Noname manuscript No.
(will be inserted by the editor)

GreenScaler: Training Software Energy Models With Automatic Test Generation

Shaiful Chowdhury · Stephanie Borle · Stephen Romansky · Abram Hindle

Received: date / Accepted: date

Abstract Software energy consumption is a performance related non-functional requirement that complicates building software on mobile devices today. Energy hogging applications (apps) are a liability to both the end-user and software developer. Measuring software energy consumption is non-trivial, requiring both equipment and expertise, yet researchers have found that software energy consumption can be modelled. Prior works have hinted that with more energy measurement data we can make more accurate energy models. This data, however, was expensive to extract because it required energy measurement of running test cases (rare) or time consuming manually written tests. In this paper, we show that automatic random test generation with resource-utilization heuristics can be used successfully to build accurate software energy consumption models. Code coverage, although well-known as a heuristic for generating and selecting tests in traditional software testing, performs poorly at selecting energy hungry tests.

Shaiful Chowdhury
Department of Computing Science
University of Alberta, Edmonton, AB, Canada
E-mail: shaiful@ualberta.ca

Stephanie Borle
Department of Computing Science
University of Alberta, Edmonton, AB, Canada
E-mail: sgil@ualberta.ca

Stephen Romansky
Department of Computing Science
University of Alberta, Edmonton, AB, Canada
E-mail: romansky@ualberta.ca

Abram Hindle
Department of Computing Science
University of Alberta, Edmonton, AB, Canada
E-mail: abram.hindle@ualberta.ca

We propose an accurate software energy model, *GreenScaler*, that is built on random tests with CPU-utilization as the test selection heuristic. *GreenScaler* not only accurately estimates energy consumption for randomly generated tests, but also for meaningful developer written tests. Also, the produced models are very accurate in detecting energy regressions between versions of the same app. This is directly helpful for the app developers who want to know if a change in the source code, for example, is harmful for the total energy consumption. We also show that developers can use *GreenScaler* to select the most energy efficient API when multiple APIs are available for solving the same problem. Researchers can also use our test generation methodology to further study how to build more accurate software energy models.

1 Introduction

Does software energy consumption matter? The answer is *yes*. Mobile device users prioritize longer battery life when investing in their next purchase [1–3]. Mobile device users also complain about battery life: recently Microsoft acknowledged that a software bug, unrelated to battery hardware, induced short battery life on Surface Pro 3 tablets [4]. Accordingly, developers are trying to write more energy efficient code to meet the need of consumers [5–7]. Research has shown that energy efficiency can be improved significantly with small code optimization [8–12]. To develop energy efficient software, developers need feedback about the energy consumption of their software. Unfortunately, developers are not sure how to measure and optimize the energy consumption of their apps [1,13].

We seek to help Android developers *accurately estimate energy consumption of their software without the need for hardware instrumentation* and without physically measuring their own software's energy consumption. Instead, developers can use an externally developed and robust model, built from physical measurements of third party apps, to accurately estimate their own software's energy consumption. Measurements of third party apps, however, are hard to find as we need repeatable test cases and corresponding energy measurements. *These test cases are costly to build manually and are the main limitation of empirically derived models* [11]. We address this limitation by demonstrating the effectiveness of automatic test generation to collect measurements for energy models.

We propose *GreenScaler*, an easy to interpret energy model for Android apps. *GreenScaler* leverages a continuous process of test generation to build an ever more robust corpus of energy measurements. As of writing, *GreenScaler* learns from a wide variety of 472 real world Android apps, which was made possible through automatic test generation. *GreenScaler* is count based and relies on counts of system calls, CPU time, and other OS-level statistics.

The contributions of this paper are summarized as follows.

1) *We propose a process of continuously building an ever more accurate software energy consumption model using automatic test generation and test selection heuristics.* The success of automatic test generation for building accurate software energy models is significant. We can continuously improve model's performance by adding more apps in training. New research ideas can be explored with this approach. Researchers can investigate further for producing better energy models. For example, can we improve model's accuracy by building domain specific models (building a separate model for gaming apps for example)?

2) For using random tests, we need test selection heuristics. *We evaluate three test selection heuristics to understand which one is the most effective for selecting energy tests.* From empirical results we show the following. i) Code coverage is not a good heuristic for selecting tests to produce energy models. ii) A simple CPU-utilization heuristic performs similar to a complex energy-estimating test heuristic. To the best of our knowledge, we are the first to evaluate test selection heuristics for producing software energy models.

3) *We propose the GreenScaler model that can accurately estimate software energy consumption of apps without hardware instrumentation. GreenScaler* is trained and tested (with leave-one-out approach) on 472 apps with randomly generated test scripts. The model shows an upper error bound of 10% when compared with the ground truths, except for few extreme cases. As *GreenScaler* is built on randomly generated tests, it is also important to evaluate its accuracy on human written meaningful tests. For such manually written tests of 984 versions from 24 real world Android apps, the upper error bound of *GreenScaler* is always less than 10%. To the best of our knowledge, no previous software energy model was evaluated on such a large number of apps.

4) *We show that GreenScaler is accurate in finding energy regressions between versions of the same app, regardless of the amount of change in the source code. GreenScaler* detects energy regression even for a single API change, when such a change has significant impact on the app's energy consumption. This is directly helpful for the developers who want to examine if a new version consumes more energy than the previous version. With *GreenScaler*, researchers can build API recommendation systems for energy-aware developers.

5) *We publicly release our dataset and tools* to enable replication and extension [14]. The dataset contains measurements that took us nearly two years to collect, including time for test generation and time for actual energy measurements. Our automatic test generation and selection tool can be used to add more apps for building better energy models. Developers can directly use our energy prediction tool to estimate their apps' energy consumption.

1.1 Paper Organization

The main focus of this paper is to build an accurate software energy model for Android systems—a model that learns from hundreds of apps. However, it is laborious to write tests to drive those hundreds apps. So we need automatic test generation. We study the previous test generation techniques in section 2, with the description of other important concepts related to this paper. We show that Android Monkey is the best available test generation technique for building software energy models. However, Monkey has its own drawbacks—generates too many redundant events and does not offer us an way to control the distribution of individual events. So we made our own Monkey, *Green-Monkey*. GreenMonkey is still a random test generation technique, and might produce test cases that do not exercise energy consuming operations. We generated several test cases for each app to select the best one. To select the best test case, we need test selection heuristic. Which test should we select? Test that cover more code? We study the effectiveness of code coverage heuristic in selecting test cases that exploit energy expensive resources, and found that code coverage would not be a good heuristic for selecting energy consuming test cases (Section 3). Instead, we focused on resource-utilization heuristics. Section 4 describes the whole *GreenScaler* methodology of building software energy models with resource-utilization heuristics. The rest of the paper is about evaluating our model from different perspectives (Section 5 to Section 8). In section 9, we discuss the future research avenues with our model building approach, followed by the description of our dataset (Section 10), Threats to Validity (Section 11), Related work (Section 12), and Conclusion (Section 13).

2 Background

This section explains the important concepts that are frequently used in this paper. It also describes the motivation for energy model building automated test generation.

2.1 Power vs. Energy

Power (P) is defined as the rate of work completion and measured in *watts* whereas energy (E) is the total amount of work done for a given time T ($E = P \cdot T$) and expressed in *joules* [15–17]. Understanding the difference between power and energy is important to develop energy efficient system. A misconception exists among developers: improving execution time automatically improves energy efficiency [1,13]. Improving execution time reduces T in the equation. However, with the reduced execution time the CPU workload may also increase, which can switch the CPU to its highest frequency,

which is also its highest power using state, thus negatively affecting the overall energy consumption. Furthermore T can be reduced by parallelizing a task across multiple cores, which could induce even higher power use. This is also confirmed by a previous study that shows that less execution time does not necessarily indicate less energy consumption [18].

2.2 System Calls and CPU Time

System calls act as the bridge between an app and the OS. For example, `socket` is a system call responsible for creating communication endpoints, whereas `read` takes the responsibility to read from a file handle. Counting system calls of different types can thus provide an estimation of the amount of different resource usage by an app [11, 19, 20].

To represent the CPU time expended by a process, we used the number of CPU jiffies provided by the Linux kernel. A CPU jiffy is a period of time assigned for a process to run without any intervention [21]. A CPU can operate in different power consuming states, which complicates software energy modeling [13]. A CPU jiffy, however, can be of different time intervals based on the CPU states. Thus considering CPU jiffies as CPU time would mitigate some intricacy involved in software energy modeling.

2.3 Energy Measurement: GreenMiner

For measuring energy consumption and resource usage, we used *GreenMiner*, which is fully described in Hindle et al. [22]. *GreenMiner* provides accurate energy measurements for Android apps and is widely accepted in the software energy research community [9, 11, 15, 20, 23, 24]. The main components of this test-bed are a lab-bench power supply (a YiHua YH-305D), a test-runner computer (a Raspberry Pi model B computer) for controlling the experiments, an energy measurement IC (Adafruit INA219 breakout board), a micro-controller (Arduino Uno) for collecting energy measurements, and a system-under-test (a Galaxy Nexus phone) (Table 1). The Arduino and Raspberry Pi are powered by a USB hub. Each testbed costs approximately $250, each phone originally cost approximately $500, and the green miner service is run on a separate server ($1000). Development of the GreenMiner hardware and software itself was more than $32000 in developer time. GreenMiner software is freely available for download [22].

A test-runner, a Raspberry Pi, is connected to a particular system-under-test, a Galaxy Nexus. The test-runner pushes and runs tests on the Galaxy Nexus, and collects measurements from the Arduino. Afterwards the test-runner downloads statistics and other meta-data from the system-under-test. The responsible test-runner, a Raspberry Pi, then uploads the measurements

Table 1: Specs of the Samsung Galaxy Nexus phones used for the experiments [25].

