases ($\approx 73\,\%$ of the time), "medium" for 204 cases ($\approx 26\,\%$ of the time), and "small" for 12 cases ($\approx 1\,\%$ of the time).

When considered individually, obfuscations also have a negative impact for applications that are executed on the Nexus 4 and Galaxy S5. In 235 out of the 282 cases ($\approx 83\,\%$ of the time) for the Nexus 4 and 127 out of the 173 cases ($\approx 73\,\%$ of the time) for the Galaxy S5, when there is a statistically significant difference in energy usage, the obfuscated version is more likely to consume *more* energy than the unobfuscated version. For applications that are executed on the Galaxy S II, the obfuscations have a more balanced impact. For only 55 out of the 107 cases ($\approx 51\,\%$ of the time) the

obfuscated version is more likely to consume *more* energy than the unobfuscated version. Finally, for applications that are executed on the Nexus 3, obfuscations have a more positive impact. For 150 out of the 229 cases ($\approx 66\,\%$ of the time) the obfuscated version is more likely to consume *less* energy than the unobfuscated version.

Next, we investigated the magnitude of the differences caused by the obfuscations. To determine the magnitude of the differences, we again focused on the cases where there is a significant difference in energy usage. For each combination of user scenario and obfuscation, we calculated the percentage change in mean of the energy usage between the obfuscated and the unobfuscated versions. The results of these computations are shown in Figures 4.12a, 4.12b, 4.12c, 4.12d. The layout of these figures is similar to the layout of Figures 4.11a, 4.11b, 4.11c, 4.11d. The y-axis shows the usage scenarios and the x-axis shows the obfuscations. The content of each cell shows the percentage change in mean energy usage. Again, the color of each cell indicates the direction and magnitude of the change. Blue cells indicate cases where the percentage change is negative (i.e., energy usage decreased), red cells indicate cases where the percentage change is positive (i.e., energy usage increased); darker colors indicate larger values, and absent values indicate cases where there is not a statistically significant difference in energy usage.

Across all platforms, the percentage change in mean energy usage ranges from $\approx -10.1\,\%$ to $\approx 6.9\,\%$ with a median and mean value of $\approx 0.5\,\%$, and a standard deviation of ≈ 2.1 percentage points. For the Nexus 3, the percentage change in mean energy usage ranges from $\approx -10.1\,\%$ to $\approx 3.2\,\%$ with a median value of $\approx -0.7\,\%$, a mean value of $\approx -1.1\,\%$, and a standard deviation of ≈ 2.2 percentage points. For the Nexus 4, the percentage change in mean energy usage ranges from $\approx -3.7\,\%$ to $\approx 6.6\,\%$ with a median value of $\approx 1.2\,\%$, a mean value of $\approx 1.5\,\%$, and a standard deviation of ≈ 1.6 percentage points. For the Galaxy S II, the percentage change in mean energy usage ranges from $\approx -5.5\,\%$ to $\approx 5.5\,\%$ with a median value of $\approx 0.2\,\%$, a mean value of $\approx 0.01\,\%$, and a standard deviation of ≈ 2.0 percentage points. For the Galaxy S5, the percentage change in mean energy usage ranges from $\approx -1.6\,\%$ to $\approx 6.9\,\%$ with a

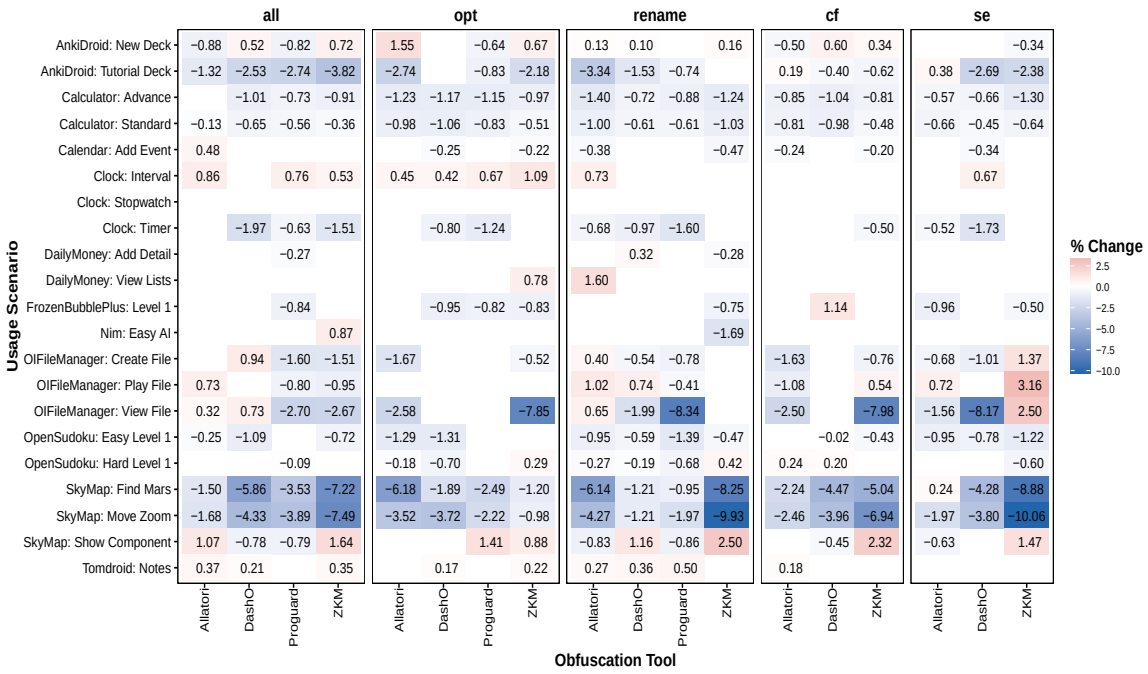


Figure 4.12a: Percent change in mean energy usage when using an obfuscated version instead of an unobfuscated version when run on the *Nexus 3* platform.

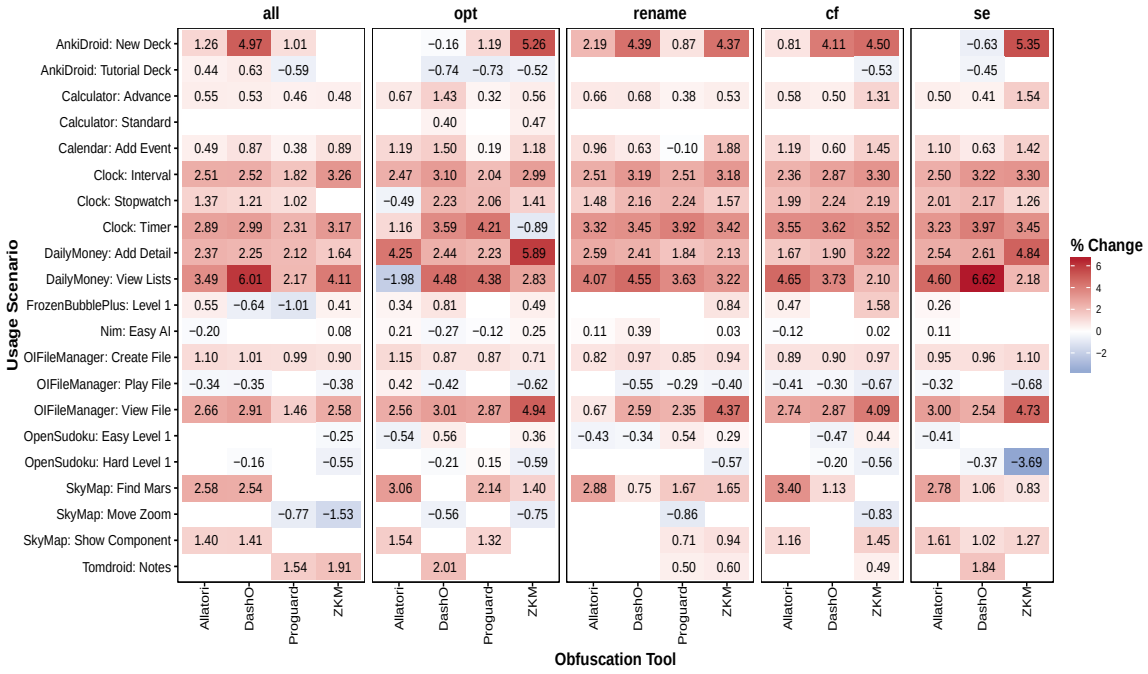


Figure 4.12b: Percent change in mean energy usage when using an obfuscated version instead of an unobfuscated version when run on the *Nexus 4* platform.

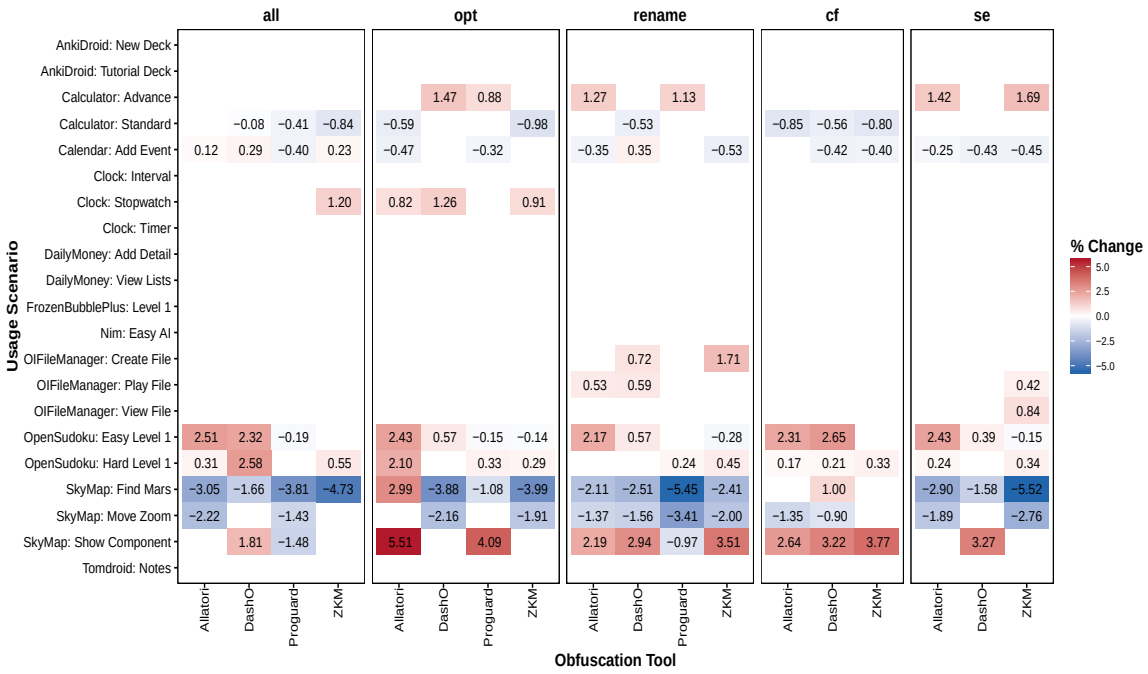


Figure 4.12c: Percent change in mean energy usage when using an obfuscated version instead of an unobfuscated version when run on the *Galaxy S II* platform.

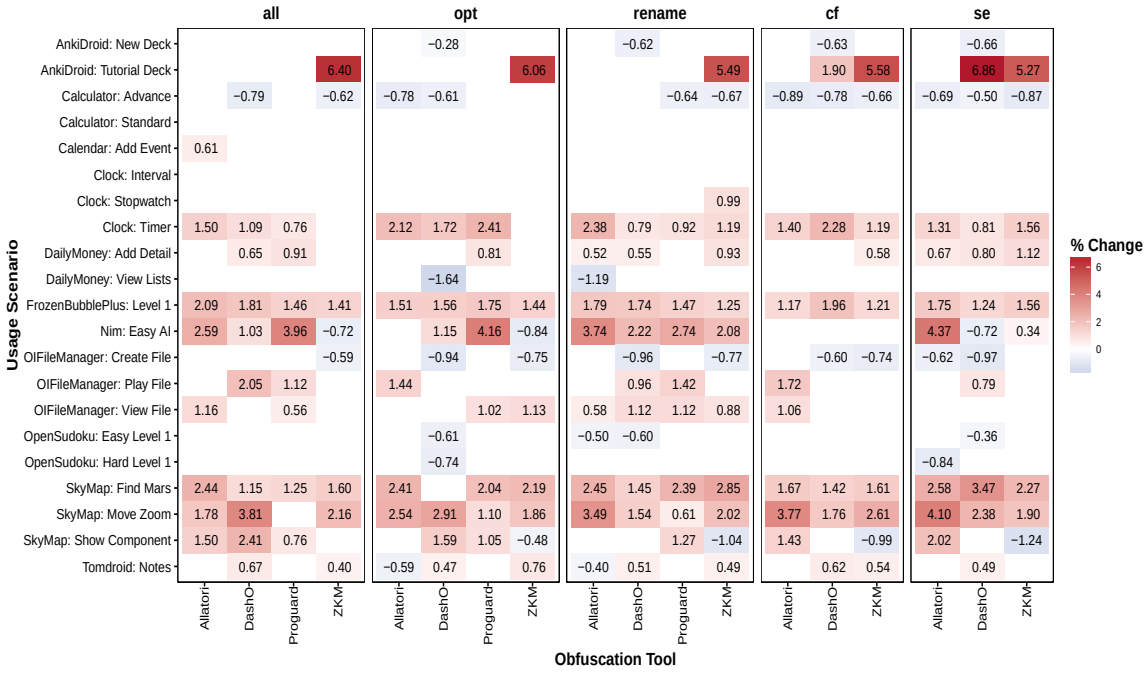


Figure 4.12d: Percent change in mean energy usage when using an obfuscated version instead of an unobfuscated version when run on the *Galaxy S5* platform.

median value of $\approx 1.2\,\%$, a mean value of $\approx 1.2\,\%$, and a standard deviation of ≈ 1.6 percentage points.

From this data, it is clear that, while overall obfuscations are more likely to cause an increase in energy usage than a decrease in energy usage, the magnitude of the change, regardless of direction, is likely to be less than $5\,\%$. When compared to the energy impacts of other code level changes, the energy impacts of obfuscations are closer to the impacts of other focused changes (e.g., refactorings, whose impacts range from $-7.50\,\%$ to $4.54\,\%$ (Section 4.4.2)) than to the impacts of more broad changes (e.g., applying design patterns, whose impacts can approach several hundred percent (Section 4.3.2)).

Based on our investigations into the impacts of obfuscations on energy usage, we have found that:

(1) Obfuscations can, and often do, impact the energy usage of an application with statistical significance.

(2) Individually, all of our studied obfuscation tools and obfuscation configurations can both increase and decrease energy usage.

(3) Across all platforms, the likelihood of causing an increase in energy usage is higher than the likelihood of causing a decrease in energy usage.

(4) Across all platforms, the magnitude of the percentage change in energy usage is most likely to be less than $5\,\%$.

RQ2: Consistency

The goal of our second research question is to determine if there is a statistically significant benefit, with respect to energy usage, to using a specific obfuscation tool or specific obfuscation configuration. To answer this question, we performed several Kruskal-Wallis tests. We chose to use the Kruskal-Wallis test because we want to compare one measurement value (the amount of energy consumed by the execution) across multiple samples (obfuscation tools or obfuscation configurations) and we do not know if our data are normally distributed. We chose an α of 0.05 and used R version

3.1.2's implementation of the test (i.e., `kruskal.test`). In general, if the p value calculated by the Kruskal-Wallis test is less than the chosen α, it indicates that at least one of the samples is significantly different from the others. It does not indicate how many differences occur or among which samples the differences exist. However, this information can be determined by running pairwise Mann-Whitney-Wilcoxon tests with an appropriate correction for performing multiple comparisons (e.g., Bonferroni correction, Benjamini & Hochberg correction, etc.).

Table 4.9: For an obfuscation configuration, is there a statistically significant difference among the obfuscation tools (% change $\sim$ tool)?

Obfuscation Configuration	p value				
	Nexus 3	Nexus 4	Galaxy S II	Galaxy S5	All
`all`	0.39	0.56	0.18	0.45	0.08
`opt`	0.51	0.92	0.20	0.38	0.56
`rename`	0.56	0.75	0.56	0.91	0.83
`cf`	0.67	0.90	0.89	0.90	0.93
`se`	0.80	0.52	0.92	0.87	0.90

Our first set of Kruskal-Wallis tests check whether there are any statistically significant differences in the percentage changes in mean energy usage among obfuscation tools for each obfuscation configuration. A p value less than our chosen alpha would indicate that one of the obfuscation configurations is statistically different from the others. The results of these computations can be seen in Table 4.9. In this table, the first column, *Obfuscation Configuration* shows the name of each obfuscation configuration. The next four columns, *Nexus 3* through *Galaxy S5*, show the p value when each platform is considered individually and the final column, *All*, shows the p value when all four platforms are considered together. Because the computed p values are never less than our chosen α (0.05), we cannot reject the null hypothesis. In practice, this means that, with respect to energy usage, there is no statistical benefit to picking one obfuscation tool over another. Consequently, developers are free to choose their preferred obfuscation tool based on other factors such as supported obfuscations, price,

ease of use, etc., without having to worry about its impact on energy usage.

Table 4.10: For an obfuscation tool, is there a statistically significant difference among the obfuscation configurations (% change ∼ configuration)?