Component	Specs
OS	Ice Cream Sandwich, 4.4.2
CPU	Dual-core 1.2 GHz Cortex-A9
GPU	PowerVR SGX540
Memory	16 GB, 1 GB RAM
Display	AMOLED, 4.65 inches
WLAN	Wi-Fi 802.11 a/b/g/n

to a central server running the GreenMiner webservice. The current GreenMiner consists of four such identical testbeds to speedup and parallelize the data collection process. Figure 1 shows the innards of one of the four identical settings of the GreenMiner. The GreenMiner service is a continuous testing service whereby users may submit tests to be run and measured. After submitting a batch of tests to the GreenMiner, one of the phones is randomly selected for executing a test. As a result, four different tests can run in parallel to reduce the measurement time. *GreenMiner* maintains the same system state for each test by cleaning any installed apps that ran previously.

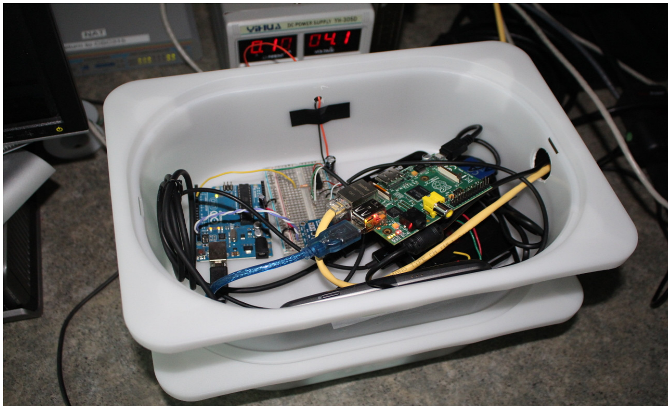


Fig. 1: One of the four identical GreenMiner settings. Photo used with permission from the Green Miner paper [22].

Variations in energy consumption and resource utilization are observed in different measurements for the same test. Consequently, all the *GreenMiner* based previous work repeated any specific measurement multiple times [9,11,15]. Similarly, all of our tests for measuring energy and resource usage were run 10 times and the mean value was used.

2.4 Energy Estimation: GreenOracle

There exists different types of software energy models: instruction based models [18,26], utilization based models [27–29], and others [11,15,19,20] (described in section 12). We followed the philosophy of count based energy modeling similar to our previous *GreenOracle* [11]. *GreenOracle* modeled software energy consumption based on the counts of different resource usages: number of CPU jiffies, number of different invoked system calls and so on. *GreenOracle*, however, has some limitations. This model was built only using 24 Android apps. Although the size of the training set was enlarged by adding different versions from those apps, this did not improve the model significantly. This is not surprising as only one manually written test case was used for each app. As a result, different versions of the same app might have executed the exact sequence in source code, offering very similar information on resource usage and energy consumption for a given test. An accurate energy model, however, requires training from a wide variety of workloads. In this paper, we show that a model based on such small dataset is not accurate in estimating energy consumption of apps from very different domains.

2.5 Energy Model Building Test Generation

To measure the energy consumption of an app we need to drive the app with some kind of test or benchmark. To run an app on *GreenMiner*, we need a test script to replay operations on the app. An example of a test case is: open Firefox, load a Wikipedia page and scroll over the page for five minutes—as if a user is reading the page.

In our previous *GreenOracle* [11] model, we demonstrated that adding new apps in training improves the accuracy of software energy models. Unfortunately, adding more apps requires manually writing test cases for each app, which makes it infeasible to have an energy model trained on hundreds of apps. Manual software testing is difficult and expensive [30,31], which motivated a significant number of research in automated software testing [32–36]. Automatic test generation is when tests are created automatically through algorithmic means to drive the software under test. There has been a significant number of research dedicated for Android test generation [37–41].

Test generation can be completely random [30] such as Android Monkey [42] where random events are injected to an app. Some random test generation strategies, such as Dynodroid [37], extract the layout of the GUI components from an app screen and generate events based on the extracted components. *Search heuristics* are also employed to guide the test generation process, known as Search Based Software Testing (SBST) [32]. For example, code coverage is a *search heuristic* where the objective is to generate test cases to maximize code coverage [43]. Techniques like Sapienz [39] uses multi-objective optimization (maximize fault detection and minimize test sequence

Table 2: A summary of the existing Android testing tools for model building test generation.

Tool	Available?	Works without source code?	Suitable for *GreenScaler*?
AndroidRipper [38]	*Yes*	*Yes*	✓
DroidFuzzer [45]	*No*	*Yes*	
NullIntentFuzzer	*Yes*	*Yes*	✓
IntentFuzzer [46]	*Yes*	*No*	
Monkey [42]	*Yes*	*Yes*	✓
MonkeyLab [47]	*No*	*Yes*	
Dynodroid [37]	*Yes*	*Yes*	✓
ACTEve [48]	*Yes*	*No*	
TrimDroid [41]	*Yes*	*Yes*	*N/A. Generates the whole test suite*
A^{3}E-DFS [49]	*Yes*	*Yes*	✓
SwiftHand [50]	*Yes*	*Yes*	✓
ORBIT [51]	*No*	*No*	
PUMA [52]	*Yes*	*Yes*	✓
EvoDroid [40]	*No*	*No*	
SPAG-C [53]	*No*	*Yes*	
Thor [54]	*Yes*	*Yes*	*N/A. Requires existing test suite*
JPF-Android [55]	*Yes*	*No*	
CrashScope [56]	*No*	*Yes*	
Sapienz [39]	*Yes*	*Yes*	N/A. Authors do not share source code

length) in order to guide the test generation. Genetic algorithms can be used in SBST to find the optimal set of test cases: from a set of candidate solutions (test cases), the test generation process applies mutation on individual candidates, or cross over across two or more candidates, or combination of both to find better solutions [39, 40].

In order to produce the *GreenScaler* model, a model trained on hundreds of AndroZoo apps (described later), a test generation tool is required that is publicly available, and does not require app's source code for test generation; the AndroZoo database only contains the executables (i.e., apks) without any source code. Choudhary *et al.* [44] and Mao *et al.* [39] performed detail surveys on the most well-known Android test generation tools. Table 2 shows the summary of the surveyed tools to identify the ones that are suitable for the *GreenScaler* model building. Only 7 out of the 19 surveyed tools are potentially suitable for the model building test generations. It is important to find the best performing tool among these 7 that can be used for the energy model building test generation. Given the time cost of test generation and the collection of energy and resource usage measurements, it is infeasible to use all the available tools for the energy model building process.

Choudhary *et al.* [44] concluded that evaluation of existing test generation tools can be biased by the apps selected for evaluation. So they did a thorough study on most of the well-known Android test generation tools that are publicly available. The authors found that some tools are hard to use, and might demand continuous communication with the actual authors which is

often not feasible. Interestingly, after their rigorous evaluation, Choudhary *et al.* [44] concluded that random test generation—Monkey and Dynodroid—outperform all the existing Android test generation techniques by a large margin, including all of the *GreenScaler* suitable tools mentioned in Table 2. AndroidRipper exhibits the worst code coverage and require *major effort* to use. PUMA, although requires little effort to use, its code coverage and framework compatibility are very poor compared to others. Instead of the very similar performance, Monkey is much simpler and 5x time faster in test generation than Dynodroid [37]. Unlike Dynodroid, monkey does not have any framework compatibility issue [44], making Monkey as the most suitable tool for generating tests for a wide variety of Android apps.

Monkey, however, is notorious for some of its limitations including app irrelevant events [37,39] like volume control, screen capture etc. Moreover, although Monkey allows setting distributions of different groups of events [42], it does not allow the user to define the distribution of events (e.g., generating 60% `tap` events). Inspired by the success and drawbacks of Monkey, we propose a very similar random test generation tool, GreenMonkey. GreenMonkey is no different than Monkey, except control over the distribution of events is allowed and app irrelevant events such as volume control are discarded. In section 6, we show that this little modification indeed improves model's performance. As a result, we continue our test generations with GreenMonkey instead of Monkey.

With automated test generation, we can generate a test to drive a given app so that we can collect resource usage (independent variables) and energy consumption (dependant variable) to build energy models. However, the generated test case might not be exercising any energy expensive resources, and thus will not provide any useful information for the models. As a result, we aim to generate more than one tests for each app and select the one that has the highest potential of exploiting energy hungry resources. So we need test selection heuristics. In the next section, we evaluate the effectiveness of code coverage heuristic in selecting test cases for energy model building.

3 Code Coverage Heuristic

In traditional software testing, code coverage is one of the most used metrics to evaluate the effectiveness of a test generation approach [30,43,57–59]. In general, a test with higher coverage is expected to reveal more faults in a system [43,59]. However, Inozemtseva and Holmes [60] found that coverage is not strongly correlated with test suite effectiveness for finding faults. In contrast to traditional testing, the objective of our test heuristic is to select test cases that exploit different energy consuming hardware components. A model built on test cases that do not observe energy expensive work, would not be accurate. Such a model would fail to estimate the energy consumption of a foreign app that accesses different energy consuming hardware components.

Table 3: Description of selected apps' master test suite coverage.

Feature	Klaxon	Password Hash	Storyhoard
Source Lines of Code	1601	541	3749
Executable Lines of Code	799	247	1959
Master Suite Size	16	17	75
Class Coverage	24/33 (73%)	11/12 (92%)	56/81 (69%)
Method Coverage	85/150 (57%)	54/60 (90%)	306/497 (62%)
Line Coverage	383/799 (48%)	217/247 (88%)	1252/1959 (64%)
Block Coverage	1937/4378 (44%)	2225/2345 (95%)	5345/8504 (63%)

In this paper, we evaluate the potential of the code coverage heuristic to select tests that can be used for producing energy models. Does covering more code necessarily indicate exercising more energy expensive resources? To answer this question, we investigate the correlation between coverage and power usage. If test suites with high code coverage implies high power usage, then code coverage would be a valid test generation heuristic. Otherwise, different avenues of test generation heuristics would need to be explored. Our methodology is a near replication of Inozemtseva and Holmes [60], but we focus on power usage instead of fault detection ability.

3.1 Methodology

In order to determine the suitability of code coverage as a heuristic for selecting energy consuming tests, we require: 1) a set of Android apps with available test cases; 2) a process to generate test suites of different sizes; 3) coverage and energy consumption of the generated test suites; and 4) a statistical method to calculate the correlation between coverage and power usage.

3.1.1 Selected Applications

The difficulty of finding open source Android apps with JUnit test suites limited the number of potential apps. To match the work of Inozemtseva and Holmes, the apps need to have a coverage of nearly 50% or more for either class, method, line, or block. This further narrowed down the available choices. We finally selected three open source Android apps: Klaxon—a pager; Password Hash—generates passwords; and Storyhoard—a choose your own adventure app. Table 3 shows the characteristics of the selected apps. The varied numbers of lines in source code, number of methods, classes and blocks help better to understand the relationship between code coverage and power usage.

3.1.2 Generating Test Suites

A test suite is a collection of sampled test cases from the master suite. The master suite contains all the test cases for an app written by the developers. Following the similar approach of Inozemtseva and Holmes [60], we generated test suites of different sizes by sampling the existing master suite (collection of all JUnit test cases). For example, a test suite of size 3 means there are 3 test cases in the test suite. In the study of Inozemtseva and Holmes, the selected sizes for generating random test suites were 3, 10, 30, 100, 300, 1000, and 3000. In their subject Java projects, the largest and smallest number of tests were 7,947 and 628 respectively. In our selected Android apps, however, the largest master suite (from the Storyhoard app) has only 75 test cases. As a result, we could not select sizes similar to Inozemtseva and Holmes.

In order to get reliable statistical results by generating large number of test suites, we selected more sizes within short intervals. We started with test suite size 2 and then repeated the procedure with test suites of sizes 4, 7, 10, 13, 16, 27, 40, 64, and 73. Once the sizes were decided, a Python program was written to randomly choose test cases from an app's master suite and create different sized test suites from them. For each test suite size, 100 test suites were generated with random sampling from the master suite. This allowed us to have a diverse collection of test suites of various sizes and coverage levels. Algorithm 1 illustrates the whole process of generating test suites.

Algorithm 1: Generating 100 test suites. Each test suite will have a given number (based on the suite size) of randomly sampled test cases.

```
   input  : master_suite, suite_size
   /* The master suite of an app and a given suite size.
      */
   output: collection_test_suites
   /* 100 test suites each with a fixed number (i.e.,
      suite_size) of randomly sampled test cases.       */
1  collection_test_suites ← [];
2  for suite_number ← 1 to 100 do
3  |  test_suite ← [];
4  |  for test_number ← 1 to suite_size do
5  |  |  test_case ← Random(master_suite);
   |  |  /* select a test case randomly from the master
   |  |     suite.                                       */
6  |  |  test_suite.append(test_case);
7  |  end
8  |  collection_test_suites.append(test_suite);
9  end
10 return collection_test_suites;
```

3.1.3 Capturing coverage and energy consumption of each test suite

In order to capture the coverage of each test suite, we used a third-party tool `emma` [61] on the source code of each app. Emma provides four types of code coverage: line coverage, method coverage, class coverage, and block coverage. We captured all the coverage reports for our analysis.

Energy was measured using the *GreenMiner* by averaging 10 runs of each test suite. However, during the energy measurement process, "*coverage true flag*" was disabled to avoid any overhead incurred from coverage calculation.

3.1.4 Kendall's τ as a measure of effectiveness

We calculated the Kendall's τ correlation coefficients between coverage and power usage for the generated test suites. Kendall's τ does not assume any distributions of the data—unlike Pearson's correlation coefficient it does not assume that the two variables (i.e., coverage and energy consumption in our case) have linear relationship. Similar to Inozemtseva and Holmes, we calculated the coefficients with both uncontrolled and controlled suite size.

Uncontrolled suite size: Combine the measurements of coverage and power usage from all the generated test suites and calculate the Kendall's τ correlation coefficient between coverage and power usage.

Controlled suite size: Combine the measurements of coverage and power usage from the generated test suites with a particular suite size (e.g., all suites with size 2), and calculate the Kendall's τ correlation coefficient.

3.2 Analysis of Results

For uncontrolled suite size, Table 4 shows the Kendall's τ correlation coefficients between code coverage and power usage (*watts*) for all the three apps. Power usage against test suite size is also presented. Table 5 and 6 show the correlations when the suite sizes are fixed to 2 and 13 respectively (i.e., controlled suite size). We did not include results for other suite sizes (e.g., correlations for suite size 4, 7, 10 and so on) as the observations are similar.

We observe good/strong correlation between energy and code coverage, especially when we do not control for test suite size (Table 4). This is not surprising; larger code coverage usually means larger execution time which has direct impact on the total amount of energy consumption (i.e., $E = P \cdot T$). For example, from our results the highest correlation is observed for the Storyhoard app when suite size is not fixed (minimum 0.78 for class coverage and maximum 0.83 for method coverage, Table 4). However, this is the same setup when we observe the highest correlation between energy consumption and test duration (correlation coefficient 0.95). The lowest correlation is found

for the same app when suite size is fixed to 13 (Table 6). Interestingly, for the same setup, the correlation between energy and test duration is the lowest. This implies that good correlation between coverage and energy consumption does not necessarily indicate that the test cases are exercising energy hungry resources—test duration might be the major factor for the observed good correlation. In order to use coverage as the heuristic for energy model building test generations, we need to observe good correlation between coverage and power usage, instead of coverage and energy consumption.

The correlations between coverage and power usage are weak for Password Hash and Storyhoard when suite size is not controlled (Table 4). In case of Klaxon, we observe moderate correlation. However, for controlled suite sizes, the correlation for Klaxon drops significantly (Table 5 and 6). Results from our subject Android apps indicate that covering more code does not necessarily indicate more exercise of power expensive source code portion. This supports the intuition that all code is not equally heavy in power usage. For example, code that makes an `HTTP` request might consume more energy than a larger segment of code without high CPU usage or network operations [62].

Table 4: Correlation between code coverage and power with uncontrolled suite size. Suite size vs. power is also presented.

Applications		Correlation with Coverage				Correlation	
		Method	**Class**	**Line**	**Block**	**Suite size**	**Duration**
Klaxon	**Power**	0.57	0.58	0.58	0.58	0.59	N/A
	Energy	0.67	0.67	0.69	0.69	0.67	0.91
Password Hash	**Power**	0.06	0.08	0.02	0.01	−0.05	N/A
	Energy	0.56	0.53	0.60	0.61	0.68	0.94
Storyhoard	**Power**	0.22	0.23	0.20	0.19	0.20	N/A
	Energy	0.83	0.78	0.82	0.81	0.85	0.95

In order to build *GreenScaler*—modeling software energy against the indirect measurement of resource usage—we need test cases that are more likely

Table 5: Correlation between code coverage and power with suite size fixed to 2.

Applications		Correlation with Coverage				Correlation
		Method	**Class**	**Line**	**Block**	**Duration**
Klaxon	**Power**	0.13	0.40	0.31	0.34	N/A
	Energy	0.12	0.41	0.31	0.35	0.79
Password Hash	**Power**	0.33	0.38	0.31	0.31	N/A
	Energy	0.69	0.67	0.67	0.66	0.85
Storyhoard	**Power**	0.45	0.52	0.35	0.13	N/A
	Energy	0.47	0.51	0.37	0.17	0.90

Table 6: Correlation between code coverage and power with suite size fixed to 13.

Applications		Correlation with Coverage				Correlation
		Method	Class	Line	Block	Duration
Klaxon	**Power**	0.26	0.27	0.26	0.27	N/A
	Energy	0.52	0.51	0.52	0.53	0.81
Password Hash	**Power**	0.21	0.21	0.13	0.13	N/A
	Energy	0.40	0.40	0.23	0.26	0.91
Storyhoard	**Power**	0.26	0.38	0.15	0.13	N/A
	Energy	0.21	0.26	0.13	0.12	0.47

to exploit different resources and energy consuming portions of the code. This short study led us to focus experimentation on resource-utilization heuristics rather than investing time on code coverage based test generation. Building automatic test cases for hundreds of apps with a heuristic, running them on GreenMiner for collecting energy consumption and resource usage counts, and then applying/tuning/validating machine learning models demand several months.

Findings: Code coverage relates more to test duration than to power usage. With code coverage as the heuristic, test with longer execution time might be selected for model building, in spite of its weakness in exercising energy expensive portions of the source code. This implies that coverage will not be a good heuristic if employed for selecting tests to model software energy consumption. We need different heuristics that use actual resource-utilization, and thus capture energy expensive portions of source code.

4 *GreenScaler* Methodology

In this section, we describe the complete *GreenScaler* methodology of using resource-utilization heuristics for generating tests to build continuously refined software energy models.

The process of producing an energy model from a large corpus of energy measurements is to: 1) collect Android apps (section 4.1); 2) generate tests for the collected apps (section 4.2); 3) collect energy consumption measurements, system calls measurements, and other process counters while running an app's test (section 4.3); 4) add measurements to the training corpus; 5) and finally train our model. Figure 2 summarizes the process of developing *GreenScaler*. We also need to evaluate the effectiveness of this process, so model selection (section 4.4), feature engineering (section 4.5), and validation (section 4.6) are also described.