Obfuscation Tool	p value				
	Nexus 3	Nexus 4	Galaxy S II	Galaxy S5	All
Allatori	0.35	0.73	0.37	0.82	0.95
DashO	0.71	0.79	0.95	0.66	0.72
Proguard	0.99	0.80	0.15	0.70	0.11
ZKM	0.98	0.58	0.77	0.98	0.69

Our second set of Kruskal-Wallis tests check whether there are any statistically significant differences in the percentage changes in mean energy usage among the obfuscation configurations for each obfuscation tool. The result of these computations can be seen in Table 4.10. The format of the table is similar to Table 4.9. The first column, *Obfuscation Tool* shows the name of each obfuscation tool. The remaining columns show the p value when each platform is considered individually, *Nexus 3* through *Galaxy S5*, and together, *All*. Again, because the computed p values are never less than our chosen α (0.05), we cannot reject the null hypothesis. In practice, this means that, with respect to energy usage, there is no statistical benefit to picking one obfuscation configuration over another. Again, application developers are free to choose their preferred obfuscation configuration based on factors other than its impact on energy usage.

RQ3: Importance

Our first two research questions were primarily concerned with discovering if and how obfuscations impact the energy usage of applications. The goal of our third research question is to assess whether the observed impacts are likely to be meaningful or noticeable to mobile application users.

To answer this question, we first used Equation 4.1 to calculate, for each platform, the percentage of battery charge that is consumed by each scenario when it is

executed using the unobfuscated version of its application and when it is executed using the obfuscated versions of its application.

$$\%_{charge} = \frac{E}{V} \times \frac{1000}{C \times 3600} \times 100 \tag{4.1}$$

In Equation 4.1, E is the amount of energy in joules (J) consumed by an execution (here we used the mean energy usage of each version of our 30 trials), V is the output voltage of the platform's battery in volts (V), and C is the electric charge of the platform's battery in milliampere hours (mA h). For the Nexus 3, $V = 3.7\,\text{V}$ and $C = 1900\,\text{mA h}$; for the Nexus 4, $V = 3.8\,\text{V}$ and $C = 2100\,\text{mA h}$; for the Galaxy S II $V = 3.6\,\text{V}$ and $C = 1800\,\text{mA h}$; and for the Galaxy S5 $V = 3.8\,\text{V}$ and $C = 2800\,\text{mA h}$.

We then calculated, using Equation 4.2, the amount of time needed to drain each platform's battery from full to empty (i.e., battery life) if the scenario were executed continuously using each version of its application.

$$t_{drain} = \frac{100\,\%}{\%_{charge}} \times D \tag{4.2}$$

In Equation 4.2, $\%_{charge}$ is the percentage of battery charge calculated using Equation 4.1 and D is the duration of the scenario (Table 4.8). Note that the unit of measurement for t_{drain} will be the same as the unit of measurement for D.

Table 4.11 shows the results of this computation. In the table, the first two columns, *Application* and *Name*, show the scenario and the remaining columns, *Nexus 3* through *Galaxy S5* show, for each platform, the mean battery life in hours (h) when the unobfuscated version is run continuously, draining the battery from full to empty.

Finally, we computed the change in battery life for each scenario and obfuscation by subtracting the battery life of each obfuscated version from the battery life of the unobfuscated version. Figures 4.13a, 4.13b, 4.13c, 4.13d show the results of these computations. The five groupings in each figure show the change in mean battery life in minutes (min) when an obfuscated version is used instead of an unobfuscated version. Again, absent values indicate instances where there was no statistically significant

Table 4.11: Battery life when using an unobfuscated version.

		Battery life (h)			
Application	Name	Nexus 3	Nexus 4	Galaxy S II	Galaxy S5
AnkiDroid	New Deck	4.6	4.6	7.7	8.8
	Tutorial Deck	4.1	4.7	7.8	8.1
Calculator	Advance	5.1	4.9	8.0	11.6
	Standard	5.5	5.4	8.7	12.1
Calendar	Add Event	3.7	4.2	7.8	8.0
Clock	Interval	4.9	4.2	8.9	7.0
	Stopwatch	3.9	5.3	7.7	10.6
	Timer	4.2	5.0	8.3	9.8
DailyMoney	Add Detail	4.3	4.8	9.9	8.8
	View Lists	3.9	4.4	7.7	9.7
FrozenBubblePlus	Level 1	3.0	4.1	6.4	5.9
Nim	Easy AI	3.8	3.9	7.2	7.1
OIFileManager	Create File	4.6	4.6	9.9	10.4
	Play File	4.5	4.5	8.7	9.1
	View File	4.0	4.1	7.6	5.7
OpenSudoku	Easy Level 1	4.7	5.4	9.0	9.6
	Hard Level 1	4.8	4.7	8.6	9.2
SkyMap	Find Mars	2.7	3.3	3.3	5.7
	Move Zoom	2.9	3.0	2.2	5.6
	Show Component	3.4	4.1	3.6	7.2
Tomdroid	Notes	4.6	4.1	7.1	7.9

difference in energy usage between the application versions and the color of each cell indicates the direction and magnitude of the change. Blue cells indicate obfuscations that increase battery life (i.e., changes that are beneficial for users) and red cells indicate obfuscations that decrease battery life (i.e., changes that are detrimental to users).

Across all configurations, the change in battery life for the Nexus 3 ranges from $\approx -8.4\,\text{min}$ to $\approx 22.0\,\text{min}$ with a mean value of $\approx 2.5\,\text{min}$, a median value of $\approx 1.9\,\text{min}$, and a standard deviation of $\approx 4.6\,\text{min}$. The change in battery life for the Nexus 4 ranges from $\approx -16.3\,\text{min}$ to $\approx 10.9\,\text{min}$ with a mean value of $\approx -3.9\,\text{min}$, a median value of

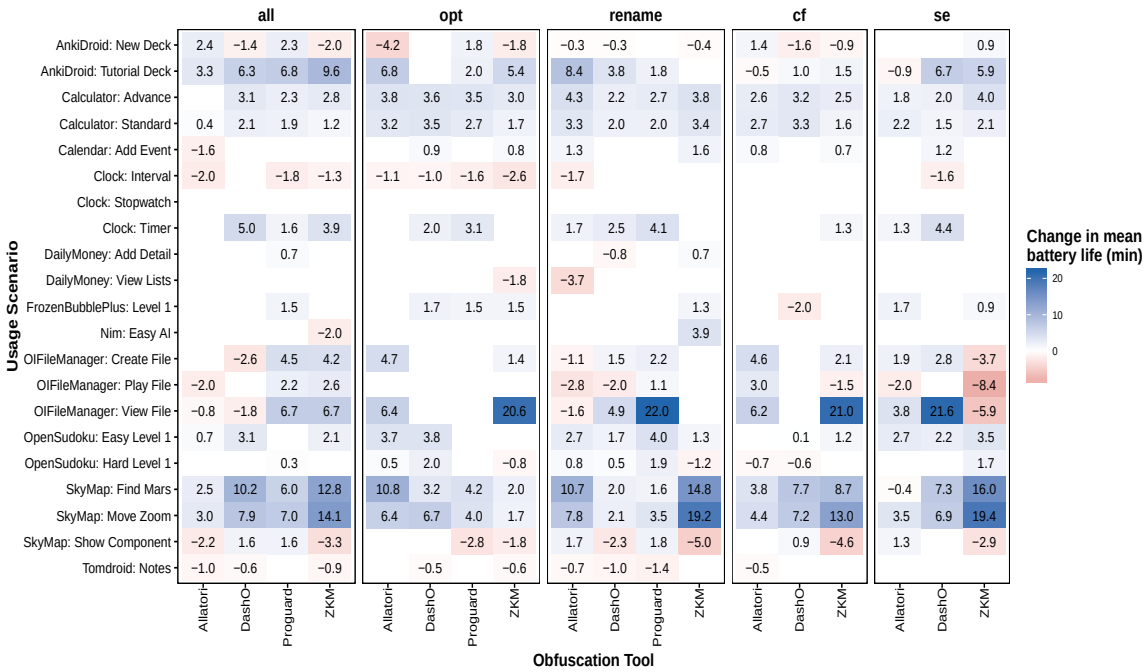


Figure 4.13a: Change in mean battery life when using an obfuscated version instead of an unobfuscated version when run on the *Nexus 3* platform.

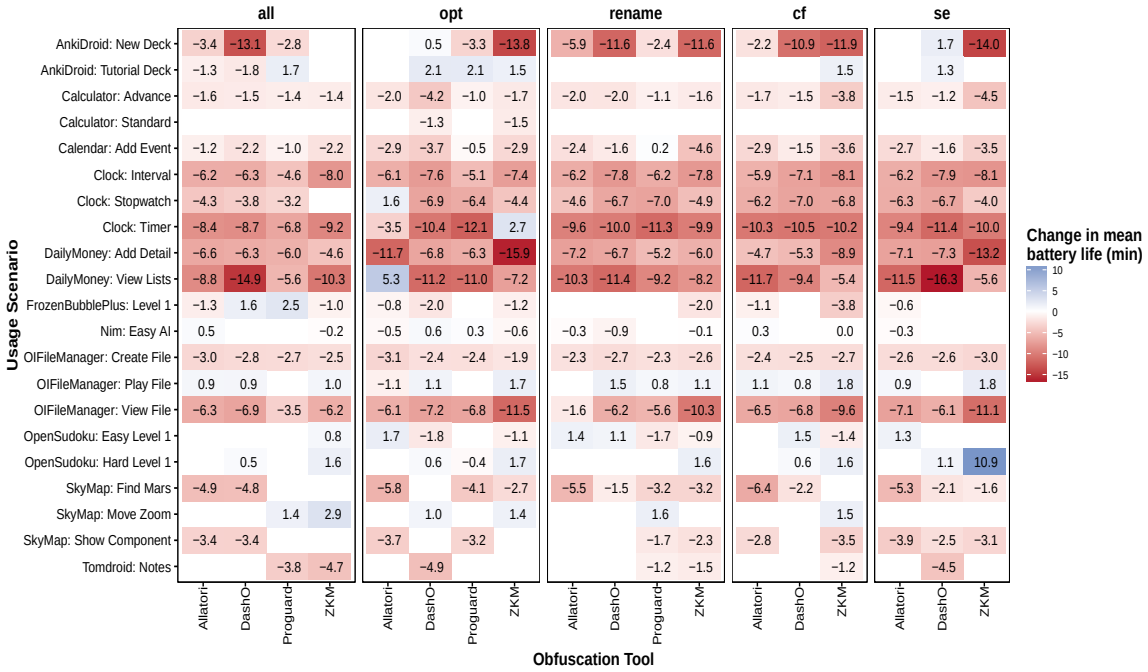


Figure 4.13b: Change in mean battery life when using an obfuscated version instead of an unobfuscated version when run on the *Nexus 4* platform.

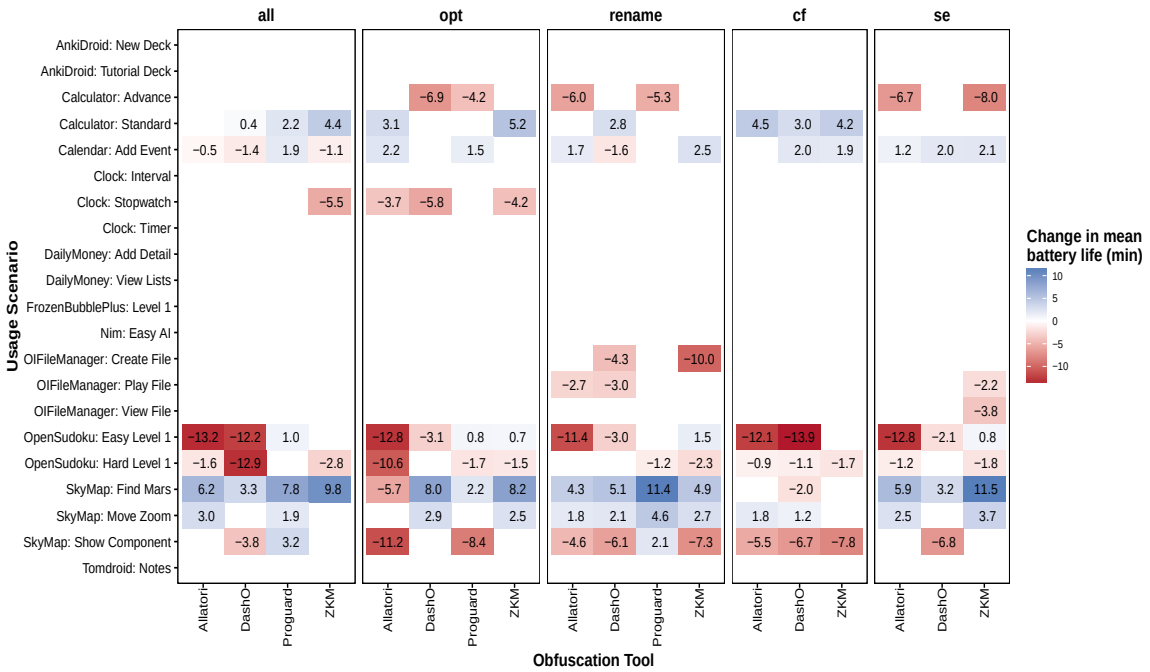


Figure 4.13c: Change in mean battery life when using an obfuscated version instead of an unobfuscated version when run on the *Galaxy S II* platform.

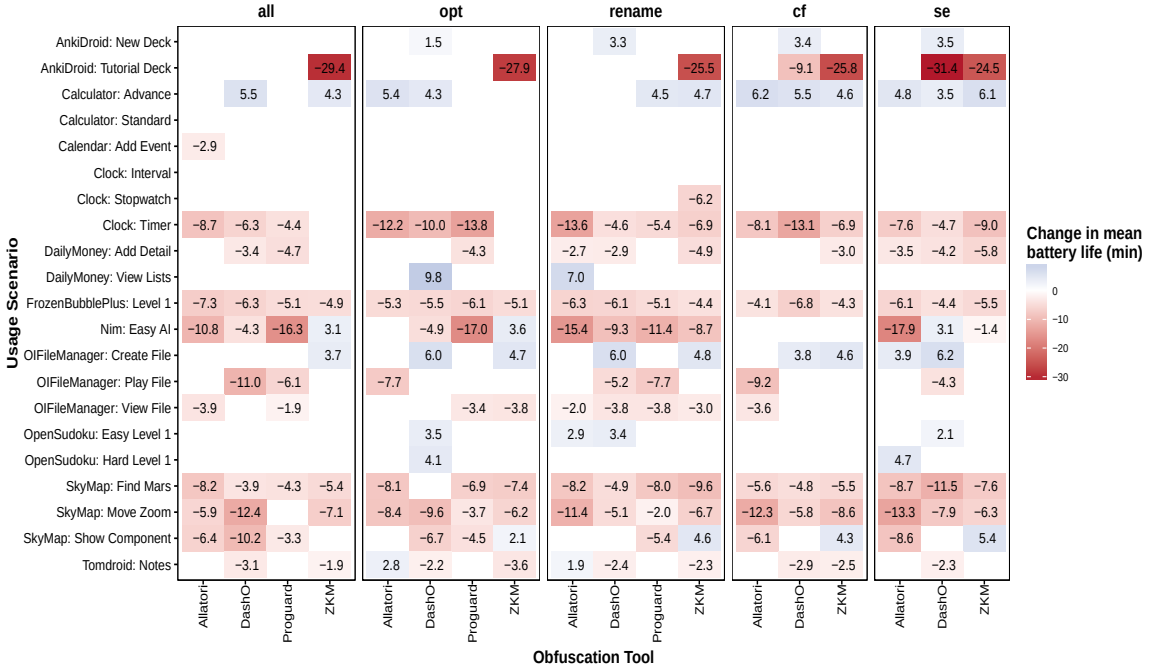


Figure 4.13d: Change in mean battery life when using an obfuscated version instead of an unobfuscated version when run on the *Galaxy S5* platform.

≈ -3.1 min, and a standard deviation of ≈ 4.2 min. The change in battery life for the Galaxy S II ranges from ≈ -13.9 min to ≈ 11.5 min with a mean value of ≈ -1.1 min, a median value of ≈ -0.9 min, and a standard deviation of ≈ 5.6 min and the change in battery life for the Galaxy S5 ranges from ≈ -31.4 min to ≈ 9.8 min with a mean value of ≈ -4.6 min, a median value of ≈ -4.9 min, and a standard deviation of ≈ 7.0 min.

When only the `all` configuration is considered, the change in battery life for the Nexus 3 ranges from ≈ -3.3 min to ≈ 14.1 min with a mean value of ≈ 2.4 min, a median value of ≈ 2.1 min, and a standard deviation of ≈ 3.9 min. The change in battery life for the Nexus 4 ranges from ≈ -14.9 min to ≈ 2.9 min with a mean value of ≈ -3.5 min, a median value of ≈ -3.2 min, and a standard deviation of ≈ 3.7 min. The change in battery life for the Galaxy S II ranges from ≈ -13.2 min to ≈ 9.8 min with a mean value of ≈ -0.4 min, a median value of ≈ 0.7 min, and a standard deviation of ≈ 6.2 min and the change in battery life for the Galaxy S5 ranges from ≈ -29.4 min to ≈ 5.5 min with a mean value of ≈ -5.7 min, a median value of ≈ -5.0 min, and a standard deviation of ≈ 6.2 min.