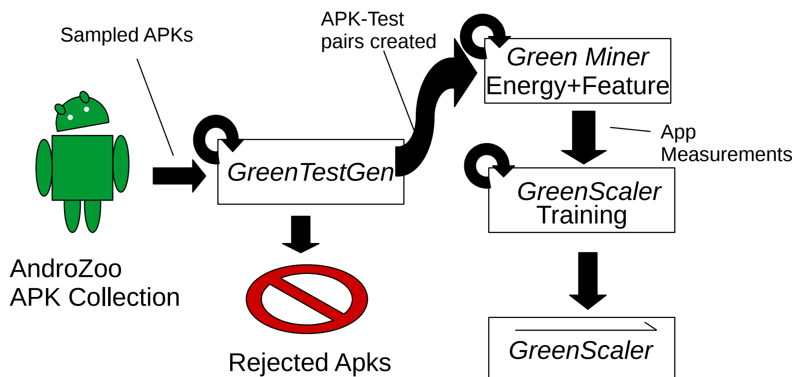


Fig. 2: The process of developing *GreenScaler*. The model learns continuously with new apps using test selection by *GreenTestGen*.

4.1 Collecting Android Applications

We sampled apps randomly using the database provided by AndroZoo [63], and collected about 500 apps. AndroZoo provides millions of apps for Android research that were collected from 3 different app stores: Google Play, Anzhi, and AppChina. Some of the apps did not install or run properly. After removing those, 472 apps were used to develop the proposed *GreenScaler* energy model. AndroZoo does not provide app categories, but using the `screencap` program, described in section 4.2, we manually investigated a sample from the 472 collected apps, and observed apps from different domains: media players, games, utility, etc. Some sampled apps (radio, streaming, and online games) heavily used the network. Table 7 shows the categories of 100 randomly selected apps. The categories of these apps were defined based on two of the authors consensus. The unknown category contains non-English apps that are difficult to categorize. It is clear from Table 7 that our sampled apps are from very different domains. This is important for building a robust energy model that should work across different types of Android apps.

4.2 Automatic Test Generation with Resource-utilization Heuristics

The futility of code coverage heuristic encouraged generating test cases based on resource-utilization heuristics. We evaluate two such heuristics for building software energy models: CPU-utilization and E-heuristic (estimated energy utilization).

CPU-utilization heuristic: Select the test case with the highest CPU time, as CPU is a major source of energy consumption. We can argue that the CPU-utilization heuristic may select test cases that are biased to CPU uti-

Table 7: Categories of the 100 randomly selected AndroZoo apps.

Category	**Count**	**Category**	**Count**
Game & Puzzles	23	Book	6
Utility	11	Entertainment	5
Unknown	11	Media	5
Business & Website	10	Finance	2
Education	9	Health	2
Communication	8	News	1
Tools	7	**Total**	100

lization only and may ignore the utilization of other resources like network, file etc. Consequently, a model built on such test cases might fail to estimate energy consumption of apps with high network or file operations. As a result, we selected another heuristic, E-heuristic, that exploits other energy heavy resources as well.

E-heuristic: Select the test case with the highest estimated energy consumption based on an actual software energy model—*GreenOracle* [11]. In this approach, a test case is selected that has the highest estimated energy consumption based on all the different hardware components utilization. In a nutshell, an existing energy model is used to generate tests towards producing an even more accurate energy model.

For generating tests based on these two heuristics, we propose *GreenTestGen*. For a given app, *GreenTestGen* creates a number of test cases with the help of GreenMonkey. With GreenMonkey, each test case consists of different randomly selected events. *GreenTestGen* runs all the test cases for the app, and selects the one that maximizes a given heuristic function. It is important to note that any test generator (e.g., Monkey, Dynodroid) can be used with *GreenTestGen* just by replacing GreenMonkey. GreenMonkey and *GreenTestGen* are fully depicted in Algorithm 2 and 3 respectively.

Algorithm 2: GreenMonkey

```
input  : no_of_events
output: test_script
1 EVENTS_POOL={adb shell events};
2 test_script ← SequenceOfEvents (no_of_events,
    DISTRIBUTION_OF_EVENTS, EVENTS_POOL);
3 return test_script;
```

Algorithm 3: GreenTestGen

```
   input  : An AndroZoo app, a testGenerator
   /* calls GreenMonkey as the default test generator   */
   output: Test case for the app
 1 if CrashCheck (App)==True then
 2 |  Exit();
 3 end
 4 no_of_events ← Random(10, 40);
 5 play_time ← 0;
 6 max_heuristic_value ← 0;
 7 while play_time≤30 mins do
 8 |  test_script ← testGenerator(no_of_events);
 9 |  heuristic_value ← Execute(App, test_script);
10 |  if heuristic_value>max_heuristic_value then
11 |  |  max_heuristic_value ← heuristic_value;
12 |  |  best_test ← test_script;
13 |  end
14 |  update play_time ;
15 end
16 ScreenCap(App, best_test);
```

GreenMonkey: A pool of `adb` events—such as `tap x y`, `swipe x1 y1 x2 y2`, `text string or number`, `ENTER`, `DEL`, `tapmenu` etc.—was created where the values of pointer locations, strings and numbers are selected randomly (line 1 of Algorithm 2). The pointer locations were restricted to a specific range to avoid clicking on the phones' `HOME` and `BACK` buttons. Different events have different impacts on generating useful test cases. We did manual observations on 30 randomly chosen apps. Not surprisingly, `input tap` was found as the most contributing event toward generating useful test cases. Events like `swipe`, `input text`, and `keyevents` were assigned similar priority, followed by `tapmenu`. So instead of selecting events uniformly randomly, GreenMonkey is biased so that a test script contains more `tap` events than any other events, whereas the event `tapmenu` occurs the least (DISTRIBUTION_OF_EVENTS, line 2). A test script is thus a set of different events—separated by a 2 second sleep time—and the number of events in a test case might differ among the apps.

***GreenTestGen*:** When an app is selected, *GreenTestGen* first checks if the app installs and runs properly (line 1, Algorithm 3). In case of a success, the app is then run to find the best test case—the test that maximizes a selected heuristic function. *GreenTestGen* selects the number of events randomly—from 10 to 40 events so that a test is neither too short nor too long (line 4). A test script that is too short (few events) might not do anything useful, whereas having too many long test scripts would prolong our data collection

period. If all the test scripts are of similar duration (same number of events), any machine learning model would ignore duration as an important feature, which would be devastating for predicting an unknown app's energy with a very different test duration. Each app is then run for a fixed 30 minutes before selecting the best test case (line 7).

GreenTestGen calls a test generator (GreenMonkey as the default, algorithm 2) to create a test case with the selected number of events (line 8). After running the test script and measuring the heuristic value (line 9), *GreenTestGen* creates another test with the previously fixed number of events and run it to evaluate if the heuristic value (e.g., CPU time) is increased, in which case it updates the best test case as the most recent one. At the end, the best test case is the one with the highest heuristic value (line $10-12$). Before running the test on GreenMiner, we also added a 10 seconds idle time at the end of the selected test case. This is to capture any associated tail energy leak [15, 19] that can occur at the end of running the test (tail energy is explained in section 12.1).

After spending 30 minutes to generate the best performing test case, *GreenTestGen* replays the best test case in order to capture the screenshots using `screencap` program and save the images (line 16). This allows us to investigate each app's behaviour and type if needed—construction of Table 7 for instance.

4.3 Collecting Energy Consumption and Resource Usage

For collecting the energy consumption and resource utilization statistics for all the apps, we used *GreenMiner* (described in section 2.3). All of our measurements, for energy and resource usage, were separate from each other so that the actual energy measurements are not affected by the programs capturing resource usage. As mentioned earlier (section 2.3), we repeated each test 10 times and took the mean value of the measurements, for both energy and resource usage.

We used the `strace` program for tracing all the different system calls invoked by an app. The `-c` option was enabled so that we only capture the summary counts of each system call. We also enabled the `-f` option so that system calls invoked by the child processes are captured as well. A script was written that starts running just before the test case of the app under test (AUT) starts its execution. The script then waits and checks for the availability of the AUT in the current running process list. Once it finds the process in the list, it immediately starts the `strace` program with the process *id*. The `strace` program stops and writes the summary counts in a file, when the test case of the app under test finishes.

To capture CPU usage, we used the GNU/Linux proc file system: `/proc/stat` for capturing global information and `/proc/pid/stat` for capturing informa-

tion local to a particular process [11]. These two files provide the CPU time in jiffies both locally and globally, in addition to other pertinent information such as number of context switches, and number of page faults. For capturing global information, we took the difference of counts between after and before running a test. The local information is collected after a test run is completed.

One more important feature that was ignored in some other software models (e.g., *GreenOracle* [11], and PETrA [64]) is an app's interface colour. In case of OLED screen, up to 40% reduction in screen-based energy consumption is achievable by switching interface with white background to dark background [65]. With such dependency on colour, an energy model would be inaccurate if it does not consider screen colour information. We used `screencap` program to capture screenshots while running a test case for an specific app. The script is very similar to the script we used for system calls, except it runs the `screencap` program instead of the `strace` program. Motivated by Dong *et al.* [66], we calculated the average red, green, and blue values (RGB) for all the pixels across all screenshots. Each of these three averages is then multiplied by the test duration—as we model energy consumption instead of power. For example, if we capture three screens, and the average red pixel values are 100, 150, and 200 in those three screens, the calculated red value is 150—i.e., $(100 + 150 + 200)/3$. And the red value used by a model is 150 multiplied by the test duration.

4.4 Algorithms for Energy Models

We have to train a model based on resource usage (independent variables) and energy consumption (dependent variable). We compared three machine learning algorithms and chose the best performing and most interpretable one for our *GreenScaler*: Ridge regression, Lasso, and *Support Vector Regression* (SVR). Ridge regression and Lasso are the simplest of the available regression algorithms and are very similar except their methods of regularization. The biggest advantage with these algorithms are that they are very easy to interpret.

Ridge: Given a set of labelled instances $\{[\mathbf{X}^i, Y^i]\}$, ridge regression finds a coefficient vector $\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_n)$, which can find the best linear fit, $\mathbf{Y_p} = \boldsymbol{\theta}^T X$, where the predicted values $\mathbf{Y_p}$ minimizes the sum of the squared error. This can be formalized as in equation 1 [67]:

$$\boldsymbol{\theta} = \arg\min_{\boldsymbol{\theta}} [\sum_{i=1}^{m} (Y^i - \sum_{j=0}^{n} \theta_j X_j^i)^2 + \lambda \sum_{j=1}^{n} \theta_j^2] \quad (1)$$

In our case, m is the number of apps, n is the number of selected features from the traces of system calls and CPU related information, $\mathbf{X}^i$s are the feature vectors, Y^is are the observed energy consumption, and $\mathbf{Y_p}$ is the vector of

predicted energy consumption. The parameter λ is used for regularization in order to avoid overfitting the training data.

Lasso: One of the characteristics of ridge regression is that it does not eliminate unnecessary features—it retains features with tiny coefficients. Lasso, on the other hand, drops features from a group of highly correlated features. The only mathematical difference between ridge and lasso is the regularization term in equation 1; lasso uses l_1 (i.e., $\sum |\theta_j|$) regularization instead of l_2 (i.e., $\sum \theta_j^2$) [67].

Support Vector Regression: SVR, in contrast to Ridge and Lasso, is more complex and in many cases can exhibit better performance than simple linear regression [68]. Interpretation of such a model, however, is difficult and can be complicated for the developers to find which features are contributing more toward energy consumption. To mitigate this, we only used the linear kernel instead of the more complicated sigmoid, radial basis function (RBF), and polynomial kernels. SVM^{light} implementation was used for SVR that is based on ϵ-SV regression [68] which finds a function $f(x)$ that does not deviate more than ϵ from the ground truth.

4.5 Feature Engineering

Some system calls are similar in functionality. For example, both `fsync` and `fdatasync` do the similar file synchronization work—"synchronize a file's in-core state with storage device"[69]. If we treat these system calls the same, then apps that use either can benefit from training. As a result, similar system calls are grouped together similar to our previous work on GreenOracle [11]. All the grouped system calls are presented in Table 8. In general, if an app invokes 10 `fsync` and 10 `fdatasync`, a new feature Fsync (group name) was used with 20 counts in our model.

Compared to the number of apps, the number of features in our dataset is quite large—22 from CPU and pertinent information, 4 for R, G, B, duration, and 99 from grouped and individual system calls. This large number of features leads to model overfitting. Among the three algorithms we used, SVR is hard to interpret and does not help in feature selection. Lasso with $l1$ regularization yields a more sparse coefficient vector (i.e., many features with coefficient 0) than Ridge, thus more suitable for feature selection. However, with high correlation among features, Lasso selects many features with negative coefficients, which made our previous *GreenOracle* model [11] less interpretable and less accurate.

We addressed this issue with the recursive feature elimination method with Lasso. After the first iteration, we manually removed the features with low coefficients. We followed this procedure until we got a set of features with

Table 8: Grouping similar system calls according to OS semantics.

Groups	**System calls**	**Semantics**
Lseek	lseek, _llseek	"Reposition read/write file offset"
Write	write, pwrite	"Write to a file descriptor"
Writev	writev, pwritev	"Write data into multiple buffer"
Read	read, pread	"Read from a file descriptor"
Readv	readv, preadv	"Read data from multiple buffer"
Open	open, openat	"Open a file"
Statfs	fstatfs64, statfs64, statfs, fstatfs	"Get filesystem statistics"
Stat	lstat64, stat, fstat, lstat, fstat64, stat64	"Get file status"
Fsync	fsync, fdatasync	"Synchronize a file's in-core state with storage device"
Pipe	pipe, pipe2	"Create pipe"
Clone	clone, __clone2	"Create a child process"
Utime	utime, utimes	"Change file last access and modification times"
Dup	dup, dup2, dup3	"Duplicate a file descriptor"

reasonably high coefficients. This procedure subsequently deleted highly correlated features. Only 80% of the measurements from AndroZoo (i.e., 377 randomly selected apps out of 472) were used for feature selection. Table 9 describes the final set of selected features. We got this same set of features for the both resource-utilization heuristics based test sets (CPU-utilization and E-heuristic). This small number of features makes a model easy to interpret.

Learning algorithms perform slowly and suffer from low accuracy with high variance in feature values [70]. In our case, we indeed observed such high variance. This was solved by using 0-1 normalization, as in equation 2, where $\mathbf{x}$ is the actual and $\hat{\mathbf{x}}$ is the normalized feature vector.

$$\hat{\mathbf{x}} = \frac{\mathbf{x} - min(\mathbf{x})}{max(\mathbf{x}) - min(\mathbf{x})} \tag{2}$$

4.6 Testing and Cross Validation

We applied 10-fold cross validation for all the three algorithms to tune the regularization parameters, known as model validation phase before testing with the test data. 10-fold cross validation is helpful when the data size is small. Because it does not require dividing the data into three parts: training data, validation data, and testing data. With 10-fold cross validation, we divided the training data into 10 segments (each containing the same number of apps). At phase i, segment i is used as the test data, whereas the other 9 segments are combined to make the training data. This way, 10 models are evaluated with 10 different test data partitions. We observe the model's performance to see if we need to adjust the regularization parameters (i.e., the penalty sizes). The process stops when no more improvement is possible and when the performance

Table 9: Selected features (CPU and others, duration, colour, and system calls) from feature selection process to model energy consumption for Android apps. This table suggests that the major sources of Android energy consumption are CPU, context switches, test duration, screen color, file operations, and network operations. The weight represents the energy consumption for each unit (e.g., one CPU jiffy) of the features. The weights of each feature are discussed in Section 7.3.

Features	Description	Weights
User	Number of CPU jiffies for normal processes executing in user mode	1.034e-2
Nice	Number of CPU jiffies for niced processes executing in user mode	8.660e-3
CTXT	Total number of context switches	7.604e-5
Major Faults	Number of major page faults for a process	1.117e-2
Duration	Length of the test case in seconds	6.300e-1
Red	Average level of red from screens · duration	5.000e-4
Green	Average level of green from screens · duration	4.000e-4
Blue	Average level of blue from screens · duration	5.200e-4
Fsync	System calls (fsync & fdatasync) to synchronize a file's state to disk	1.310e-3
bind	System call to bind a name to a socket	6.033e-2
recvfrom	System call to receive a message from a socket	5.260e-5
sendto	System call to send a message to a socket	1.761e-2
Dup	System calls (dup, dup2, dup3) to duplicate a file descriptor	5.406e-2
Poll	System calls (poll and ppoll) to wait for some event on a file descriptor	2.920e-3

of the 10 different models are similar. The coefficients of the regularization (e.g., λ for ridge and lasso) for all the three algorithms were finalized during this cross validation phase. It is important to note that, we only used 80% of the apps for the cross validation. Also, for evaluating a model's accuracy (starting from section 5), an app under test was always excluded from the training set.

5 Evaluating Resource Utilization Heuristics

This section evaluates and compares the two resource-utilization based test generation heuristics: CPU-utilization heuristic (CPU time) and E-heuristic (energy estimation).

Using the 472 collected AndroZoo apps, we built two energy models.

- *Model* $_{CPU-H}$: trained with all the selected features from Table 9, but exclusively using measurements from tests generated by the CPU-utilization heuristic.
- *Model* $_{E-H}$: trained with the same feature set, but exclusively using measurements from tests generated by the E-heuristic.

Table 10: 99% mean confidence interval (percent of error in *joules*) of tests versus models. Results suggest that for mean confidence interval both the models have similar accuracy. The difference is negligible/small according to Cliff's delta.

Test Heuristic	**Model $_{CPU-H}$ confidence interval**	**Model $_{E-H}$ confidence interval**	**Wilcoxon p**	**Cliff's delta**
CPU-Heuristic	3.60 - 4.50	4.62 - 5.72	2.20e-16	*negligible*
E-Heuristic	5.36 - 6.26	3.02 - 4.03	2.21e-16	*small*
Combined Tests	4.63 - 5.24	3.90 - 4.68	2.20e-16	*negligible*

Both the models were trained with Lasso because of its superior performance over the others (discussed later). Regularization parameters were used from the cross validation phase (section 4.6). Comparing the accuracy of these two models would tell us which test generation heuristic produces better tests for developing software energy models.

We evaluate the models' accuracy (percent of error in *joules*) following the leave-one-out approach [71]. This ensures that an app under test was never seen in training. Each model was evaluated on two different sets: measurements from the CPU-utilization heuristic based tests and from the E-heuristic tests. This produces two error distributions for each model.

We applied the Anderson-darling normality test [72] and found that none of the error distributions are normally distributed. This led us to select a non-parametric test to decide if the error distributions from *Model* $_{CPU-H}$ and *Model* $_{E-H}$ are statistically different. We used Kruskal-Wallis test [73], which does not assume data is normally distributed, and found that these two models (from two different heuristics) produce statistically different error distributions ($\alpha = 0.05$ and $p = 0.01$). In order to find which model offers better accuracy, we used the pairwise Wilcoxon-rank-sum test [74] to calculate the 99% confidence interval of mean percent error in *joules*. We also calculated Cliff's delta [75] to measure the effect size between the two error distributions (presented in Table 10).