Based on these results, we believe that it is unlikely for an application user to notice a decrease in battery life due to an obfuscation. The observed changes in battery life range from ≈ -31.4 min to ≈ 22.0 min, which, even for the maximum and minimum, represents a change of less than 10 % of the respective phone's total battery life. Recall that these are the expected changes if the scenarios were executed continuously, draining the battery from full to empty. In practice, this is unlikely since mobile phone users rarely use an application continuously.

In retrospect, this result makes sense. For mobile applications, recent studies show that the majority of energy is consumed by the phone's screen, radios, and sensors [20, 68]. The changes made by the obfuscations do not change how the applications interact with or use these resources. Because the obfuscations make changes to parts of the application that do not consume much energy, the impacts of the obfuscations are overshadowed by the more energy expensive parts of the execution.

While users are likely to be indifferent to this conclusion because obfuscations

neither harm nor improve their battery life, it is good news for application developers. Now developers are able to protect their applications by applying obfuscations without needing to consider the obfuscation's impacts on energy usage.

4.5.3 Summary

In this section, we have presented an empirical study that investigated the impact of code obfuscations on the energy usage of mobile applications. We considered 11 commonly used Android applications, four obfuscation tools, five obfuscation configurations, 21 usage scenarios, and four platforms. In total, we ran 47 000 executions on our EMPs. The results of this study demonstrate that:

(1) Obfuscations can, and often do, impact the energy usage of applications with statistical significance.

(2) Obfuscations can both increase and decrease energy usage, but they are more likely to increase energy usage.

(3) The magnitude of the impacts of obfuscations are comparable to the magnitude of the impacts of other code level changes, such as applying refactorings.

(4) The differences between the impacts of the considered obfuscations on energy usage are not statistically significant.

(5) The impacts of obfuscation on battery life are unlikely to be meaningful to mobile application users.

4.6 Studies of Performance Tips

Recent studies have provided initial evidence that applying performance tips—best practices oriented towards runtime performance—is an effective mechanism for decreasing energy usage. More specifically, Li and Halfond [67], Tonini et al. [114], and Mundody and K [84] all report that applying performance tips can decrease energy usage from 10 % to 67 % for Android applications. This is promising because such tips are both easy to understand and easy to apply. In addition, these results support the common wisdom that applications can save energy by "racing to sleep"—speeding

up computation to allow the CPU to reach a low power state faster. These results also show that performance tips are related to energy code smells, where energy code smells are implementation choices at the source code level that cause higher energy consumption [116]. Consequently, performance tips are potentially more likely to be used in practice. However, these studies are limited in scope in several ways. The most severe of these limitations is that none of the existing studies evaluated the impacts of the performance tips when applied to real applications. Rather, they applied the performance tips to kernels or micro-benchmarks—small pieces of code that focus on the specific issue under study. While the targeted nature of kernels is beneficial, it remains unclear whether the observed results will transfer to real applications, which are characteristically larger and more complex.

To better understand the energy impacts of performance tips on Android applications, we investigated the energy impacts of applying four commonly recommended performance tips by creating a total of 32 modified versions of eight real Android applications. This study provides deeper insight into whether Android application developers can effectively reduce the energy consumption of their applications by applying performance tips.

4.6.1 Experiment-Specific Methodology

This section describes the details of our study design, including our independent and dependent variables; considered applications and scenarios; studied performance tips; and experimental procedure. In planning this work, we followed a methodology that is nearly identical to the one used in our prior work on investigating the impacts of code obfuscation on energy usage (Section 4.5.1)).

4.6.1.1 Experimental Variables

In this study, we considered one dependent variable, the amount of energy consumed by an execution, and two independent variables: (1) the performance tip applied to the application, and (2) the platform where the application executes.

To isolate the impacts of changing our independent variables on our dependent variable, it is necessary to precisely control how the applications are executed. We again chose to use RERAN as our capture/replay tool to prevent inconsistencies in executing the Android application (see Section 4.5.1.1). Since RERAN is designed to allow for the deterministic replay of a sequence of recorded events, any observed variations in energy usage are likely to be the result of the performance tips applied.

4.6.1.2 Considered Applications

We investigated the impacts of applying performance tips on popular, easily accessible Android applications. We selected Android applications for several reasons. First, as is the case for most software engineers, Android developers often care about the performance of their applications. As such, there are numerous performance tips that have been suggested for Android applications. Second, Android application developers typically care about the energy efficiency of their applications. Third, the source code of many Android applications is freely available, allowing us to easily modify the applications to apply the performance tips. Finally, we have extensive infrastructure to run Android applications and measure their energy usage.

Table 4.12: Considered applications.

Application	Description	LoC
Calculator	Android calculator	1427
Clock	Android clock	13 477
DailyMoney	Daily financial tracker	8723
Nim	Strategy game	1475
OIFileManager	File manager	7200
OpenSudoku	Sudoku game	6079
SkyMap	Astronomy application	10 921
Tomdroid	Note taking application	7955

Table 4.12 lists the specific applications that we used in this study. The first two columns, *Application* and *Description*, list the name of each application and a brief description of its functionality, respectively and the final column, *LoC*, shows

the application's number of lines of code. These specific applications were chosen in the same manner as described in the code obfuscation study. For example, they are representative of a wide variety of common application types, popular and widely used, and supported by RERAN.

4.6.1.3 Considered Usage Scenarios

To drive our user input driven applications, we examined each application and created one or more usage scenarios. In creating these scenarios, we focused on typical usage patterns for the application (i.e., actions that users are likely to perform). In this way, we were able to gain a better understanding of the impacts of applying performance tips on a user's daily interactions with their mobile device.

Table 4.13 shows the specific usage scenarios that we created. The first two columns, *Application* and *Name*, show the application that is used in the scenario and a distinguishing name, respectively. For example, Calculator has two scenarios, Calculator: Advance and Calculator: Standard. The third column, *Description*, provides a brief description of the user actions that are performed during the scenario. For example, during the Calculator: Advance scenario, several advanced arithmetic calculations are performed. The third column, *% Coverage*, shows the statement coverage for each scenario. To obtain the coverage information, we used Atlassian's Clover for the Android coverage tool (version 4.0). In total, we created 17 scenarios for our applications: three for Clock, OIFileManager and SkyMap; two for Calculator, DailyMoney, and OpenSudoku; and one for Nim and Tomdroid.

4.6.1.4 Studied Performance Tips

Performance tips that are studied in our study cover source code level implementation choices that can improve overall app performance. They are recommended by the Android Developers' web page particularly for Android apps written in Java [6].

To select the performance tips that we investigated, we first examined all the performance tips in the Android Developers' web page [6]. We then chose tips that

Table 4.13: Considered usage scenarios.

Application	Name	Description	Coverage (%)	Duration (s)	
				Galaxy S5	Nexus 4
Calculator	Advance	Perform several advanced arithmetic calculations.	50.4	74	84
	Standard	Perform several basic arithmetic calculations.	46.7	43	58
Clock	Interval	Create intervals while running the stopwatch.	19.6	64	65
	Stopwatch	Run the stopwatch for 10 seconds.	14.8	19	20
	Timer	Run a 10 second countdown timer.	21.1	20	19
DailyMoney	Add Detail	Enter two transactions.	28.5	75	74
	View Lists	View details and balances.	27.3	44	37
Nim	Easy AI	Play three rounds with increasing difficulty levels.	59.6	79	75
OIFileManager	Create File	Create 2 folders, nest folders, delete folders.	32.8	63	64
	Play File	View 4 pictures and play a ringtone 3 times.	27.8	58	52
	View File	Open a file, get details, and rename it.	27.4	64	60
OpenSudoku	Easy Level 1	Complete the first “easy” Sudoku grid.	35.4	172	273
	Hard Level 1	Complete the first “hard” Sudoku grid.	35.4	129	135
SkyMap	Find Mars	Set time to a fixed past date, searches for Mars.	55.7	58	42
	Move Zoom	Arbitrarily zoom in/out, moves along the map.	53.8	63	65
	Show Component	Show each component, toggle night mode.	48.8	101	105
Tomdroid	Notes	Create a note, search for text, open note, delete note.	19.0	131	173

were previously investigated in the literature, easily applicable to applications, and not specific to a particular application domain from that list. Since we are interested in identifying general trends about how the studied performance tips impact energy usage, tips that can only provide a single data point are not very useful.

We investigated the energy impacts of the following four performance tips on the Android applications:

* Tip 1: Use `final` for static constants. Declarations of String and primitive static (class) fields such as `static int intVal = 42;` result in the creation of a static initializer (`<clinit>`) that is executed when a class is loaded. Later, when these values are referenced, they are accessed using field lookups. Adding the final keyword to such declarations (e.g., `static final int intVal = 42;`) removes the need for the static initializer and eliminates the field lookups by replacing all references to the field with the declared value.

* Tip 2: Avoid Using Floating Point. In general, floating point operations are approximately twice as slow as their integer equivalents on most Android-based platforms [6]. Switching fields and local variables from floating point primitive types (i.e., `float` and `double`) to their integer primitive equivalents (i.e., `int` and `long`, respectively), where possible, can eliminate this unnecessary overhead.

* Tip 3: Avoid Internal Getters/Setters. Getters and Setters support encapsulation of a class's data. By preventing direct access to its fields, a class can more easily enforce constraints on its state. Unfortunately, method calls are significantly more expensive than field lookups. For example, accessing a field directly is typically between three and seven times faster than invoking a trivial getter [6]. Directly accessing fields (e.g., by inlining getters and setters) eliminates this overhead. In prior work, Li and Halfond [67] report that Tip 3 improved energy usage from 31 % to 35 %; Tonini et al. [114] report percentage improvements from 24 % to 27 %; and Mundody and K [84] report improvements from 17 % to 67 %.

* Tip 4: Avoid accessing array length in loop body. Currently, the Dalvik just-in-time compiler (JIT) is unable to optimize accesses to the length of an array or size of a collection during iterations of a loop. To avoid the cost of repeated accesses as the loop iterates, the array length (or collection size) should be cached across iterations. For example, the loop `for (int i = 0; i < a.length; ++i)` should be rewritten so that the value of `a.length` is stored in a local variable that then should be compared to `i` or, equivalently, an enhanced for loop (`for(Element e : Collection)`) can be used for collections that implement the Iterable interface. In prior work, Li and Halfond [67] and Mundody and K [84] found that Tip 4 improved energy usage 10 % and Tonini et al. [114] found that it improved energy usage from 36 % to 52 %.

4.6.1.5 Experimental Procedure

Figure 4.14 shows, at a high-level, the procedure we followed in this study, divided into four main steps: *Subject Creation*, *Replay-able Execution Creation*, *Data Collection*, and *Post Processing*. The remainder of this section describes these steps in detail.

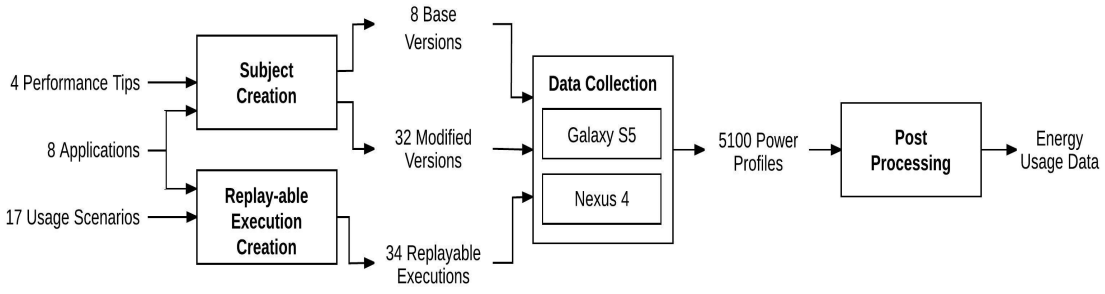


Figure 4.14: High-Level Experimental Procedure of Performance Tips.

Replay-able Execution Creation

The first step in our procedure is to create a set of replay-able executions. To create the replay-able executions, we manually performed the actions contained in

each scenario while using RERAN's recording tool. Because the replays produced by RERAN are not portable across mobile phone platforms, we created two replay-able executions for each scenario, one for each of our considered EMP platforms. This resulted in a total of 34 replay-able executions (17 scenarios $\times$ 2 platforms). The fourth and fifth columns of Table 4.13, *Galaxy S5* and *Nexus 4*, report the durations of the replay-able executions for the corresponding platform in seconds (s).

Subject Creation

The second step in our procedure is to create our set of experimental subjects. Because we are interested in the impacts of applying a performance tip to an application, our experimental subjects are versions of our considered applications with a performance tip applied. In total, we created 32 subjects (four performance tips applied to eight applications) by carrying out the following sequence of actions.

The first step is to create a suitable base version for each application. Sometimes the original versions of the applications are not suitable for this purpose because, in many cases, the performance tips have already been partially or completely applied. To create suitable base versions, we manually modified the original versions of the applications to undo any previously applied performance tips. This also has the benefit of establishing a best-case scenario for the performance tips; allowing them to be applied in as many places as possible gives them a greater chance to impact the energy usage of the applications.

Next, we created the modified versions of each application by exhaustively applying each performance tip to a fresh copy of the base version. To actually apply the performance tips, we manually edited the source code of each application, using the automated refactoring support available in Eclipse when applicable.

After creating each modified version, we manually examined the coverage information for each scenario to determine how many of the changes are covered by the scenario. Table 4.14 shows, for each performance tip, how many changes were covered by each scenario. The first two columns, *Application* and *Scenario*, show the

Table 4.14: Number of covered changes.

Application	Scenario	Tip 1	Tip 2	Tip 3	Tip 4
Calculator	Advance	20	5	18	4
	Standard	19	5	15	4
Clock	Interval	81	16	38	5
	Stopwatch	76	15	29	2
	Timer	93	20	38	8
DailyMoney	Add Detail	75	4	188	17
	View List	68	5	163	18
Nim	Easy AI	1	17	62	16
OIFileManager	Create File	56	18	64	10
	Play File	49	13	42	5
	View File	49	12	53	5
OpenSudoku	Easy Level 1	27	19	90	13
	Hard Level 1	27	19	90	13
SkyMap	Find Mars	135	57	133	50
	Move Zoom	130	52	137	48
	Show Component	121	50	118	46
Tomdroid	Notes	29	2	57	16

application that is used in the scenario and the scenarios, respectively. The remaining columns list the number of covered changes. For Tips 1 and 2, which modified potentially non-executable lines of code (i.e., variable declarations without an initial assignment), we checked whether a statement that uses the modified variable was covered. To do that, we used Eclipse's call hierarchy view feature. For Tips 3 and 4, which modify executable lines of code, we simply checked whether the modified lines were covered. In addition, to ensure that the changes did not introduce any behavioral differences, we verified that RERAN could accurately replay each execution on each application version by running RERAN with the replay-able execution as input and observing the replayed executions.

Data Collection

The third step in our procedure is to collect power usage data. This step is similar to how we collected power usage data in the code obfuscation study (see Section 4.5.1.6). The only difference is that we used RERAN to replay each replay-able execution on the corresponding EMP, using the base version and the four modified versions for each of the application's scenarios. For each EMP, each replay-able execution was executed on each version of the application (base and optimized separately for each performance tip) 30 times.

Note that we upgraded the version of Android running on the Galaxy S5 from 4.4 (Kit Kat) to 5.0 (Lollipop) before this study. Android 5.0 uses the newer Android runtime (ART) instead of the Dalvik runtime. While Dalvik is currently used by more Android versions including the version of Android running on the Nexus 4, ART will be the default for future Android versions. The main feature of ART compared to Dalvik is ahead-of-time (AOT) compilation which offers better performance than just-in-time (JIT) compilation.

In total, we ran 5100 executions—17 scenarios $\times$ (4 versions with performance tips applied + 1 base version) $\times$ 30 repetitions $\times$ 2 EMPs—which took 110 hours (over four days) of continuous execution time and resulted in over 3 GB of raw power consumption data.

Post Processing

The final step in our procedure is to post-process the collected data. We converted the power measurements to total energy consumption in joules in the same way as explained in the code obfuscation study (see Section 4.5.1.6).

4.6.2 Data Analysis and Discussion

We refined our overall question of whether applying performance tips can impact the energy usage of an application into the following specific research questions:

- *RQ1: Impact* — Do performance tips impact the energy usage of an application? If so, how?
- *RQ2: Importance* — Are the impacts of applying performance tips likely to be meaningful or noticeable to a typical mobile application user in terms of battery life?

The remainder of this section discusses the results of our study in terms of these research questions. Note that in answering these questions, we are analyzing the data for each platform separately. Because the replay-able executions are not identical, it would be inappropriate to analyze the impacts of the performance tips across platforms.

RQ1: Impact

To analyze the collected energy usage data, we performed Mann-Whitney-Wilcoxon (wilcox) tests to determine whether the difference between the amount of energy consumed by each scenario when run using the base version of the application and each modified version of the application is statistically significant. To check for statistical significance, we chose to use the Mann-Whitney-Wilcoxon test because we have one nominal variable (the performance tip applied to the application), one measurement value (the amount of energy consumed by the execution), and we do not know whether our data are normally distributed. We chose an alpha (α) of 0.05 and used R version 3.1.3's implementation of the test (i.e., `wilcox.test`).