When applied to CPU-utilization heuristic based tests, *Model* $_{CPU-H}$'s mean confidence interval is lower than *Model* $_{E-H}$ with negligible effect (Cliff's delta). *Model* $_{E-H}$, however, similarly outperforms *Model* $_{CPU-H}$, with slightly better accuracy, when evaluated for the E-heuristic based tests. In other words, each model slightly outperforms the other when evaluated on measurements arising from its own test heuristic. We select *Model* $_{CPU-H}$ as *GreenScaler* for the following reasons:

1) For mean confidence interval both the models perform similarly. However, for upper error bound *Model* $_{CPU-H}$ is better. *Model* $_{CPU-H}$ and *Model* $_{E-H}$ estimate 94% and 90% apps within 10% error respectively. Also the mean error of the worst 5% estimations with *Model* $_{CPU-H}$ is 13%, in contrast to

16% for *Model* $_{E-H}$.

2) *Model* $_{CPU-H}$ is built on test cases generated with a simple CPU-utilization heuristic. This is much simpler and easier than a complex energy model heuristic. CPU-utilization heuristic requires only capturing CPU time for a test from the Linux file system. On the other hand, the model based heuristic also requires tracing all the invoked system calls by an app by running a separate `strace` program.

3) We were concerned that CPU-utilization tests would ignore other resources. Thus we compare calls to resources (i.e., system calls) with the Kruskal-Wallis test on the number of `recvfrom` (network receive), `sendto` (network send), and `fsync` (file operations) between CPU-utilization and E-heuristic based generated tests. We found that the distributions of these system call counts between the two test sets are not statistically different ($p >> \alpha$ where $\alpha = 0.05$). This suggests that CPU-utilization heuristic based tests exploit other resources similar to the E-heuristic tests. We further examined a sample of the CPU-utilization heuristic based tests and found that accessing other resources can impact CPU utilization. For example, for an on-line video player, CPU utilization is highest when a test starts playing a video across the network. Thus CPU-utilization based tests *do* use other hardware components.

Findings: Tests based on CPU-utilization heuristic exploit other resources similar to those exercised by the complex E-heuristic based tests. Consequently, energy models built on test cases from both heuristics perform similarly. For simplicity, we recommend CPU-utilization heuristic for generating tests to build software energy models.

6 Monkey vs. GreenMonkey

In section 2.5, we mentioned the drawbacks of Monkey and proposed GreenMonkey to mitigate Monkey's problems. Before generating tests and collecting measurements for 472 AndroZoo apps, we conducted a short study to verify if these little changes can indeed improve the performance of energy model building test generation. We selected 100 random apps for training, and 50 different apps for testing models' accuracy. These are two subsets of the 472 AndroZoo apps. With CPU-utilization test generation heuristic, we used *GreenTestGen* with Monkey and GreenMonkey to generate tests for these 150 apps. Two energy models were built: one with Monkey generated tests and another with GreenMonkey. Both the models were tested on the 50 selected apps. Figure 3 clearly shows that the energy model built on GreenMonkey generated tests outperforms the energy model built on Monkey generated tests. In case of GreenMonkey based model, 90% of the apps' energy was estimated within only 5% error. On the other hand, only 64% of the apps were estimated within 5% error by the model based on Monkey. Table 11 also confirms that this difference is statistically significant.

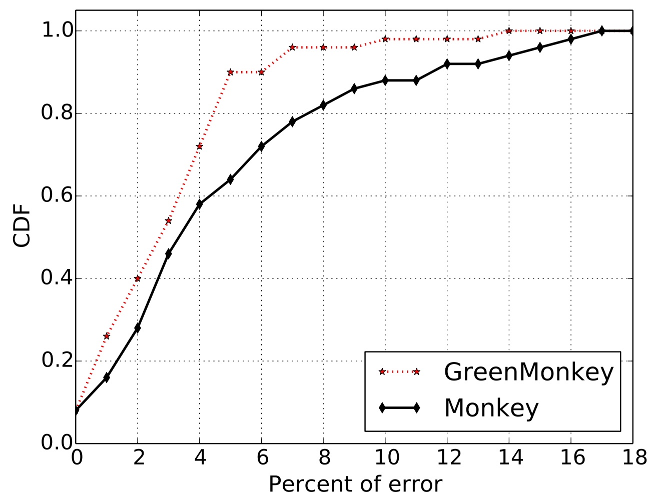


Fig. 3: Comparing performance between Monkey and GreenMonkey for building energy models. GreenMonkey outperforms Monkey in generating tests that are more suitable for building energy models.

Table 11: 99% mean confidence interval (percent of error in joules) of models based on Monkey and GreenMonkey generated tests. Model based on GreenMonkey tests is significantly more accurate than the model based on Monkey generated tests.

Models	**confidence interval**	**Wilcoxon p**
Monkey tests	3.60 - 7.00	7.80e-10
GreenMonkey tests	2.80 - 4.60	7.80e-10

These early findings led us to generate test for all the 472 apps with GreenMonkey instead of the Android Monkey.

7 Evaluating *GreenScaler*

In this section, we evaluate the performance of *GreenScaler* from different aspects: accuracy of *GreenScaler* on randomly generated tests and manually written tests, *GreenScaler*'s ability to explain different sources of energy consumption, and *GreenScaler*'s ability to detect energy regressions. We also observe *GreenScaler*'s sensitivity to the size of changes in SLOC between versions of different apps. This is to check if *GreenScaler* detects energy regression only for large commit sizes. Next, we evaluate *GreenScaler* from developers' perspectives.

7.1 Evaluation on Randomly Generated Tests

We compare the performance of *GreenScaler* (based on CPU-utilization heuristic tests) with the previous *GreenOracle* model [11]. We applied all the three machine learning algorithms from section 4.4.

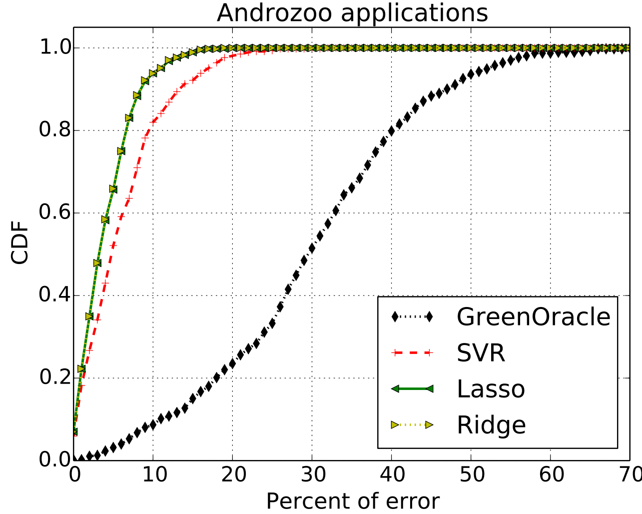


Fig. 4: *GreenScaler* (Lasso) outperforms *GreenOracle* with very large margin on the 472 AndroZoo apps with randomly generated tests.

GreenScaler has significantly less estimation error than *GreenOracle*. Figure 4 shows the Cumulative Distribution Function (CDF) of estimation percentage error in *joules* of our three selected algorithms compared with the previous *GreenOracle* for all the collected AndroZoo apps. The significantly worse performance of *GreenOracle* is not surprising, as it was trained only on 24 apps, which led to the selection of inappropriate features with inaccurate coefficients. With the new set of 472 apps and accurate feature set, all the three models outperform *GreenOracle* by a large margin. Lasso and Ridge perform very similarly and show better accuracy than SVR. In case of Lasso, for example, almost 94% of the apps' energy estimations had an upper bound of 10% error. For outliers, the upper bound was only ≈15% error compared to the 70% worst case error with *GreenOracle*. The indistinguishably similar performance of Lasso and Ridge is because of the very small (close to zero) regularization coefficient obtained from the cross validation phase. With no regularization, there is no difference between Lasso and Ridge. With the very small number of features, none of the models overfit the training set, which led to a negligible regularization coefficient. However, during the feature selection phase, Lasso was very different than Ridge and helped us to find a

good performing feature set. Therefore, we select Lasso for our *GreenScaler*. In other words, the final *GreenScaler* model is built on CPU-heuristic based test generations with Lasso.

7.2 Evaluation on Manually Written Tests

The randomly generated tests with utilization heuristics, although good for energy model building, might not observe any meaningful sequence of actions. The *GreenScaler* model is built on such test cases. So far, we do not know how a model built on random test cases performs on meaningful human written test cases. As a result, it is important to evaluate *GreenScaler*'s performance on human written test cases. Our previous energy model *GreenOracle* [11] is trained on 24 Android apps, with 984 versions in total, where the test cases were written manually. These test cases represent how an average user might interact with these 24 apps, and were written based on the consensus of several computing science grad students from the Software Engineering Research Lab, University of Alberta, Canada. Table 12 shows the test scenarios for all the *GreenOracle* apps. Complete and subsets of *GreenOracle* dataset were used in several published papers on software energy consumption [11, 15, 16, 20, 25, 76].

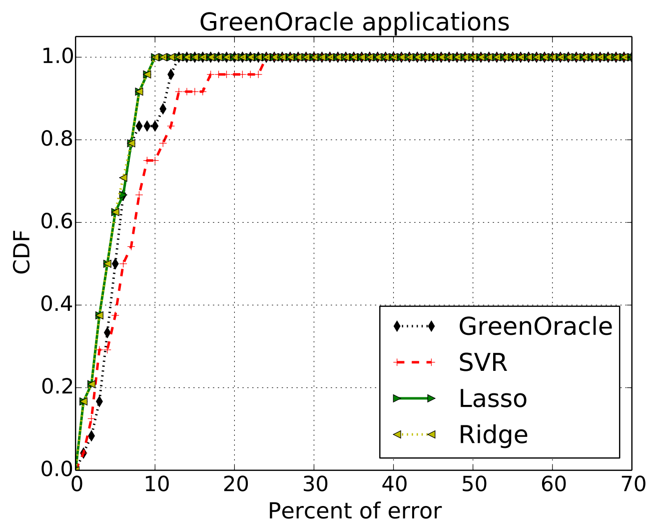


Fig. 5: *GreenScaler* (Lasso) outperforms *GreenOracle* even on *GreenOracle* dataset. Mean error was considered for apps with multiple versions.