For the cases where there is a statistically significant difference (i.e., $p \leq 0.05$), we computed Vargha and Delaney's $\hat{A}_{12}$ statistic to calculate the size of the effect of applying the performance tip (for more details about $\hat{A}_{12}$, see Section 4.5.2). For our data, $\hat{A}_{12}$ represents the probability that the *base* version consumes *more* energy than the *modified* version.

Figure 4.15 shows, for each platform, the $\hat{A}_{12}$ statistics that we calculated. In each facet, the y-axis shows the considered scenarios, and the x-axis shows each performance tip. The color of each cell indicates the size and direction of the effect.

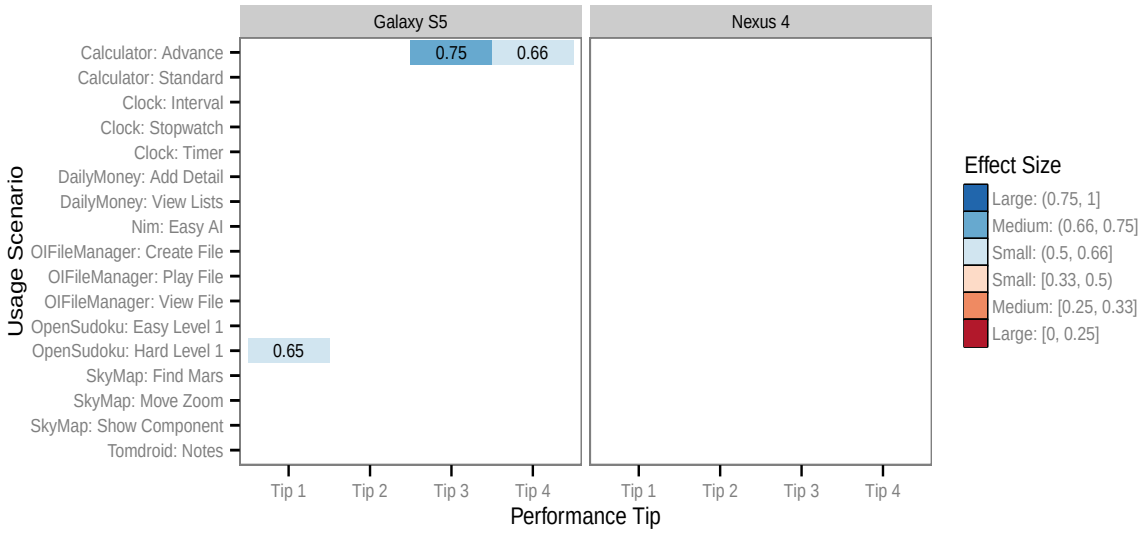


Figure 4.15: Vargha and Delaney's $\hat{A}_{12}$—probability that a base version consumes *more* energy than the modified version.

Cells colored blue indicate cases where the base version is more likely to consume *more* energy than the modified version (i.e., $\hat{A}_{12} > 0.5$) and cells that are colored red indicate cases where the base version is more likely to consume *less* energy than the modified version (i.e., $\hat{A}_{12} < 0.5$). In addition, the color's saturation indicates the size of the effect with the highest saturation indicating a "large" effect ($\hat{A}_{12}$ between 0.75 and 1.0 or between 0 and 0.25), a "medium" effect ($\hat{A}_{12}$ between 0.66 and 0.75 or between 0.25 and 0.33), or a "small" effect ($\hat{A}_{12}$ between 0.5 and 0.66 or between 0.33 and 0.5). Absent values indicate cases where there is not a statistically significant difference in energy usage between the versions.

From the data shown in Figure 4.15, it is clear that the performance tips are unlikely to impact the energy usage of real applications. Even though we provided a best-case scenario for the performance tips by creating base versions that allowed the tips to be applied in as many locations as possible, of the 136 wilcox tests that we conducted (17 scenarios × 4 performance tips × 2 platforms), only 3 (2 %) indicated a statistically significant difference in the amount of energy consumed by the base and modified versions. Moreover, the effect size of the performance tips was never large. As a point of comparison, our previous study on the energy impacts of code

obfuscations, which considered many of the same applications and scenarios, found that: the considered obfuscations had a statistically significant impact on energy usage $\approx 52\,\%$ of the time, and when there was a significant difference in energy usage, the size of the effect was "large" $73\,\%$ of the time, "medium" $26\,\%$ of the time, and "small" $1\,\%$ of the time (Section 4.5.2). Although the performance tips are making the same types of low level, localized changes as the obfuscations, they do not impact energy usage with the same frequency or size.

Overall, these results are not surprising although they are contrary to the common wisdom that, in order to save energy, applications should race to sleep. Unlike traditional desktop or data center software, which are often CPU bound, mobile applications are often much more interactive. In addition, the CPU is one of the least energy-expensive components. For mobile devices, the screen, radios, and sensors consume the majority of a device's battery. As a result, it is commonly the case that a larger proportion of energy is used when an application is idle, waiting for user input [68]. While the CPU can race to these idle periods, the energy-expensive components are still using large amounts of energy.

The nature of mobile applications also explains why the results that we observed are markedly different than those observed in prior investigations of the energy impacts of performance tips. The micro benchmarks are essentially traditional desktop software in that they are CPU bound. As soon as the benchmark is finished, the task is completed and power samples are no longer recorded. In that environment, racing to sleep makes sense and explains why prior studies observed substantial reductions in energy usage.

RQ2: Importance

To answer this question, we computed the change in battery life that a user could expect if they were to use a modified version of an application instead of the base version. In the same way as in Section 4.5.2, we used Equation 4.3 to calculate, for each platform, the percentage of battery charge that is consumed by each scenario

when it is executed using the base version of its application and when it is executed using the modified versions of its application. As a reminder, for the Nexus 4, $V = 3.8\,\text{V}, C = 2100\,\text{mA h}$; and for the Galaxy S5, $V = 3.8\,\text{V}, C = 2800\,\text{mA h}$.

$$\%_{charge} = \frac{E}{V} \times \frac{1000}{C \times 3600} \times 100 \tag{4.3}$$

We then calculated, using Equation 4.4, the amount of time needed to drain each platform's battery from full to empty (i.e., battery life) if the scenario were executed continuously using each version of its application. The duration of the scenario, D, is presented in Table 4.13.

$$t_{drain} = \frac{100\,\%}{\%_{charge}} \times D \tag{4.4}$$

Table 4.15 shows the results of this computation for the scenarios where there was a statistically significant change in energy usage. In the table, the first two columns, *Application* and *Name*, show the scenario and the remaining columns, *Galaxy S5* and *Nexus 4*, the mean battery life in hours (h) when the base version is run continuously, draining the battery from full to empty.

Table 4.15: Battery life when using a base version.

		Battery life (h)	
Application	Name	Galaxy S5	Nexus 4
Calculator	Advance	14.4	5.7
OpenSudoku	Hard Level 1	10.2	5.6

Finally, we computed the changes in battery life for each scenario and performance tip by subtracting the battery life of each modified version from the battery life of the base version. Figure 4.16 shows the results of these computations. The layout of this figure is similar to the layout of Figure 4.15: it is grouped by platform, the y-axis shows the usage scenarios, and the x-axis shows the performances tips. The content of each cell shows the change in battery life in minutes (min). Again, the color

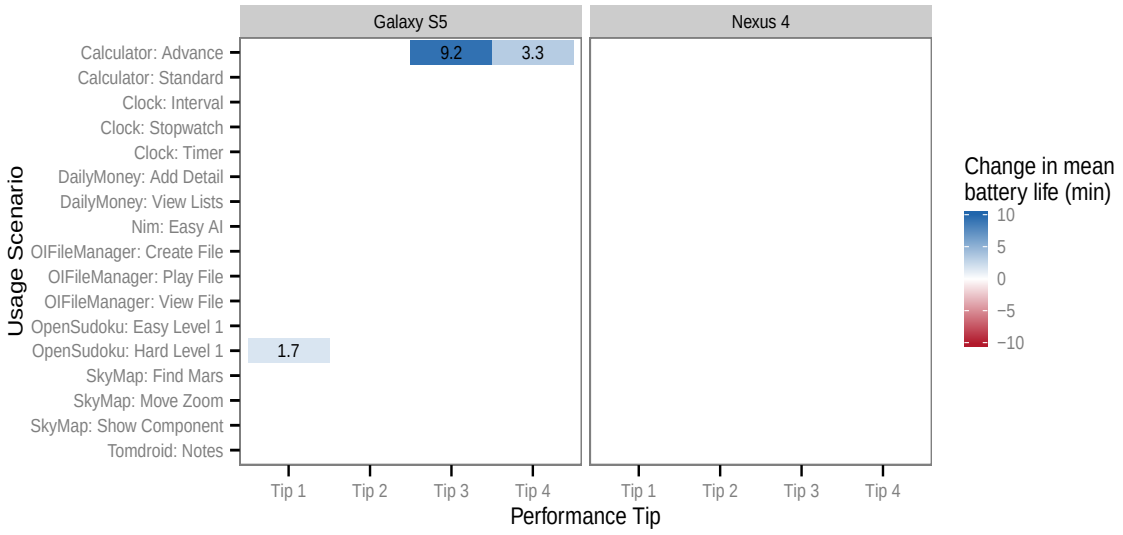


Figure 4.16: Change in mean battery life when using a modified version instead of a base version.

of each cell indicates the direction and magnitude of the change. Blue cells indicate cases where battery life is increased (i.e., changes that are beneficial for users); red cells indicate cases where battery life decreased (i.e., changes that are detrimental to users), which did not occur in our data; darker colors indicate larger values; and absent values indicate cases where there is not a statistically significant difference in energy usage.

From the data shown in Table 4.15 and Figure 4.16, it is clear that, even in the unlikely case when the performance tips cause a statistically significant difference in energy usage, the impacts of the changes are unlikely to be noticed by typical mobile application users. For the scenario with the largest change, Calculator: Advance run on the Galaxy S5, the percentage change in mean battery life is $\approx 1\,\%$ (864 min for the base version compared to 873 min for the modified version).

4.6.3 Summary

In this section, we have presented an empirical study that investigated the impact of commonly recommended performance tips on the energy usage of mobile applications. We considered eight commonly used Android applications, four performance tips, 17 usage scenarios, and two platforms. In total, we ran 5100 executions on our

EMPs. The results of this study demonstrate that:

(1) Despite initial evidence to the contrary, considered performance tips that are commonly recommended for Android applications are unlikely to impact the energy usage in a statistically significant manner.

(2) Even in the unlikely event that a performance tip changes energy usage in a statistically significant manner, the impact of the performance tip on battery life is negligible.

4.7 Related Work

This section describes the prior and more recent related work in investigating the energy impacts of software engineering decisions that we considered in our studies.

4.7.1 Design Patterns

Prior to our work, the impacts of design patterns on energy usage were explored by Litke et al. [71]. In their study, they examined three patterns (factory method, observer, and adapter) and observed an increase in energy usage caused by the observer and factory method patterns. However, their study is preliminary in nature and the relation between design pattern and energy consumption is speculative rather than based on empirical evidence.

In addition to our design pattern study, researchers have continued to investigate energy impacts of design patterns by comparing energy consumption of applications using design patterns against applications not using design patters. For example, Bunse and Stiemer [17] compared the energy consumption of six design patterns (facade, abstract factory, observer, decorator, prototype, and template method) on Android-based mobile phones. They executed small Android applications and measured energy consumption via the PowerTutor app. Their results agree that some design patterns have relatively high impacts on the energy consumption while some design patters have small impacts. For example, decorator design pattern increased the energy consumption of the application $\approx 134\,\%$.

Noureddine and Rajan [88] examined energy impacts of 21 design patterns including design patterns that we considered in our study. They executed the applications on Lenovo Thinkpad X220 laptop, estimated the energy consumptions by using Jolinar 2, and measured the CPU energy overhead (positive or negative) for each design pattern. Their findings support our results, suggesting that applying design patterns can decrease (6 out of 21) or increase (15 out of 21) energy consumption, and the impacts of applying design patterns can vary greatly (up to $\approx 30\,\%$).

4.7.2 Code Refactorings

The prior study conducted by Silva et al. [26] and more recent studies [98, 100, 89] indicate that applying code refactorings can have impact on the energy usage of an application.

More specifically, Silva et al. [26] measured the performance and energy impacts of inlining methods on three embedded Java applications (an address book, a game called Sokoban, and an MP3 audio decoder). While inlining decreased energy consumption of the address book and Sokoban applications, it had the opposite effect on the MP3 decoder, the most complex of the considered applications. The results of our study agree with their observations; inlining methods can increase energy usage in some instances while decreasing it in others.

Since our work, Park et al. [89] investigated energy impacts of 63 out of the 68 code refactoring techniques defined by Fowler [1]. For each refactoring technique, they estimated power consumption of the original and refactored versions of the sample `C++` code with XEEMU power estimation tool. The results of their study demonstrate that code refactoring techniques may increase (30 techniques), decrease (26 techniques), or not change (7 techniques) energy consumption. Although these results are not based on empirical observations, they confirm that code refactoring techniques have the potential to impact energy usage.

In object-oriented application development, particular patterns that negatively impact an application quality in terms of nonfunctional attributes are called code

smells [1]. To eliminate code smells, developers typically apply refactorings. For example, the God Class and Brain Method code smells are eliminated by applying Extract Class and Extract Method refactorings, respectively. However, applying refactorings may have detrimental impacts on the application's energy consumption as we showed in our work. Similarly, Prez-Castillo and Piattini [98] and Rodriguez et al. [100] suggested that applying Extract Class and Extract Method refactorings to eliminate the God Class and Brain Method code smells can lead to higher energy consumption due to an increase of object creations and message exchanges.

4.7.3 Performance Tips

Prior studies indicate that applying performance tips—best practices oriented towards runtime performance—can decrease energy usage from 10 % to 67 % for Android applications [67, 114, 84]. These studies investigated the impacts of the performance tips *Tip 3* and *Tip 4* (see Section 4.6.1.4) on kernels or micro-benchmarks. In their experiments, Li and Halfond [67] used one mobile device and measured energy consumption via a Monsoon power meter, which is similar to our energy measurement platforms. Tonini et al. [114] and Mundody and K [84] used three mobile devices and an emulator, respectively. Both of them measured energy consumption via the Power Tutor app. Both the mobile devices and the emulator run Android version 4.2.2 or older which uses the Dalvik runtime. Our study is different from those prior studies in several ways. First, performance tips are applied to real applications instead of small pieces of code that focus on the specific issue. Second, two additional commonly suggested performance tips are examined. Third, our mobile devices run on newer Android versions, Android 4.3 and 5.0. Lastly, ART runtime, which will be the default for future Android versions, is also considered in our work as well as the Dalvik runtime.

Chapter 5

PREDICTION OF ENERGY TESTING REQUIREMENTS

To satisfy the demands and expectations of users, developers must consider the energy efficiency of their applications. Therefore, in addition to allocating resources to test for traditional requirements like correctness, developers must also allocate resources to conduct energy testing to check their applications for energy consumption issues.

Unfortunately, energy testing is often more expensive than traditional types of testing that aim to verify and validate an application's correctness. The costs of energy testing are due to its characteristic features. First, to collect accurate energy usage data during an execution, energy tests should be run on real devices with specialized energy measurement hardware. Second, since the low sampling rates of energy measurement hardware may affect the measurements, energy tests must have long running times. Finally, energy tests have to be performed for each supported platform, which, in the case of mobile applications, can be a significant number of devices.

The high costs of energy testing can adversely impact the planning process of application evolution. Similar to traditional testing, energy testing should be performed in response to code changes, which occur frequently in mobile applications. Currently, developers plan code changes without knowing the energy test requirements of the changes. To detect and prevent energy issues as early as possible, they must anticipate conducting energy testing after each change by running all energy tests. However, a proposed code change might not require all energy tests to be run or might not even require any energy testing at all. This lack of information prevents developers from making decisions on code changes such as ordering, postponing, or canceling them. The majority of existing work on energy testing has focused on minimizing test suites with respect to their energy consumption, finding energy bugs, and reducing the energy

consumption of test suites [56, 69, 11, 61]. To the best of our knowledge, none of the existing work focuses on identifying the amount of energy testing required for proposed changes.

In this chapter, we present a technique, Energy Retest Umpire (ERU), that provides feedback on the energy test requirements of proposed code changes. This feedback can help developers plan for changes and allocate testing resources. Basically, ERU informs developers about whether energy testing is required for a proposed code change before they actually make the change. It also identifies what energy tests need to be executed when energy testing is necessary. At a high level, ERU leverages change impact analysis and pre-computed API energy usage information.

To evaluate ERU, we implemented a prototype for Android applications. Using the prototype, we performed a preliminary study on ten freely-available, open source Android applications. The goal of the study is to investigate the feasibility of ERU and how it performs when changes are expressed at differing granularity levels.

The remainder of this chapter is organized as follows: Section 5.1 introduces necessary background information about energy testing; Section 5.2 presents a motivating example; Sections 5.3 and 5.4 describe the technique and prototype implementation, respectively; and Section 5.5 discusses the evaluation of the technique.

5.1 Background

Software bugs that lead to energy inefficiencies in applications are known as energy bugs. These energy bugs cause excessive battery drain, which is a main user complaint about applications. Different types of energy bugs can be found in mobile applications including resource overuse and misuse bugs, no-sleep bugs, sleep conflicts bugs, loop bugs, immortality bugs, activity bugs, and event bugs [90, 52, 11]. For example, no-sleep bugs keep at least one component of the mobile device awake erroneously. This prevents the mobile device from going to a lower power state and increases the battery drain significantly.