Encouragingly, *GreenOracle*, even for its own dataset, is outperformed by *GreenScaler* (Figure 5). The upper error bound for the new model is 10% (i.e., with Lasso), in contrast to the 13% error with *GreenOracle*. This suggests that

Table 12: Description of the *GreenOracle* applications [11]. The table shows the 24 apps in the dataset with their types, numbers of versions, and the execution scenarios of the manually written test cases.

Applications	**Type**	**No. of versions**	**Test Scenario**
Firefox	Browser	156	Loads a Wikipedia page and scrolls over the page.
Calculator	Android Calculator	97	Does simple and complex calculations.
Bomber	Bombing game	79	Starts the game and drops bombs at fixed intervals.
Blockinger	Tetris game	74	Moves, rotates blocks randomly
Wikimedia	Wikipedia mobile	58	Searches and loads the Bangladesh page, and scrolls.
Sensor Readout	Read sensor data	37	Reads and draws graphs for different sensors' data.
Memopad	Free-hand Drawing	52	Opens a canvas, draws an object.
Temaki	To do list	66	Creates, updates, searches, and deletes a to-do list.
2048	Puzzle game	44	Tries different moves to solve the problem.
ChromeShell	Browser	50	Opens a web page and scrolls.
Vector Pinball	Pinball game	54	Throws several balls, and plays with them.
Budget	Manage income & expense	59	Calculates by depositing and withdrawing money.
Acrylic Paint	Finger painting	40	Draws objects.
VLC	Video/Audio player	46	Loads and plays a video for 2 mins.
Eye in Sky	Weather app	1	Searches for Edmonton, and looks for temperature.
AndQuote	Reading quotes	21	Reads some famous quotes.
Face Slim	Connect to Facebook	1	Connects with facebook, and browse the help page.
24game	Arithmetic game	1	plays some random tries.
GnuCash	Money Management	16	Opens an account and saves transactions.
Exodus	Browse 8chan	3	Reads some selected threads.
Agram	Word anagrams	3	Generates single and multiple anagrams.
Paint Electric Sheep	Drawing app	1	Draws objects.
Yelp	Travel & Local app	12	Finds a restaurant and reads users' reviews.
DalvikExplorer	System information	13	Reads system's information.

GreenScaler, although built on measurements from randomly generated test cases, can accurately estimate energy consumption of manually written tests.

7.3 Qualitative Evaluation of *GreenScaler* Model

With the leave-one-out approach, described in Section 5, we have developed 472 different linear models with the CPU-utilization test generation heuristic. These models, however, are almost identical, as excluding one app from training does not affect a model. We chose one of these models randomly to represent the *GreenScaler*. Table 9 shows the final *GreenScaler* energy model (i.e., weights of the selected denormalized features). The accuracy of the model stems from what the table shows. The model suggests that the main sources of energy consumption in Android systems are CPU usage, test duration, screen colour for OLED screen, file operations (`fsync`, `dup`, and `poll`), and data communication (`sendto`, `recvfrom`, and `bind`). The very high coefficients for CPU usage and test duration are similar to the findings of Miranskyy *et al.* [77], who found that often energy consumption was too highly correlated with CPU and

run-time on database systems. According to the model, transmitting (`sendto`) is more expensive than receiving (`recvfrom`), which is complemented by previous research [78]. Moreover, in terms of pixel colour intensity, blue is the most expensive and green is the least expensive, which is also observed by Dong *et al.* [66].

Findings: *GreenScaler* considers many of the known major sources of energy consumption on an smart-phone. This includes CPU time, context switches, page faults, test duration, and interfaces' colors. *GreenScaler* relies on system calls to estimate energy consumption by other components like disk, networks.

7.4 Evaluation on Detecting Energy Regressions

GreenScaler is a model, and similar to any previous energy estimation approaches [18, 19, 64], *GreenScaler* is not 100% accurate. Considering the estimation error, we ask: *will GreenScaler be useful to app developers?* We argue that usefulness is how *GreenScaler* model performs where a developer would actually use it: during the implementation and maintenance of their own app, comparing version against version. In this section, we evaluate *GreenScaler*'s ability to detect energy regression between versions at different levels—two versions separated by a single commit or two versions from two subsequent releases. If *GreenScaler* is successful in detecting energy regression, developers can know if their changes have negative effect on the app's energy consumption.

The main strength of *GreenScaler* is that it maintains similar shape between estimations and ground truths for all the versions of any particular app. We selected four apps from *GreenOracle* dataset that have lots of versions. With multiple versions, we have a separate error distribution function for each app. Figure 6 shows that although *GreenScaler* accuracy varies among the apps, the error distribution is very similar among all the versions for the same app.

This observation is significant: it indicates that *GreenScaler* should accurately estimate the energy consumption difference between two versions of an app. To further demonstrate the adeptness of *GreenScaler* for such cases, we select six apps from *GreenOracle* dataset. Unlike other apps, these six apps contain versions with very different energy profiles. Moreover, two of these six apps (Yelp and Agram) contain versions that are actual releases, whereas versions from other four apps are separated by a single commit.

Figure 7 shows that for all the six apps, *GreenScaler* successfully separates out the energy inefficient versions. For Yelp, a travel & local information app, only one version has a very different energy profile. *GreenScaler* distinguished that version accordingly. Memopad, a drawing app, exhibits three interesting

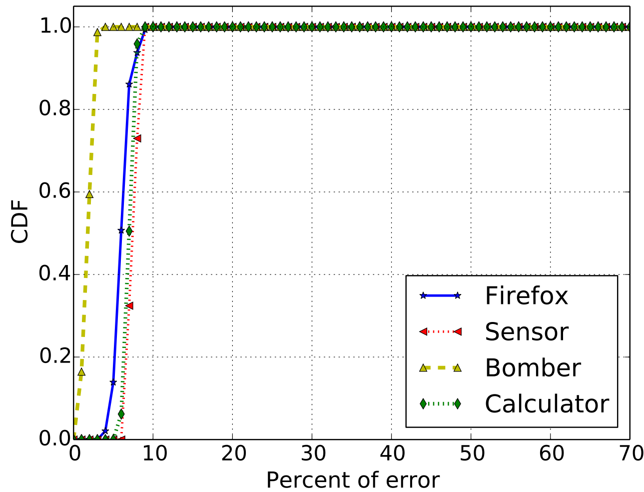
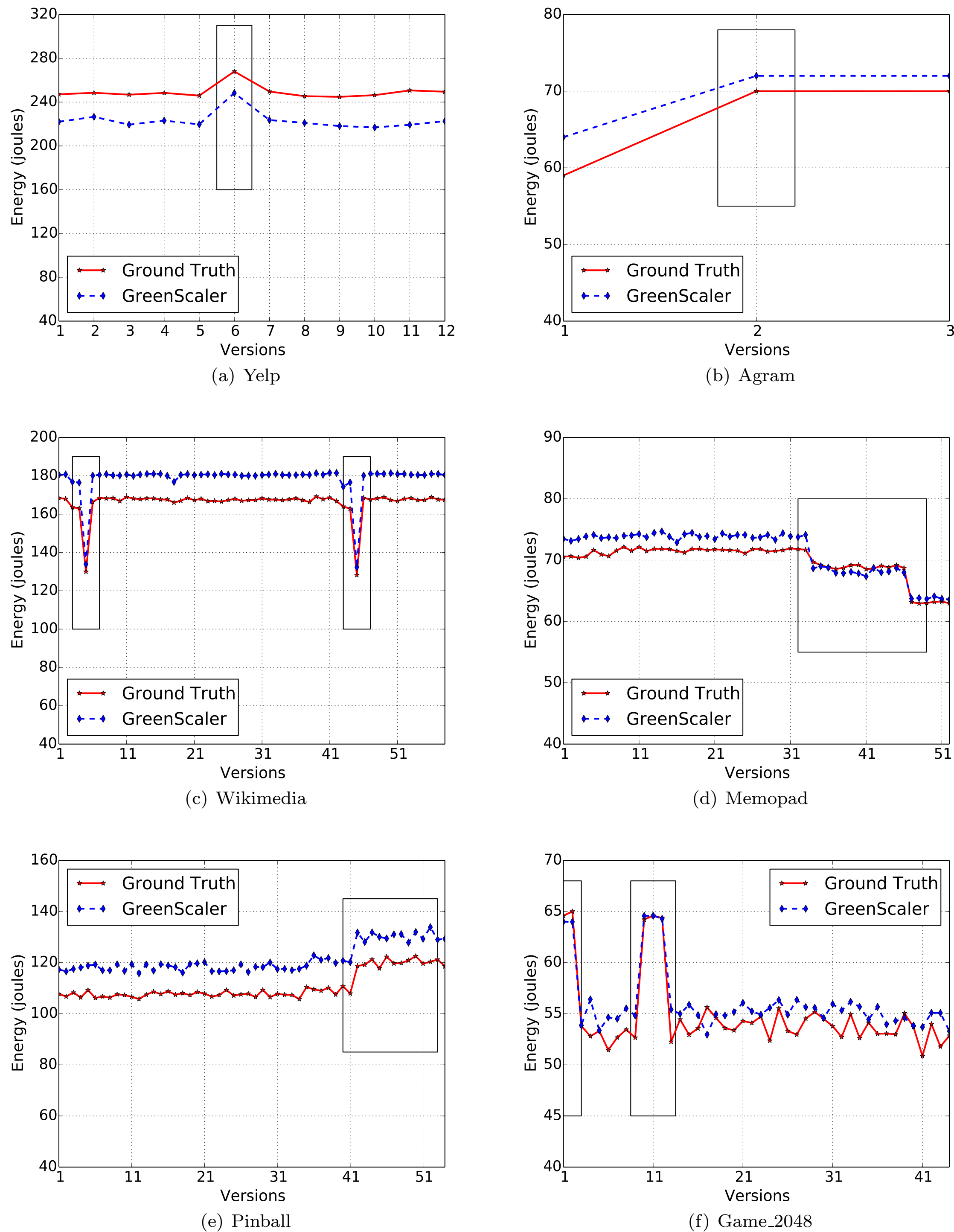


Fig. 6: *GreenScaler* maintains similar error distribution among different versions of an app.

energy profiles throughout its life time—it became more and more energy efficient over time in contrast to Agram and Pinball. Our proposed model accurately distinguished those three phases.