To combat energy issues in applications, developers need to perform energy testing. Energy testing differs from traditional testing in several ways. In particular, it often requires developers to execute energy tests on real devices and collect energy usage data by using a special energy measurement device. In general, the energy testing process works as follows. Developers install the application under test on a real device. Then, they run the application and execute energy tests while energy usage data is collected by the energy measurement device. The collected data is analyzed to detect energy issues. This manual, labor intensive process is then repeated several times for each supported platform.

Functional tests that are designed to test correctness of the applications are generally not usable as energy tests for several reasons. First, energy tests should have long running times, due to the low sampling rates of energy measurement hardware. Second, they should focus on energy bugs and features of the application that use energy-greedy hardware components in the device to detect energy issues. In addition, energy tests need to be user interaction scenarios for mobile applications since they are interactive and event-driven. Developers can generate deterministic scenarios manually or generate random scenarios via an automatic event generator (e.g., Monkey [81]). These generated scenarios are recorded and reproduced by using capture/replay tools (e.g., RERAN [40]). Manually reproducing scenarios can influence energy consumption because developers can perform the same sequence of actions (e.g., enter text into a textbox or click a button) but cannot maintain the same timing between the actions.

Since energy tests are user interaction scenarios for mobile applications, each energy test has different event and action ordering with a focus on some application features. For example, adding a note and deleting a note might be two different energy tests for Tomdroid, a note-taking application. Because energy tests focus on application features instead of individual source code units of the application, executing all energy tests that cover the modified source code unit is needed to obtain good accuracy in energy testing.

To accomplish high accuracy in energy measurement, special energy measurement devices are used. These devices measure actual power consumption externally and do not introduce any measurement overhead. In Section 3.3.3, we have presented two custom-built energy measurement platforms that are appropriate for energy testing of Android applications.

5.2 Motivating Scenario

Developers frequently need to evolve their applications to add new features, enhance or adjust existing features, fix bugs, meet new requirements, improve performance, restructure the source code, etc. The process of evolving an application most likely involves code changes and testing in response to those code changes. When developers make code changes, they perform traditional and energy testing since both correctness and energy-efficiency of the application are important. Due to lack of information about the energy testing requirements of the proposed code changes, developers plan to conduct energy testing after each change and run all energy tests. This might negatively affect the application evolution timeline and restrict the number of changes that developers plan to include in a release.

Knowing the energy test requirements of proposed code changes can help developers to develop a realistic and effective application evolution timeline. For example, some changes may not require energy testing, and thus the allocated time for energy testing can be used to add more changes in a release. When developers are up against a deadline and have to decide what code changes are made in this release, they may postpone code changes that require energy tests to a later release. They can even decide to cancel changes that require enormous amount of energy testing efforts. As a hardware-based testing environment is needed to perform energy testing, developers can order the changes to reduce setup costs. In addition, there are usually various ways to implement a feature or functionality. Comparing the energy test requirements of different implementations can also help developers to decide which choice to apply.

5.3 Approach: Energy Retest Umpire (ERU)

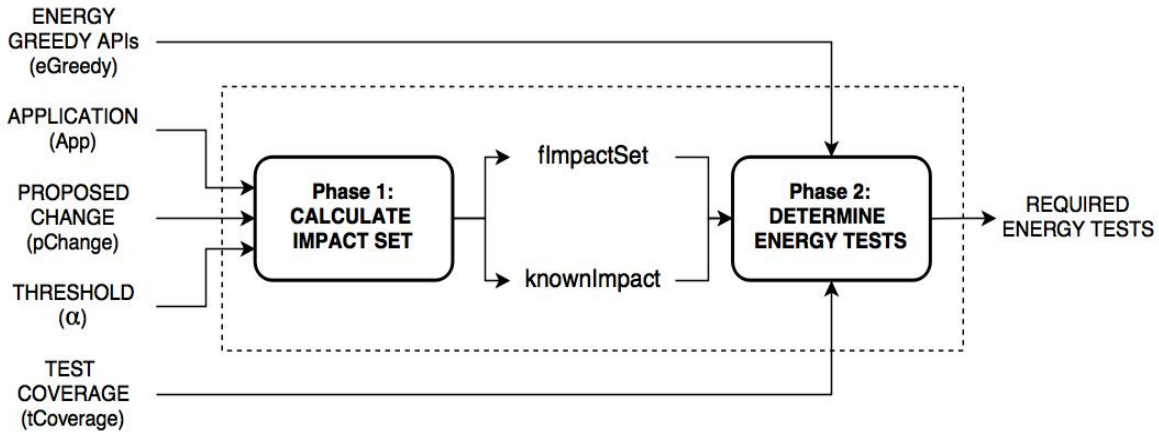


Figure 5.1: High-level overview of Energy Retest Umpire Technique.

This section presents our approach (ERU) for providing feedback on the energy test requirements of proposed code changes to developers. Figure 5.1 shows a high-level view of ERU. The main insight behind ERU is that energy test requirements of a proposed change depend on the energy greedy API usage of the proposed change and the source code units impacted by the change. ERU takes the following as input:

* Application (App): Source code of an application.

* Proposed Change (pChange): Potential change that the developer plans to make to the input application. It can be expressed at different source code unit granularities (e.g., package, file, method, etc.).

* Energy Greedy APIs (eGreedy): API methods that are known, a priori, to consume significant amounts of energy. ERU uses a standard set, but developers can also provide additional methods.

* Test Coverage (tCoverage): Coverage information indicating which application source code units (expressed at the same granularity as the proposed changes) are executed by each energy test. For example, at the method level $t_1 \mapsto \{m_1, m_2, m_3\}$ indicates that energy test 1 covers methods 1, 2, and 3.

* Threshold (α): Confidence threshold used to filter the impact set of the proposed change (see Section 5.3.2).

ERU produces the following as output:

* Required Energy Tests: A list of energy tests that need to be run for the proposed change. In general, there are three possible outcomes.
  - No Energy Tests: The proposed change does not require any energy tests to be run (i.e., the list of tests is empty).
  - Partial Energy Tests: The proposed change requires some energy tests be run (i.e., the list contains some, but not all, energy tests).
  - All Energy Tests: The proposed change requires all energy tests to be run (i.e., the list contains all energy tests).

To decide whether energy testing is required, and if so, which energy tests need to be run, ERU is divided into two main phases: the *Calculate Impact Set* phase and the *Determine Energy Tests* phase.

The first phase, *Calculate Impact Set*, identifies the potential impacts of the proposed change on the correctness of other source code units, requiring that they be changed to maintain correctness. It takes as input the application, proposed change, and threshold and generates *knownImpact*—a flag that indicates whether the impact of the proposed change is known and *fImpactSet*—the source code units that will likely be changed along with the proposed change for the given threshold. These source code units are at the same granularity with respect to the proposed change. For example, if the proposed change is an application method, then its *fImpactSet* contains the potentially impacted application methods, which are the co-changes of the proposed change.

The second phase, *Determine Energy Tests*, takes as input the outputs of the first phase as well as the Energy Greedy APIs (eGreedy) and Test Coverage (tCoverage)

information. It uses these inputs to determine the energy test requirements of the proposed change.

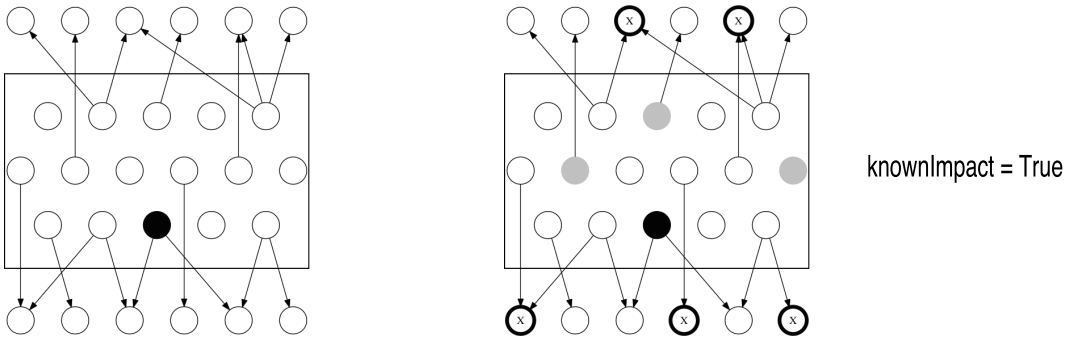
5.3.1 Example Scenarios

As intuitive examples of how ERU determines the energy test requirements of a proposed change, consider the scenarios shown in Figures 5.2a, 5.2b, and 5.2c. These scenarios represent the possible outputs of ERU.

In each figure, the left subfigure, (a), shows the API method calls of the application's source code units such as application's method, files, classes, etc. Nodes inside the rectangle represent application source code units and nodes outside the rectangle represent API methods. The application source code unit where the developer plans to make code changes (pChange) is solid black. The right subfigure, (b), illustrates impacted application source code units (fImpactSet), energy greedy API methods (eGreedy), and *knownImpact* flag computed by *Calculate Impact Set* and the *Determine Energy Tests* phases. Source code units in fImpactSet are lightly shaded if fImpactSet $\neq \emptyset$. API methods that are marked with a cross ('X') show the eGreedy methods that consume high energy. They can influence application energy consumption.

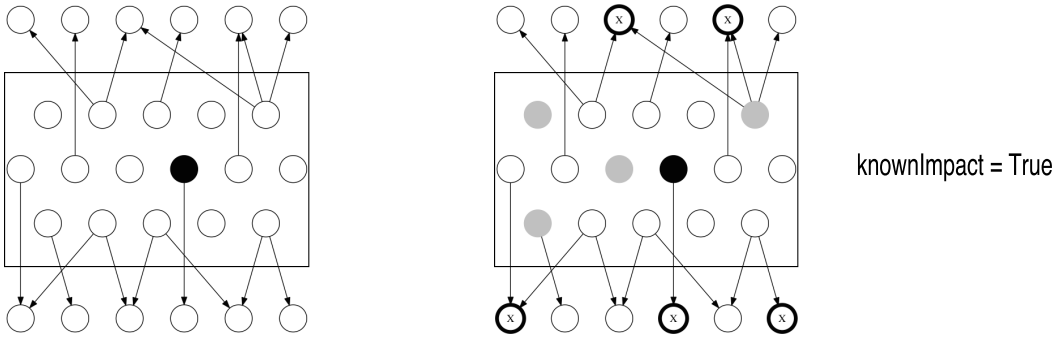
Figure 5.2a shows the scenario where pChange does not require energy testing. The impact of pChange is known as the *knownImpact* flag is set to true. It means that, depending on the use of eGreedy methods, energy testing may or may not be required. In this case, since none of the application source code units in fImpactSet and pChange call any of the eGreedy methods, no energy tests needed to be run for pChange.

Similarly to Figure 5.2a, the impact of the pChange is known in Figure 5.2b. The difference is that some of the source code units in fImpactSet and pChange call eGreedy methods. Therefore, energy testing is required and the energy tests that cover any of these application source code units should be run for pChange. Note that, depending on the specific coverage of the tests, they may or may not all need to be run.



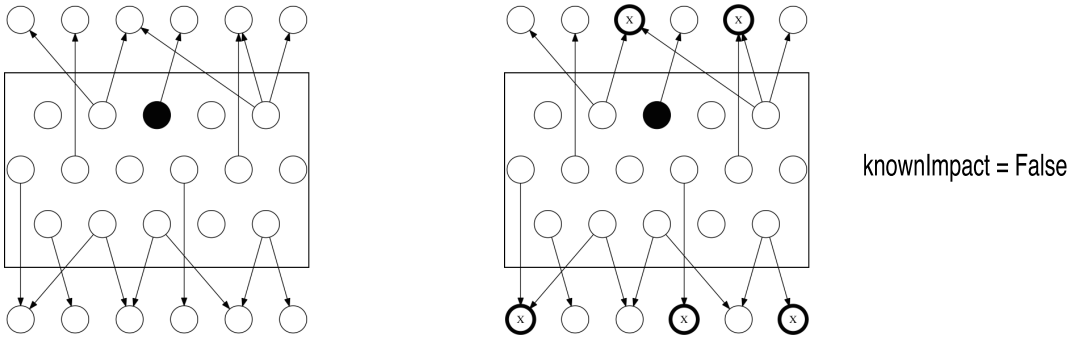
(a) Proposed change (b) After phases

Figure 5.2a: No energy tests scenario.



(c) Proposed change (d) After phases

Figure 5.2b: Partial or all energy tests scenario.



(e) Proposed change (f) After phases

Figure 5.2c: All energy tests scenario.

Finally in Figure 5.2c, the *knownImpact* flag is false because the impact of the proposed change is unknown. In this case, all energy tests should be run for pChange whether pChange uses eGreedy methods or not.

5.3.2 Phase 1: Calculate Impact Set

The goal of the *Calculate Impact Set* phase is to identify the impacts of the proposed change on other application source code units. One way to achieve this goal is to use change impact analysis [14]. Change impact analysis computes the impact set of the proposed change, formulated as a set of pairs $< n, p >$, where each pair is composed of a source code unit name n and the probability p of the source code unit being impacted by the proposed change. The probability ranges from 0 to 1, where 1 is the highest probability of being impacted.

The *Calculate Impact Set* phase outputs a flag called *knownImpact* that indicates whether the impact set is empty because no other changes are anticipated or because there is insufficient information to make a prediction. The flag is set to false when there is insufficient information. The other output of the *Calculate Impact Set* phase, *fImpactSet*, is a filtered version of the impact set that only includes pairs where the probability is greater than or equal to the selected threshold (α). Source code units in *fImpactSet* are considered co-changes of the proposed change that will be changed along with the proposed change. We also add the proposed change into its *fImpactSet* with the probability of 1. For example, assume A is the proposed change, and B and C are impacted with probabilities of 0.9 and 0.8, respectively. The impact set of A is $(< B, 0.9 >, < C, 0.8 >)$. Then, the *fImpactSet* of the proposed change becomes $(< A, 1.0 >, < B, 0.9 >)$ for threshold 0.85 and $(< A, 1.0 >, < B, 0.9 >, < C, 0.8 >)$ for threshold 0.8.

5.3.3 Phase 2: Determine Energy Tests

Algorithm 1 shows how ERU determines the energy test requirements of a proposed change in the *Determine Energy Tests* phase.

Algorithm 1 Determine Energy Tests

Input: fImpactSet - Filtered impact set.
Input: eGreedy - Set of methods of interest.
Input: tCoverage - Desired coverage of energy tests.
Input: knownImpact - Flag indicating whether the proposed change impact is known.
Output: Required Energy Tests

```
1: procedure DETERMINE ENERGY TESTS(fImpactSet, eGreedy, tCoverage, knownImpact)
2:     List RequiredEnergyTests
3:     if knownImpact == False then
4:         RequiredEnergyTests ←all energy tests
5:     else
6:         for source code unit ϵ fImpactSet do
7:             for callee in Callees(source code unit) do
8:                 if callee ϵ eGreedy then
9:                     for energy test ϵ tCoverage do
10:                        if source code unit ⊂ energy test then
11:                            RequiredEnergyTests ←energy test
12:                        end if
13:                    end for
14:                    break
15:                end if
16:            end for
17:        end for
18:    end if
19:    return RequiredEnergyTests
20: end procedure
```

This phase takes as input the fImpactSet, eGreedy, tCoverage, and knownImpact. It first checks the knownImpact flag, and adds all energy tests to *RequiredEnergyTests* if the knownImpact flag is False, which means there is insufficient information. If not, all *callee* API methods of each *source code unit* in fImpactSet are examined. When any of the *callee* API methods is in eGreedy, energy tests in tCoverage that cover the *source code unit* are added to *RequiredEnergyTests*. This process continues until all source code units in fImpactSet are inspected. Finally, *RequiredEnergyTests*, which consists of energy tests that should be run for testing energy impact of the proposed change is returned as output.

5.4 Implementation

This section describes the prototype implementation of ERU designed for Android applications. Since energy greedy APIs (eGreedy) is a necessary input for ERU, we used pre-computed Android API energy usage information to obtain this input. To generate the impact set of a proposed change (pChange) in *Calculate Impact Set* phase, the prototype leverages Historical Change Impact Analysis. Additionally, identifying the callees of the Android application source code units is needed in the *Determine Energy Tests* phase to determine required energy tests. Therefore, we integrated *DependencyFinder* into *Determine Energy Tests* phase [31]. We provide more details about the implementation in following subsections.

5.4.1 Energy Greedy APIs

The set of energy greedy APIs that we considered in our prototype is derived from two sources. First, we used the results of the findings of an empirical study conducted to find the most energy-greedy Android API methods [70]. This study examined the energy consumption of 55 Android apps from different domain categories by using real-usage scenarios and analyzed 807 Android API methods. Based on the results of their study, they categorized 131 methods as energy greedy.

In addition, we also considered recent studies that show *wakelock*, *GPS*, and *GSM* related APIs can cause battery drain because of either misuses of APIs such as *wakelock* APIs or uses of energy greedy hardware components such as *GPS* and *GSM* [20, 92, 91, 120]. However, these studies did not investigate API methods individually, therefore we added *wakelock*, *GPS*, and *GSM* API packages with all of their methods to our *eGreedy* set.

5.4.2 Change Impact Analysis

Software change impact analysis (CIA) estimates co-changes that need to be made to accomplish a change [14]. Change impact analysis approaches use different

scopes, including source code, formal models, and miscellaneous artifacts [65]. Source-code based CIA approaches focus mainly on identifying the part of the code that needs to be modified along with the proposed change. These approaches can be static or dynamic [66]. Dynamic CIA techniques require execution of the source code to collect information after a change while static CIA techniques analyze information about the source code before a change. Source-code based CIA techniques generate impact sets at different granularity levels such as file, class, method, field, or statement. Further, both dynamic and static techniques can be divided into subtypes. Dynamic CIA can use either offline or online analyses. Static CIA can perform historical, textual, or structural static analyses.

In our prototype implementation, we chose to use historical CIA. Historical change impact analysis (HCIA) is a static, source-code based analysis that extracts co-change couplings by mining changes in the source code repository of an application. HCIA computes a confidence value with the predicted changes. We chose to use HCIA in our implementation for several reasons. First, HCIA is the most used technique among the source-code based change impact analysis approaches [66]. Second, it does not require actually applying changes and executing application source-code to generate an impact set. Finally, HCIA can generate an impact set at different granularity levels as a set of pairs composed of source code unit name and its probability of being impacted ($< n, p >$).

There are several ways to compute the probability of a source code unit being impacted. For example, the association rule shown in Equation 5.1, proposed by Zimmermann et al. [121], can be used to compute the probability.

$$P(A \rightarrow B) = \frac{N(A \cap B)}{N(A)} \tag{5.1}$$

In Equation 5.1, A and B are source code units in an application. $N(A \cap B)$ represents the *support value*, which is the number of times A and B have been changed together in the source code repository of the application. $N(A)$ represents the *frequency* of A, which is the number of times A has been changed. $P(A \rightarrow B)$ denotes the

probability that B will be changed if A is changed and is also called the *confidence value*.

Although Equation 5.1 is effective, it assumes that the support value term only includes intentional co-changes, that is, co-changes where A and B are modified together for the same reason. In many cases, this assumption may not hold. For example, A and B may be related to separate features that are included in the same release. For instance, assume that $N(A)$=1, $N(B)$=10, and $N(A \cap B) = 1$. If A is the location of the proposed change, $P(A \rightarrow B) = 1$ which means that the change in A impacts B and B also needs to be changed. However, it is possible that A and B have been changed together incidentally in the past since B has been changed many times. As a result, changing A might not necessitate changing B. To avoid such bias, Equation 5.1 can be modified to require a minimum support value (i.e., only co-changes that occur a sufficient number of times are considered) as shown in Equation 5.2.

$$P(A \rightarrow B) = \begin{cases} \dfrac{N(A \cap B)}{N(A)} & N(A \cap B) \geq MinSuppValue \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

A second method for calculating probabilities, proposed by Jashki et al. [58], is to use the Jaccard similarity coefficient [57]. This method is shown in Equation 5.3.

$$P(A \rightarrow B) = \frac{N(A \cap B)}{N(A) + N(B) - N(A \cap B)} \tag{5.3}$$

In Equation 5.3, $N(A)$ and $N(B)$ again represent the number of times that these source code units have been changed in the source code repository of the application. The support value is divided by the frequency of A plus the frequency of B minus the support value. This equation considers the closeness of the methods to calculate the probability.

In our prototype implementation, developers are easily able to select any of the three equations or define any minimum support value for the probability calculation.

5.4.3 Identifying the Callees of the Source Code Units

Almost all Android applications are Java-based. Thus, we use DependencyFinder in our prototype to identify the callees of the Android application source code units [31]. The callee relationships are used to specify whether a source code unit calls any of the eGreedy methods. The use of eGreedy methods necessitates energy testing for the source code unit.

DependencyFinder analyzes compiled Java codes and builds dependency graphs. In the dependency graph, a dependency indicates that one element requires the presence of another element to function. DependencyFinder has 3 different element types: package, class, and feature.

Table 5.1: Dependencies.

Elements	`Package`	`Class`	`Feature`
Package	✓	✓	✓
Class	✓	✓	✓
Feature	✓	✓	✓

Table 5.1 shows dependencies that DependencyFinder can infer with a checkmark (✓). As the table shows, there are 9 dependency options including "package to package", "package to class", "package to feature", etc. We selected the "feature to feature" option of DependencyFinder to obtain dependencies at different granularity levels. With this option, it is possible to identify the callees of the application packages, files, and methods by parsing the dependency graph.

5.5 Evaluation

We designed three case studies in which we use the prototype of ERU on ten Android applications to investigate the following research questions:

- *RQ1: Usability* — Can ERU help to plan the application evolution process?

- *RQ2: Energy Testing* — Do proposed single source code unit changes ever result in not requiring energy testing for real applications using energy-greedy API methods?

- *RQ3: Impact* — How does the choice of history granularity, threshold (α) and equation for computing confidence values affect the energy testing requirement of proposed code changes?

- *RQ4: Cost* — What is the cost of getting feedback regarding the required energy tests?

The remainder of this section provides information about the selected applications, explains the experimental procedure that we followed, and discusses the results.

5.5.1 Subject Applications

Table 5.2 lists the applications that we used in our evaluation. The first two columns, *Application* and *Description*, list the name of each application and a brief description of its functionality, respectively. The third and fourth columns, *# Releases* and *# Commits*, show the number of releases and commits that we used to create release history and commit history of each application from the application's source code repository, respectively. Note that, we only considered the commits that contain source code changes that occurred in the current source code directory structure. Commits with only non-code changes (e.g., changes in user interface, Android SDK version, non-Java files, etc.) or within a different directory structure (e.g., before migrating Eclipse to Android Studio) were ignored. The fifth column, *Version*, indicates the latest version number of each application that was analyzed in our evaluation. The next two columns, *# Files* and *# Methods*, provide the number of source files and the number of methods in the latest version of each application, respectively. To gather all method and file names, and the number of methods and files in the application, we used *DependencyFinder*. The number of lines of code, *LoC*, is reported in the final column and only counts lines of code in the source files of the application. For example,

Table 5.2: Subject applications.

Application	Description	# Releases	# Commits	Version	# Files	# Methods	LoC
AdAway	Ad bloker	19	48	3.1.2	55	317	9421
AnkiDroid	A flashcard-based study aid	42	862	2.7	250	3027	69 206
Budget	Financial tracker	11	84	4.2	38	305	5564
Clover	4chan image board browser	22	663	2.2.0	230	2117	37 612
Photo Manager	Search photos in local media	14	281	5.3.16	72	706	15 200
Red Moon	Screen filter for night time use	21	143	2.9.1	39	209	5899
TintBrowser	Web browser	10	171	1.8	96	919	19 647
Tomdroid	Note taking application	11	574	0.7.5	50	446	12 123
Vanilla Music	Music player	38	1184	1.0.45	85	1000	25 036
Wikipedia	Official Wikipedia application	35	1214	2.4.160	519	4147	52 189

TintBrowser is a web browser application and it has 10 releases and 171 commits in its source code repository. Version 1.8 of TintBrowser consists of 96 Java files, 919 methods, and 19 647 lines of code.

We chose these specific applications because they have available source code repositories and they are representative of common Android application types. These applications also vary in the number of releases, commits, files, methods, and lines of code.

5.5.2 Experimental Procedure

We evaluated the prototype of ERU to determine whether energy testing is required for the proposed changes in ten Android applications. We conducted three case studies and followed the same procedure for each study. The first step was to select the source code unit granularity of the study and to identify the proposed changes. Because we are interested in source code changes, all source code units at the selected granularity level were considered as possible proposed changes. Then, the energy test requirements of each proposed change were investigated using each of the equations described in Section 5.4.2 with different thresholds, 0.5, 0.6, 0.7, 0.8, and 0.9. The details of the case studies are presented below:

* Case Study 1 - Method Level with Release History: The first case study was conducted at method level, and each application method was considered as a proposed change. The impact sets of the proposed changes were generated based on the release history of the application.

* Case Study 2 - File Level with Release History: The second case study was conducted at the file level, and each source file was considered as a proposed change. The impact sets of the proposed changes were generated based on the release history of the application.

* Case Study 3 - File Level with Commit History: Similar to the second study, the third case study was conducted at the file level. The difference is that the impact

sets of the proposed changes were generated based on the commit history of the application.

We have not conducted a case study at the method level with commit history. The only way we could find to obtain modified application methods in a commit is to use SVN repository although all of our considered applications are in Git repository. The underlying reason is the difficulty of migrating application's Git repository to SVN repository due to many branching-outs and merges in Git that SVN cannot handle. Therefore, we manually created SVN repository for each application by committing only source code of the application in its releases with respect to release order for the first case study. To use same release history in the second case study, we automatically converted application SVN repositories to Git repositories. Lastly, we used existing application Git repositories in the third case study.

5.5.3 Data Analysis and Discussion

In this section we describe results of our study in terms of our research questions.

RQ1: Usability

Table 5.3 shows the data used to answer the first research question. We gathered data the ERU prototype produces and uses internally to determine the energy test requirements of the proposed changes. The first column, *Application*, lists the name of each application. The remaining columns, *% Use eGreedy* and *% Known Impact*, show the percentage of proposed changes that use energy greedy API methods in eGreedy and the percentage of proposed changes whose impact is known, respectively. For example, in Tomdroid, 10.1 % of the proposed changes at the method level and 32.0 % of the proposed changes at the file level use energy greedy API methods. 20.9 %, 50.0 %, and 84.0 % of the proposed changes' impact are known at the method level with release history, at the file level with release history, and at the file level with commit history, respectively.

Table 5.3: Proposed changes.

	Method Level		File Level		
		Release History		Release History	Commit History
Application	% Use eGreedy	% Known Impact	% Use eGreedy	% Known Impact	% Known Impact
AdAway	19.2	15.1	40.0	87.3	83.6
AnkiDroid	5.5	29.2	24.8	64.8	63.6
Budget	14.8	20.3	57.9	60.5	92.1
Clover	4.3	13.4	22.2	48.3	74.8
Photo Manager	10.8	37.3	45.8	69.4	79.2
Red Moon	7.2	22.5	35.9	82.1	97.4
TintBrowser	11.1	13.4	43.8	60.4	87.5
Tomdroid	10.1	20.9	32.0	50.0	84.0
Vanilla Music	8.6	27.8	48.2	76.5	84.7
Wikipedia	3.2	23.4	16.0	66.5	75.0
Average	9.5	22.3	36.7	66.6	82.2

From the data shown in Table 5.3, we can observe that Android applications often use energy greedy API methods as all of the considered applications invoke some energy greedy API. However, the use of energy greedy API methods varies from application to application and ranges from 3.2 % (Wikipedia) to 19.2 % (AdAway) with an average value of 9.5 % for method level proposed changes. At the file level, it ranges from 16.0 % (Wikipedia) to 57.9 % (Budget) with an average of 36.7 %. As code changes may have an effect on the energy consumption of the applications since they contain energy greedy API methods, this data motivates the necessity of energy testing after the changes.

Developers currently anticipate performing energy tests after each source code change. However, a proposed change might not require all energy tests to be run, and time can be wasted on unnecessary energy tests. Therefore, providing feedback on energy test requirements of the proposed changes helps to improve the planning process of application evolution. ERU achieves this for the proposed changes whose impact is known.

Based on the data in Table 5.3, the percentage of proposed changes whose impact is known varies from application to application. For example, it ranges from 13.4 % (Tomdroid and TintBrowser) to 37.3 % (Photo Manager) with an average value of 22.3 % for the proposed changes at the method level with release history. More importantly, selected source code unit granularity and history have influence on the known impact.

When we compare studies at method level and file level with release history, it is clear that the percentage of known impact increases significantly at the file level for the same application. This is an expected result since a file might consist of several methods and has a higher chance of being changed than a method. In addition, some of the application methods may not need to be changed once they have been defined (e.g., getter/setter methods). This may lower the percentage of known impact at the method level.

Studies at the file level with release history and commit history show that release

history sometimes provides higher percentage of known impact than commit history (e.g., AdAway and AnkiDroid). This is possible when the source code directory structure has been changed in the existing application repository as we only examined the application source code changes in the current source code directory structure for commit history. For example, migrating Android applications built with Eclipse to Gradle causes directory structure changes. For release history, directory structure changes are not a constraint because release history that we created is independent from the directory structure since it only includes the source code of the application in its releases. Besides this exception, commit history provides a higher percentage of known impact than release history, as expected.

RQ2: Energy Testing

One of the major criteria that developers take into account to plan their application evolution process is the total cost of testing. In response to code changes, both traditional and energy testing should be performed, especially for mobile applications. However, the high cost of energy testing significantly increases the total testing cost, which can adversely impact the planning process of application evolution. In our evaluation, we investigated whether a proposed single source code unit change can ever result in not requiring energy testing.

Figures 5.3, 5.4, and 5.5 show the result of our investigation at method level with release history, file level with release history, and file level with commit history, respectively. These figures are faceted by the considered applications. In each facet, the y-axis shows the percentage of proposed changes that require energy testing, and the x-axis shows the selected threshold. The plot lines indicate the equation that is used to calculate the probability of being changed.

As the figures show, in the majority of cases, energy testing is not always required for some of the proposed changes in an application. Although the result may vary in each study depending on selected threshold and equation, there is no case in which all of the proposed changes in an application require energy testing regardless of

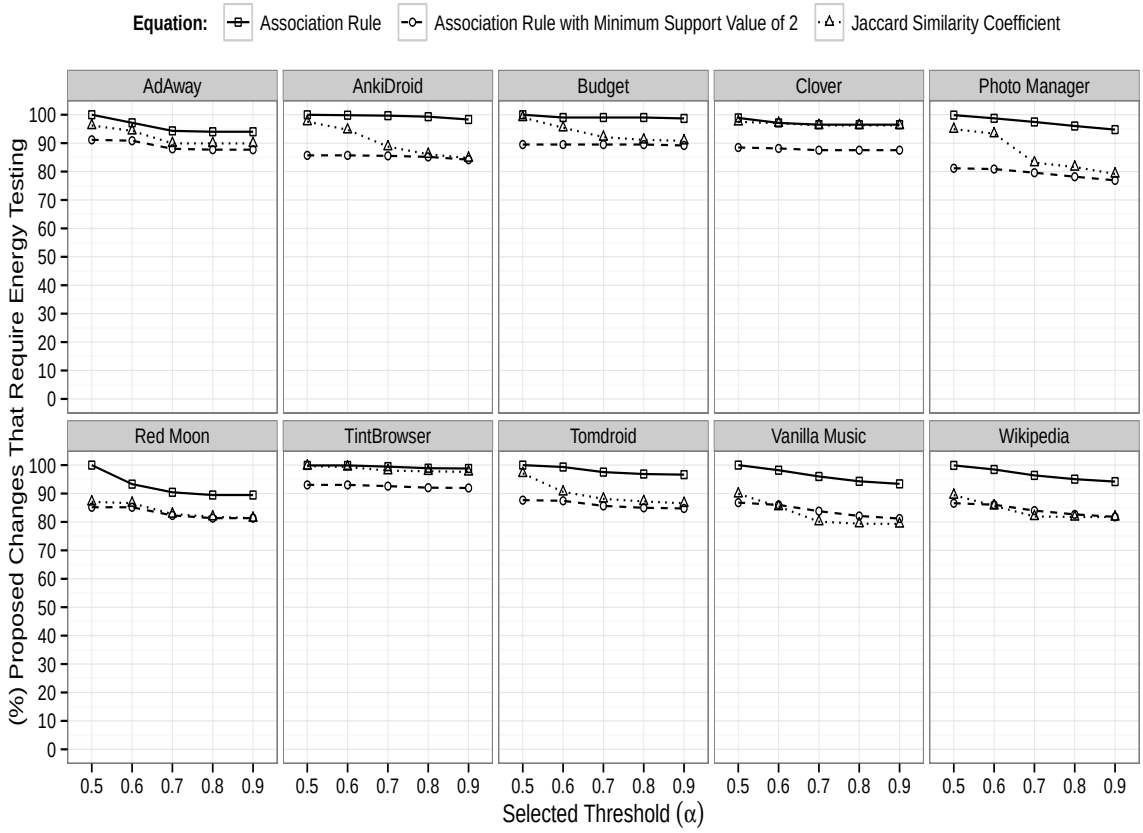


Figure 5.3: Energy Testing Feedback for Case Study 1: Method Level with Release History.

the selected threshold and equation except the Budget application for the association rule equation at the file level with release history. The underlying reason might be that the Budget application has limited release history and small number of source files.

RQ3: Impact

We examined the data in Figures 5.3, 5.4, and 5.5 again to address the third research question.

History: Studies at the file level can be comparable in terms of history effect on the energy testing requirement of proposed code changes because the only difference between these studies is that the impact sets are generated based on release or commit history. When we compare the Figures 5.3, 5.4, and 5.5, we can observe that

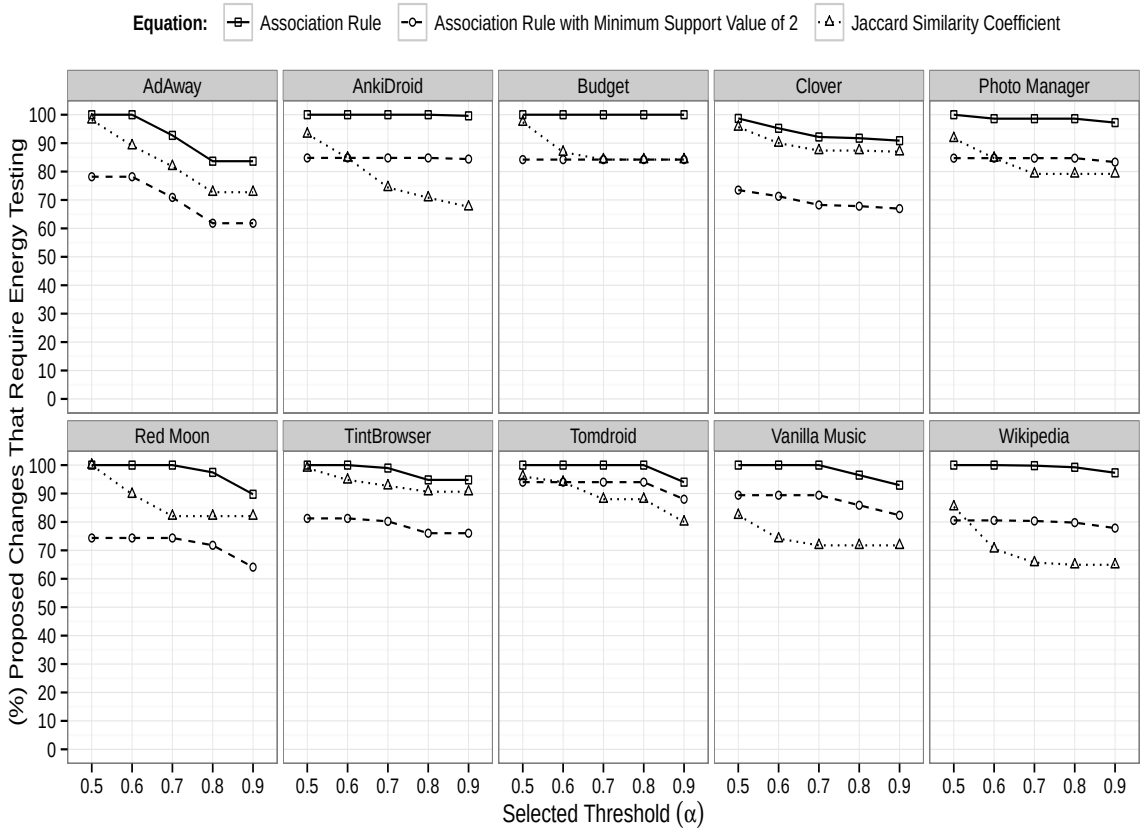


Figure 5.4: Energy Testing Feedback for Case Study 2: File Level with Release History.

using commit history most likely decreases the amount of required energy testing for the proposed changes. On average, the difference ranges from 9.0 % to 26.1 % in the threshold and equation, which is used to compute confidence values, configurations. The underlying reason is the possibility of over estimating the energy testing requirements due to coarse-grained release history. For example, modified application files between two releases are assumed to have been changed at the same time. However, it is possible that files have been changed and committed at different commits within two releases. While using fine-grained commit history may provide a more accurate change impact analysis result, using release history is still beneficial, as the result in Figure 5.4 shows that energy testing is not always required.

Threshold: Based on the data in Figures 5.3, 5.4, and 5.5, it is clear that the

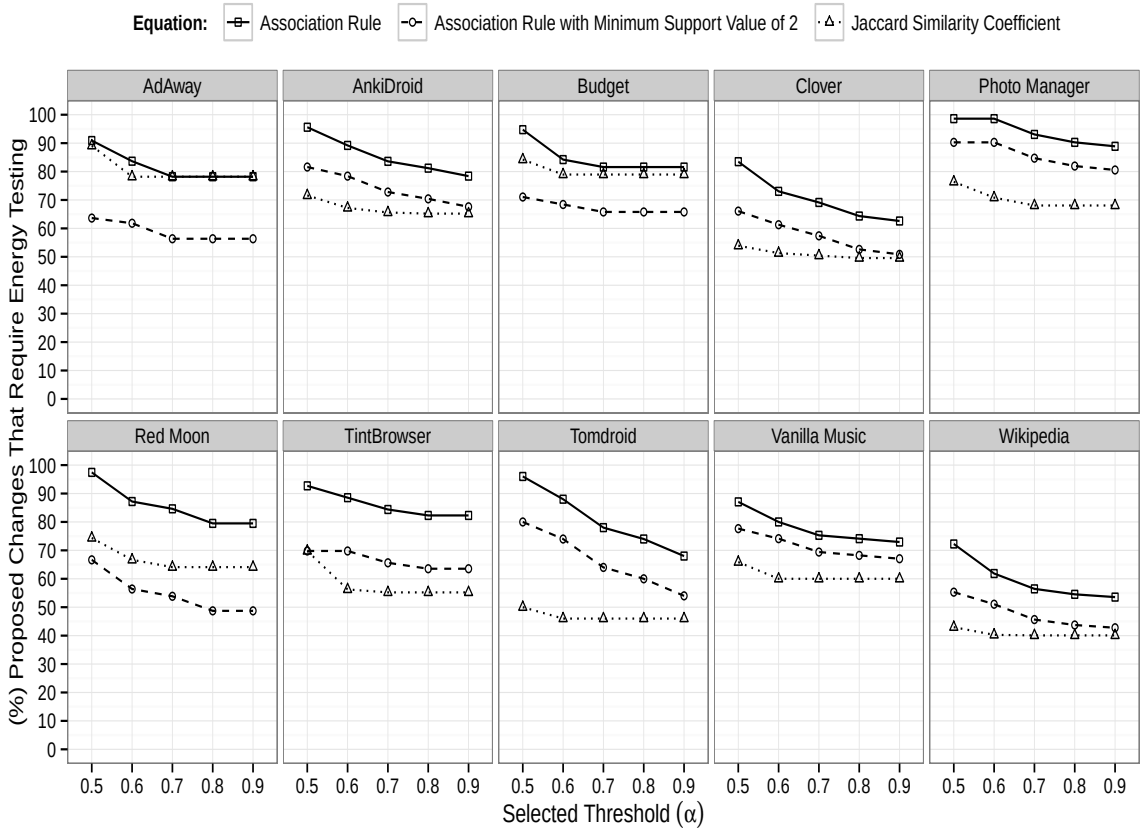


Figure 5.5: Energy Testing Feedback for Case Study 3: File Level with Commit History.

effect of the threshold varies according to application and equation used to compute confidence values. In general, we expected that selecting a higher threshold value would decrease the percentage of energy testing required for the proposed changes. However, it is not always the case. For example, the percentage of required energy testing for the proposed changes in AdAway application is steady for the thresholds 0.7, 0.8, and 0.9 at the file level with commit history. Conversely, in some cases, selecting higher threshold significantly decreases the percentage of proposed changes that require energy testing. For example, increasing the threshold from 0.5 to 0.6 at the file level with commit history decreases the percentage of proposed changes that require energy testing 13.5 % for TintBrowser application.

Equation: Different equations can be used to calculate the probability of being

impacted for source code units due to the proposed change in historical change impact analysis. From the data in Figures 5.3, 5.4, and 5.5, the plot lines are very similar for association rule and association rule with minimum support value of 2 in each study. The difference is that defining a minimum support value decreases the percentage of proposed changes that require energy testing. More interestingly, the Jaccard similarity coefficient equation is not consistent in and across case studies. It suggests more or less energy testing than association rule with minimum support value of 2 equation, although it most likely suggests less energy testing than the basic association equation.

RQ4: Cost

The cost of getting feedback regarding the energy test requirements is an important criteria of usability of ERU. The amount of time needed by the prototype of ERU to provide energy testing feedback of the proposed changes depends on several factors. These factors are the number of considered source code units as proposed changes and number of releases or commits in the application source code repository. In our case studies, time spent by the prototype was a few minutes for all proposed changes in an application. This time is negligible compared to the time that would have been unnecessarily spent performing expensive energy testing.

5.5.4 Potential Threats to Validity

One of the most significant threats to the validity of our results is that we considered 131 API methods and all methods in 3 API packages as energy greedy. There might exist other energy greedy API methods. However, we believe that we added all known energy greedy API methods into our Energy Greedy APIs (eGreedy). eGreedy is also extendable and developers can freely add API methods that are energy greedy.

A more specific concern is the possibility of over estimating the required energy testing due to the coarse-grained release history of considered applications. While using application commit history eliminates this threat, it might not be applicable all

the time. For example, we are not able to use commit history at the method level. Additionally, using release history is still beneficial since results of our case studies with release history show that some of the proposed changes do not require energy testing.

There are several additional threats to validity of our evaluation. First, considered applications do not have energy tests and test coverage information publicly available. As a result, the prototype of ERU could only determine whether energy testing is required. Second, dynamic impact analysis can be more precise than static impact analysis. However, dynamic impact analysis does require actually applying changes and executing application source code to generate an impact set. Therefore, we chose to use a static impact analysis. Third, we only considered ten Android applications. Although we selected these applications to cover different application types, it is possible that they may not be representative of all applications.

5.6 Related Work

Unlike our approach, which determines energy test requirements before making any code changes in a source code unit, the majority of existing work on energy testing has focused on minimizing the test suite with energy consideration [56, 69], finding energy bugs [11], and reducing the energy consumption of test suite [61].

For example, Jabbarvand et al. [56] propose an energy-aware test suite minimization approach to test the energy properties of an Android application with the minimum set of tests. They used integer programming and a greedy algorithm to reduce the test suite size while maintaining the test suite coverage for energy-greedy segments of an application that consume more energy. Similar to our study, they consider energy greedy APIs to determine energy-greedy segments of an application [70]. Their result shows that they are able to minimize test suite size without losing effectiveness of the test suite in revealing most of the energy bugs such as wakelock bugs, recurring callback bugs, and loop bugs.

Li et al. [69] optimize a test suite in terms of energy. They measure the energy consumption of the test cases using hardware and uses this information to generate

energy-efficient test suites by applying integer programming. While energy-efficient test suites that can be performed post-deployment testing on mobile and embedded systems have reduced energy consumption, their test coverage is equally effective with traditionally generated minimized test suites.

Banerjee et al. [11] present an automated test generation framework. Their framework systematically generates tests to detect energy hotspots and bugs in Android applications by combining a graph-based search algorithm and guidance heuristics. After generating a test that is a user interaction scenario, the framework executes the test on a smartphone and measures energy consumption simultaneously using a power meter. Then, the framework examines energy bugs and hotspots in different categories such as hardware resources, sleep-state transition heuristics, background services, and defective functionality. While it detects an energy bug based on the statistical dissimilarities in energy consumption of the device before and after executing the respective application with the test, it determines an energy hotspot that causes anomalous energy consumption by using an anomaly detection technique.

Kan [61] presents a technique to reduce the energy consumption of the CPU via the Dynamic Voltage and Frequency Scaling (DVFS) during the regression testing. This technique is conducted on the assumption that over the versions of a program that do not have significant changes in functionality, CPU-bound tests remain CPU-bound. Therefore, optimizing CPU frequency for execution of CPU-bound tests saves energy and helps to reduce the energy consumption of the test suite.

5.7 Summary

In this chapter, we have presented a new approach, ERU, that provides feedback on energy test requirements of the proposed code changes for helping developers plan their application evolution timeline effectively. ERU leverages change impact analysis and pre-computed API energy usage information. To evaluate the prototype of ERU, we used ten Android applications to determine energy testing requirements of the

proposed changes at different source code unit granularities with release or commit history. The results of this study demonstrate that:

(1) ERU can provide feedback on energy testing requirements of the proposed code changes.

(2) Android applications most likely use energy greedy API methods.

(3) Energy testing is not always required for proposed single source code unit changes.

(4) The percentage from proposed changes that required energy testing varies application to application, and it is affected by selected history granularity, threshold, and equation used to compute confidence values.

(5) The cost of the prototype implementation of ERU will allow it to be run as part of the application evolution cycle.

As such, we believe that getting feedback on energy test requirements of the proposed code changes is positive news for application developers. By using the feedback information, developers can plan the application evolution process and make decisions on code changes more informatively.

Chapter 6

CONCLUSION

The overall goal of my research is to enable and support software engineers in developing and maintaining energy-efficient applications. My dissertation work addresses this goal by first gathering knowledge about how software engineering decisions impact the overall energy usage of an application and second by developing a technique for supporting the software engineering process. This chapter summarizes contributions of the dissertation and discusses the potential future work.

6.1 Summary of Contributions

The main contributions of this dissertation are as follows:

(1) Guidelines to design and conduct high-quality empirical studies on software engineering decisions with energy consideration. The quality of empirical studies is important to make accurate observations. By following each step of the methodology in our empirical studies, researchers can acquire the skills and experience necessary to empirically investigate the energy impacts of software engineering decisions.

(2) Data generated by four empirical studies of major software engineering decisions including design patterns, code refactorings, code obfuscations, and performance tips. The experimental data is generated to investigate how the considered software engineering decisions impact the energy consumption of applications. In empirical studies, 15 design patterns, six code refactorings, 18 code obfuscations, and four performance tips were considered. In total, approximately 75 000 executions were run on a suitable hardware-based energy measurement platform.

(3) Analyses of the generated data to determine how software engineering decisions impact energy usage. To analyze the generated data, we used appropriate statistical approaches. In general, this means using non-parametric methods (e.g., Mann-Whitney-Wilcoxon test, the Kruskal-Wallace test, Vargha and Delaney's $\hat{A}_{12}$ statistic, etc.). The analyses of data demonstrate that all of the decisions have the potential to both increase and decrease energy usage of applications except performance tips. This finding confirms that given a better understanding of the implications of software engineering decisions with regard to energy consumption, software engineers can play an important role in reducing the energy usage of the applications they write.

(4) A technique to predict energy testing requirements of proposed code changes. We have presented a new approach to provide developers with feedback on the energy testing requirements of proposed code changes. Our technique leverages change impact analysis and pre-computed API energy usage information. More specifically, for a proposed change, the technique predicts whether energy testing will be required, and if so, which energy tests will need to be run. Such information allows developers to develop an effective application evolution timeline. Because they have more accurate information about the amount of energy testing that is required, time that would have been unnecessarily used for energy testing can be allocated to performing additional changes in a release.

(5) A prototype implementation of the technique for Android applications. We have implemented a prototype for Android applications to evaluate the technique. The prototype leverages Historical Change Impact Analysis and pre-computed Android API energy usage information. Using the prototype, we performed a preliminary study on ten Android applications to investigate the feasibility of the technique and how it performs when changes are expressed at differing granularity levels. The results of the evaluation are promising and show that the technique is feasible and able to provide useful feedback.

6.2 Future Work

In the future, it is likely that developing and maintaining energy-efficient applications will be continue to be an important research area. Since the work in this dissertation benefits both researchers and developers, it can be extended in several ways.

(1) Replicate and improve existing empirical studies. Although replication is not as common for studies in the software engineering community as it is in other areas, we believe that replication is an important part of the research process. Replicating our studies by using additional platforms (e.g., tablets), architectures (e.g., Windows phone), and applications written in different programming languages (e.g., Python) enlarges the scope of studies and helps to generalize or limit our observations.

(2) Investigate the energy impacts of additional software engineering decisions. Empirical studies on other software engineering decisions that possibly impact the energy consumption can be conducted to provide more knowledge and satisfy the expectations of developers [76]. Such decisions might include removing and adding layers of abstraction, using different algorithms (e.g., incorporating parallelism), offloading or moving computation to the cloud or other accelerators (e.g., GPUs), using alternative data representations, and using alternative architectural styles (e.g., event-driven architecture instead of polling).

(3) Build and release tool implementation of the technique. We believe that, with further research and development, our technique has the potential to become a practical tool for planning application evolution. For the first tool release of the technique, we plan to improve prototype implementation of the technique for Android applications and implement it as a tool. To improve the prototype, we will first develop a way to use commit history of the application at the method level. Then, we will investigate the best choice of the threshold and equation used to compute confidence values for the prototype by analyzing real code changes and interviewing developers.

(4) Develop decision support tools. Developing decision support tools that help managing energy consumption at all levels of the development process, from design

to implementation to maintenance, can be the most promising approach in the future. These tools can enable developers to discover and apply right choices for reducing the energy usage of their applications without the low-level, tedious work in analyzing software, applying changes, and monitoring the resulting impacts to energy usage.

BIBLIOGRAPHY

[1] *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] http://www.allatori.com.

[3] Jean Alzieu, Hassan Smimite, and Christian Glaize. Improvement of intelligent battery controller: State-of-charge indicator and associated functions. *Journal of Power Sources*, 67(1-2):157–161, 1997.

[4] Nadine Amsel, Zaid Ibrahim, Amir Malik, and Bill Tomlinson. Toward sustainable software engineering (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, pages 976–979, 2011.

[5] Nadine Amsel and Bill Tomlinson. Green tracker: A tool for estimating the energy consumption of software. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems: Extended Abstracts*, pages 3337–3342, 2010.

[6] Android performance tips. http://developer.android.com/training/articles/perf-tips.html, 2013. Accessed 30 October 2014.

[7] Users reveal top frustrations that lead to bad mobile app reviews. http://apigee.com/about/pressrelease/apigee-survey-users-reveal-top-frustrations-lead-bad-mobile-app-reviews, 2012.

[8] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1–10, 2011.

[9] Sarala Arunagiri, Victor J. Jordan, Patricia J. Teller, Joseph C. Deroba, Dale R. Shires, Song J. Park, and Lam H. Nguyen. Stereo matching: Performance study of two global algorithms. *SPIE Proceedings*, 8021:80211Z–80211Z–17, 2011.

[10] Woongki Baek and Trishul M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 198–209, 2010.

[11] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598, 2014.

[12] Thomas W. Bartenstein and Yu David Liu. Green streams for data-intensive software. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 532–541, 2013.

[13] Yoav Benjamini and Yosef Hochberg. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995.

[14] Shawn A. Bohner and Robert S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[15] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 83–94, 2000.

[16] C. Bunse, H. Hopfner, S. Roychoudhury, and E. Mansour. Choosing the 'best' sorting algorithm for optimal energy consumption. In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 199–206, 2009.

[17] Christian Bunse and Sebastian Stiemer. On the energy consumption of design patterns. In *Proceedings of the 2nd Workshop EASED@BUIS Energy Aware Software-Engineering and Development*, pages 7–8, 2013.

[18] http://www.businessinsider.com/android-piracy-problem-2015-1.

[19] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th International Symposium on Computer Architecture*, pages 225–236, 2012.

[20] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, pages 21–21, 2010.

[21] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, Marco Torchiano, and P. Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *Proceedings of 17th IEEE International Conference on Program Comprehension*, pages 178–187, 2009.

[22] Mariano Ceccato, Andrea Capiluppi, Paolo Falcarin, and Cornelia Boldyreff. A large study on the effect of code obfuscation on the quality of Java code. *Empirical Software Engineering*, pages 1–39, 2014.

[23] C.-F. Chiasserini and R. R. Rao. Energy efficient battery management. *IEEE Journal on Selected Areas in Communications*, 19(7):1235–1245, 2001.

[24] https://www.atlassian.com/software/clover/overview.

[25] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 831–850, 2012.

[26] Wellisson G. P. da Silva, Lisane Brisolara, Ulisses B. Correa, and Luigi Carro. Evaluation of the impact of code refactoring on embedded software efficiency. In *Proceedings of the 1st Workshop de Sistemas Embarcados*, pages 145–150, 2010.

[27] http://www.preemptive.com/products/dasho.

[28] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 189–194, 2010.

[29] J. W. Davidson and S. Jinturkar. Memory access coalescing: A technique for eliminating redundant memory accesses. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 186–195, 1994.

[30] V. Delaluz, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. DRAM energy management using software and hardware directed power mode control. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 159–169, 2001.

[31] http://depfind.sourceforge.net/, 2016. Accessed 1 June 2016.

[32] Mian Dong and Lin Zhong. Self-constructive, high-rate energy modeling for battery-powered mobile systems. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, pages 335–348, 2011.

[33] F. Douglis, P. Krishnan, and B. N. Bershad. Adaptive disk spin-down policies for mobile computers. In *Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 121–137, 1995.

[34] M. Doyle and J. S. Newman. Analysis of capacity-rate data for lithium batteries using simplified models of the discharge process. *Journal of Applied Electrochemistry*, 27(7), 1997.

[35] http://www.eclipse.org/org/usagedata.

[36] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 301–312, 2012.

[37] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance setting for dynamic voltage scaling. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 260–271, 2001.

[38] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1:213–226, 1992.

[39] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[40] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. RERAN: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 72–81, 2013.

[41] http://developer.android.com/tools/help/proguard.html.

[42] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering*, pages 100–110, 2015.

[43] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mary Jane Irwin, N. Vijaykrishnan, Mahmut Kandemir, Tao Li, and Lizy Kurian John. Using complete machine simulation for software power estimation: The SoftWatt approach. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, pages 141–151, 2002.

[44] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating Android applications' CPU energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 1–7, 2012.

[45] Shuai Hao, Ding Li, William G. J. Halfond, and Ramesh Govindan. Estimating mobile application energy consumption using program analysis. In *Proceedings of the 35th International Conference on Software Engineering*, pages 92–101, 2013.

[46] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *Proceedings of the 2003 International Symposium on Low Power Electronics and Design*, pages 217–222, 2003.

[47] Abram Hindle. Green mining: A methodology of relating software change to power consumption. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 78–87, 2012.

[48] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 12–21, 2014.

[49] C.-H. Hsu and U. Kremer. The design, implementation, and evaluation of a compiler algorithm for CPU energy reduction. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 38–48, 2003.

[50] C. Hu, D. A. Jiménez, and U. Kremer. Efficient program power behavior characterization. In *Proceedings of the 2nd International Conference on High Performance Embedded Architectures and Compilers*, pages 183–197, 2007.

[51] C. Hu, D. A. Jiménez, and U. Kremer. Combining edge vector and event counter for time-dependent power behavior characterization. In *Transactions on High Performance Embedded Architectures and Compilers II*, pages 85–104. Springer-Verlag, 2009.

[52] Cuixiong Hu and Iulian Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, pages 77–83, 2011.

[53] P.-K. Huang and S. Ghiasi. Efficient and scalable compiler-directed energy optimization for realtime applications. *ACM Transactions on Design Automation of Electronic Systems*, 12:27:1–27:16, 2008.

[54] N. Hunt, P.S. Sandhu, and L. Ceze. Characterizing the performance and energy efficiency of lock-free data structures. In *Proceedings of the 15th Workshop on Interaction between Compilers and Computer Architectures*, pages 63–70, 2011.

[55] A. Iyer and D. Marculescu. Power aware microarchitecture resource scaling. In *Proceedings of the 2001 Design, Automation and Test in Europe, Conference and Exhibition*, pages 190–196, 2001.

[56] Reyhaneh Jabbarvand, Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Energy-aware test-suite minimization for android apps. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 425–436, 2016.

[57] https://en.wikipedia.org/wiki/Jaccard_index, 2016. Accessed 30 October 2016.

[58] Mohammad-Amin Jashki, Reza Zafarani, and Ebrahim Bagheri. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 84–90, 2008.

[59] http://junit.org.

[60] I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam. Compiler-directed high-level energy estimation and optimization. *ACM Transactions in Embedded Computing Systems*, 4:819–850, 2005.

[61] E. Y. Y. Kan. Energy efficiency in testing and regression testing a comparison of dvfs techniques. In *Proceedings of the 13th International Conference on Quality Software*, pages 280–283, 2013.

[62] Miryung Kim, Dongxiang Cai, and Sunghun Kim. An empirical investigation into the role of api-level refactorings during software evolution. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 151–160, 2011.

[63] R. Kravets and P. Krishnan. Power management techniques for mobile communication. In *Proceedings of the 4th ACM/IEEE International Conference on Mobile Computing and Networking*, pages 157–168, 1998.

[64] U. Kremer. Low power/energy compiler optimizations. *Low-Power Electronics Design*, pages 2–5, 2005.

[65] Steffen Lehnert. A review of software change impact analysis. 2011.

[66] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Journal of Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[67] Ding Li and William G. J. Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 46–53, 2014.

[68] Ding Li, Shuai Hao, Jiaping Gui, and William G. J. Halfond. An empirical study of the energy consumption of android applications. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, pages 121–130, 2014.

[69] Ding Li, Cagri Sahin, James Clause, and William G. J. Halfond. Energy-directed test suite optimization. In *Proceedings of the 2Nd International Workshop on Green and Sustainable Software*, pages 62–69, 2013.

[70] Mario Linares-Vasquez, Gabriele Bavota, Carlos Eduardo Bernal Cardenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy API usage patterns in Android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11, 2014.

[71] Andreas Litke, Kostas Zotos, Alexander Chatzigeorgiou, and George Stephanides. Energy consumption analysis of design patterns. In *Proceedings of the International Conference on Machine Learning and Software Engineering*, pages 86–90, 2005.

[72] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving dram refresh-power through critical data partitioning. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 213–224, 2011.

[73] Yu David Liu. Energy-efficient synchronization through program patterns. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 35–40, 2012.

[74] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Requester-aware power reduction. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 18–23, 2000.

[75] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 335–346, 2008.

[76] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, pages 237–248, 2016.

[77] Irene Manotas, Cagri Sahin, James Clause, Lori Pollock, and Kristina Winbladh. Investigating the impacts of web servers on web application energy usage. In *Proceedings of the Second International Workshop on Green and Sustainable Software*, 2013.

[78] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh. Techniques for low energy software. In *Proceedings of the 1997 International Symposium on Low Power Electronics and Design*, pages 72–75, 1997.

[79] http://www.microsoft.com/en-us/download/details.aspx?id=7490.

[80] Matteo Monchiero, Ramon Canal, and Antonio González. Design space exploration for multicore architectures: A power/performance/thermal view. In *Proceedings of the 20th Annual International Conference on Supercomputing*, pages 177–186, 2006.

[81] https://developer.android.com/studio/test/monkey.html, 2016. Accessed 19 January 2016.

[82] https://www.msoon.com/LabEquipment/PowerMonitor/.

[83] T. Mudge, T. Austin, and D. Grunwald. The reference manual for the SimPanalyzer version 2.0. http://web.eecs.umich.edu/~panalyzer/.

[84] Sona Mundody and Sudarshan K. Evaluating the impact of android best practices on energy consumption. *IJCA Proceedings on International Conference on Information and Communication Technologies*, (8):1–4, 2014.

[85] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How we refactor, and how we know it. In *Proceedings of the 31st International Conference on Software Engineering*, pages 287–297, 2009.

[86] E. Musoll. A thermal-friendly load-balancing technique for multi-core processors. In *Proceedings of the 9th International Symposium on Quality Electronic Design*, pages 549–552, 2008.

[87] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on GreenIT. In *First International Workshop on Green and Sustainable Software*, pages 21–27, 2012.

[88] Adel Noureddine and Ajitha Rajan. Optimising energy consumption of design patterns. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, pages 623–626, 2015.

[89] Jae Jin Park, Jang-Eui Hong, and Sang-Ho Lee. Investigation for software power consumption of code refactoring techniques. In *The Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering (SEKE 2014)*, 2014.

[90] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, pages 5:1–5:6, 2011.

[91] 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 *In Proceedings of the 7th ACM European Conference on Computer Systems*, pages 29–42, 2012.

[92] Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, pages 267–280, 2012.

[93] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *Proceedings of the 1998 International Symposium on Low Power Electronics and Design*, pages 76–81, 1998.

[94] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the IpARM microprocessor system. In *Proceedings of the 2000 International Symposium on Low Power Electronics and Design*, pages 96–101, 2000.

[95] N. Pettis, J. Ridenour, and Y.-H. Lu. Automatic run-time selection of power policies for operating systems. In *Proceedings of the Conference on Design, Automation, and Test in Europe*, pages 508–513, 2006.

[96] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proceedings of the 7th International Conference on Mobile Computing and Networking*, pages 251–259, 2001.

[97] http://proguard.sourceforge.net.

[98] R. Prez-Castillo and M. Piattini. Analyzing the harmful effect of god class refactoring on power consumption. *IEEE Software*, 31(3):48–54, 2014.

[99] Arun Rangasamy, Rahul Nagpal, and Y.N. Srikant. Compiler-directed frequency and voltage scaling for a multiple clock domain microarchitecture. In *Proceedings of the 5th Conference on Computing Frontiers*, pages 209–218, 2008.

[100] Ana Rodriguez, Mathias Longo, and Alejandro Zunino. Using bad smell-driven code refactorings in mobile applications to reduce battery usage. In *Simposio Argentino de Ingenieria de Software*, pages 56–68, 2015.

[101] C. Sahin, F. Cayci, I. L. M. Gutirrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. In *First International Workshop on Green and Sustainable Software (GREENS)*, pages 55–61, 2012. http://dx.doi.org/10.1109/GREENS.2012.6224257.

[102] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 36:1–36:10, 2014. http://dx.doi.org/10.1145/2652524.2652538.

[103] Cagri Sahin, Lori Pollock, and James Clause. From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. *Journal of Systems*

and Software, 117:307–316, 2016. http://dx.doi.org/10.1016/j.jss.2016.03.031.

[104] Cagri Sahin, Mian Wan, Philip Tornquist, Ryan McKenna, Zachary Pearson, William G. J. Halfond, and James Clause. How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, 28(7):565–588, 2016. http://dx.doi.org/10.1002/smr.1762.

[105] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. Enerj: Approximate data types for safe and general low-power computation. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 164–174, 2011.

[106] H. Saputra, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, J. S. Hu, C-H. Hsu, and U. Kremer. Energy-conscious compilation based on voltage scaling. In *Proceedings of the Joint Conference on Languages, Compilers, and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, pages 2–11, 2002.

[107] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. An energy consumption framework for distributed Java-based systems. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 421–424, 2007.

[108] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Component-level energy consumption estimation for distributed Java-based software systems. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, pages 97–113, 2008.

[109] Digvijay Singh, Peter A. H. Peterson, Peter L. Reiher, and William J. Kaiser. The Atom LEAP platform for energy-efficient embedded computing: Architecture, operation, and system implementation. 2010.

[110] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, pages 161–174, 2007.

[111] V. Srinivasan, G. R. Shenoy, S. Vaddagiri, D. Sarma, and V. Pallipadi. Energy-aware task and interrupt management in Linux. In *Proceedings of the Linux Symposium*, volume 2, 2008.

[112] C.-L. Su, C.-Y. Tsui, and A. M. Despain. Low power architecture design and compilation techniques for high-performance processors. In *Compcon Spring '94, Digest of Papers*, pages 489–498, 1994.

[113] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. In *Proceedings of the 1994 IEEE/ACM International Conference on Computer-aided Design*, pages 384–390, 1994.

[114] Aline Tonini, Leonardo Fischer, Jùlio Mattos, and Lisane Brisolara. Analysis and evaluation of the Android best practices impact on the efficiency of mobile applications. In *Brazilian Symposium on Computing Systems Engineering*, pages 157–158, 2013.

[115] Andras Vargha and Harold D. Delaney. A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[116] Antonio Vetro, Luca Ardito, Giuseppe Procaccianti, and Maurizio Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. In *Proceedings of the Third International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies*, pages 34–39, 2013.

[117] https://www.wattsupmeters.com/secure/index.php.

[118] Zhenchang Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported—an Eclipse case study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, 2006.

[119] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. *ACM SIGOPS Operating Systems Review*, 36:123–132, 2002.

[120] Jack Zhang, Ayemi Musa, and Wei Le. A comparison of energy bugs for smartphone platforms. In *Proceedings of the 1st International Workshop on the Engineering of Mobile-Enabled Systems*, pages 25–30, 2013.

[121] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.

[122] http://www.zelix.com/klassmaster/.

Appendix

PERMISSIONS

JOHN WILEY AND SONS LICENSE TERMS AND CONDITIONS

Jul 02, 2017

This Agreement between Cagri Sahin ("You") and John Wiley and Sons ("John Wiley and Sons") consists of your license details and the terms and conditions provided by John Wiley and Sons and Copyright Clearance Center.

License Number	4141100296668
License date	Jul 02, 2017
Licensed Content Publisher	John Wiley and Sons
Licensed Content Publication	Journal of Software Maintenance and Evolution: Research and Practice
Licensed Content Title	How does code obfuscation impact energy usage?
Licensed Content Author	Cagri Sahin,Mian Wan,Philip Tornquist,Ryan McKenna,Zachary Pearson,William G. J. Halfond,James Clause
Licensed Content Date	Jan 6, 2016
Licensed Content Pages	24
Type of Use	Dissertation/Thesis
Requestor type	Author of this Wiley article
Format	Print and electronic
Portion	Full article
Will you be translating?	No
Title of your thesis / dissertation	Empirically Investigating Energy Impacts of Software Engineering Decisions
Expected completion date	Aug 2017
Expected size (number of pages)	140
Requestor Location	Cagri Sahin 101 Smith Hall University of Delaware NEWARK, DE 19716 United States Attn: Cagri Sahin
Publisher Tax ID	EU826007151

ELSEVIER LICENSE
TERMS AND CONDITIONS

Jul 02, 2017

This Agreement between Cagri Sahin ("You") and Elsevier ("Elsevier") consists of your license details and the terms and conditions provided by Elsevier and Copyright Clearance Center.

License Number	4141091360469
License date	Jul 02, 2017
Licensed Content Publisher	Elsevier
Licensed Content Publication	Journal of Systems and Software
Licensed Content Title	From benchmarks to real apps: Exploring the energy impacts of performance-directed changes
Licensed Content Author	Cagri Sahin,Lori Pollock,James Clause
Licensed Content Date	Jul 1, 2016
Licensed Content Volume	117
Licensed Content Issue	n/a
Licensed Content Pages	10
Start Page	307
End Page	316
Type of Use	reuse in a thesis/dissertation
Portion	full article
Format	both print and electronic
Are you the author of this Elsevier article?	Yes
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	Empirically Investigating Energy Impacts of Software Engineering Decisions
Expected completion date	Aug 2017
Estimated size (number of pages)	140
Elsevier VAT number	GB 494 6272 12
Requestor Location	Cagri Sahin 101 Smith Hall University of Delaware NEWARK, DE 19716 United States Attn: Cagri Sahin
Publisher Tax ID	98-0397604