We also investigate if *GreenScaler* can help developers to understand the type of modification that impacted the energy consumption. *GreenScaler* does not locate source code responsible for energy regression. However, the simplistic philosophy of *GreenScaler*—simple counts of different features—helps understanding the type of energy expensive modification. We provide two such examples. For Agram, an app to generate anagrams, the *GreenOracle* dataset contains only three versions. Figure 7(b) shows that version 2 and 3 consume more energy than version 1. *GreenScaler* suggests that the number of context switches has increased significantly from version 1 and stays similar to 2 and 3. Our first impression was that code for thread interaction could have been modified. We used `git diff` and found that Java methods for generating anagrams were indeed synchronized. It is well-known that unoptimized `synchronized` methods fight excessively for shared locks, which leads to more context switches and CPU usage [6, 79]. Similarly, we investigated the continuous improvements of Memopad in terms of energy consumption. The only significant difference in our model among all the versions of Memopad was their RGB counts, clearly suggesting the background colour was changed over time. Indeed we found three distinct background colours. White background (the most expensive for OLED screen) was used for versions up to 33, which was modified to more efficient yellow, followed by even more efficient red. This articulates how significant a simple choice of background colour can be for devices with OLED screens, as also observed by previous research [65].



(a) Yelp (b) Agram

(c) Wikimedia (d) Memopad

(e) Pinball (f) Game_2048

Fig. 7: *GreenScaler*'s efficiency in differentiating between versions with different energy consumption. Versions are sorted based on their committed times. Whenever there is significant energy difference between two versions of the same app, *GreenScaler* detects the difference. Developers can use *GreenScaler* to check for energy regression before releasing a new version.

Findings: *GreenScaler* accurately identifies energy inefficient versions for a given app—for versions separated by a single commit and multiple commits (i.e., subsequent releases). Developers can use our tool [14] to evaluate if a new version of an app is more energy expensive than the previous one. In case of energy regression, they can consult *GreenScaler* to understand the type of modification that might have impacted the energy negatively. For example, if a new version calls `fsync` more than before, a developer can focus on file I/O related code.

7.5 Accuracy vs. Commit Size

For our subject apps, *GreenScaler* was successful in detecting energy regression. However, this does not tell us if *GreenScaler* is sensitive enough to detect regression even when the code change is minimal—e.g., when two versions differ by a single line of source code. *Does GreenScaler's accuracy in detecting regression depend on the commit size?* To answer this question, we used five apps (with all the versions) from section 7.4, as these apps show energy regression with source code modifications. We could not use Yelp as we do not have access to its source code.

We calculated the commit sizes (the sum of the number of additions and number of deletions in SLOC) between all the successive versions. We then calculated the differences between mean energy consumption of the successive versions (difference between mean of 10 energy measurements of version x_i and mean of 10 energy measurements of version x_{i+1}, difference between mean of 10 energy measurements of version x_{i+1} and mean of 10 energy measurements of version x_{i+2}, and so on). Similarly, the differences between mean energy estimations (with *GreenOracle*) of all the successive versions were calculated. This way we calculated the absolute estimation error of *GreenScaler* for a commit size in case of energy regression detection. For example, for a commit size of 10, if the difference in mean energy consumption between two versions of an app is 2 joules and the difference in mean estimated energy consumption is 5 joules, this is a 3 joules of estimation error for a commit size 10. We combined the data from all the five apps and show the result in Figure 8.

Figure 8 suggests that there is no notable relationship between the commit size and *GreenScaler*'s accuracy in detecting energy regression. We also calculated the Kendall's τ correlation coefficient between commit size and absolute error in joules. The coefficient is only 0.08. This implies that *GreenScaler* accuracy does not rely on the commit size. Thus, the model is expected to identify energy regression regardless of the number of changes (small or large) made in the source code. We provide more empirical evidence as follows.

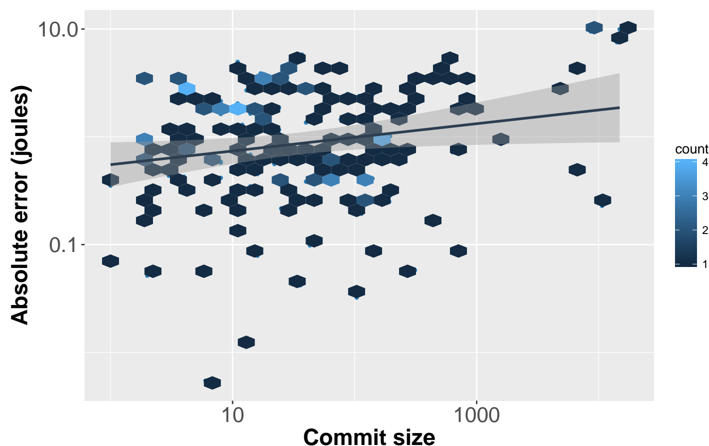


Fig. 8: *GreenScaler*'s sensitivity to commit size in detecting energy regression. Apparently, there is no (or very weak) relation between accuracy and commit size.

7.5.1 GreenScaler's Accuracy in Detecting Energy Regressive API Changes

Can *GreenScaler* detect energy regression for a single API change? We consider two case studies with experiments for the evaluation: 1) changing a Java collection data structure, and 2) changing the HTTP library for HTTP requests.

Hasan *et al.* [9] showed that selecting the most energy efficient Java collection can have enormous effect in reducing software energy consumption. In order to evaluate *GreenScaler*'s sensitivity on a single line of code change, we selected four different Java collections: TreeList from Apache Commons Collections (ACC); TreeMap, LinkedHashMap, and LinkedList from Java Collections Framework (JCF). These four collections have different energy profiles [9] and are suitable for our study. We developed an Android app that has only one activity. This activity does only one thing: insert 50,000 elements into a Java collection. This way, we created four different versions of the app with the four collection APIs. For example, in one version the app inserts 50,000 elements into a TreeMap, and in the next version the TreeMap is replaced by the LinkedHashMap—a single line of source code modification.

We measured the actual energy consumption of these four different versions (10 times each) with the *GreenMiner*, and ranked them according to their mean energy consumption. We then compared this ranking with the ranking obtained from our *GreenScaler* model by taking the mean of 10 energy estimations for each collection. Figure 9 shows the energy consumption ranking (ranking from actual measurements as well from model's estimation) of the four selected Java collections. In case of energy consumption estimation, *GreenScaler* has the smallest percentage of error (2.63%) for TreeList(ACC)

and the largest percentage of error (10.75%) for LinkedHashMap. In spite of these estimation errors, *GreenScaler* is very accurate while detecting energy consumption differences between two successive versions ordered by their ranking, even when the difference is very small. For example, the actual difference between the versions with LinkedList and LinkedHashMap is only 0.5 joules, whereas the difference is 0.56 joules with *GreenScaler*. As we mentioned earlier, this is one of the main strengths of *GreenScaler*—it can identify energy inefficient versions. It is also important to note that although there exists variation in different energy measurements with *GreenMiner* (box-plot), the variation is very small. This is also true for the estimations from *GreenScaler*.

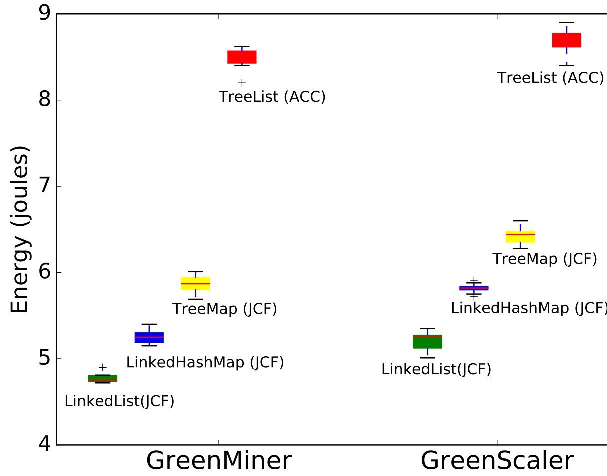


Fig. 9: Comparing Java collections' energy consumption. Actual energy measurement from *GreenMiner* suggests that TreeList (ACC) is the most energy expensive and LinkedList (JCF) is the least energy expensive for inserting 50,000 integer elements. *Greenscaler* suggests the same and comes up with the exact same ranking.

Collections are typically CPU and memory bound, thus we provide another case study that employs the network. We compare the performance of *GreenScaler* in identifying energy consumption difference between two Java HTTP libraries. In this case, our single activity app downloads the homepage of *CNN.com* 20 times at one second interval. One version of the app uses the `Jsoup` library for the HTTP requests, which is replaced by the `URLConnect` library to produce the second version of the app. Figure 10 shows the comparison with 10 energy measurements and 10 energy estimations for each versions. *GreenScaler* estimated the energy consumption within around 12% error (by calculating the mean of energy measurements and estimations) in both cases. However, the estimated difference (around 4 joules between the means) is very

similar to the ground truths. This difference can be significant for an app that continuously communicates over a network.

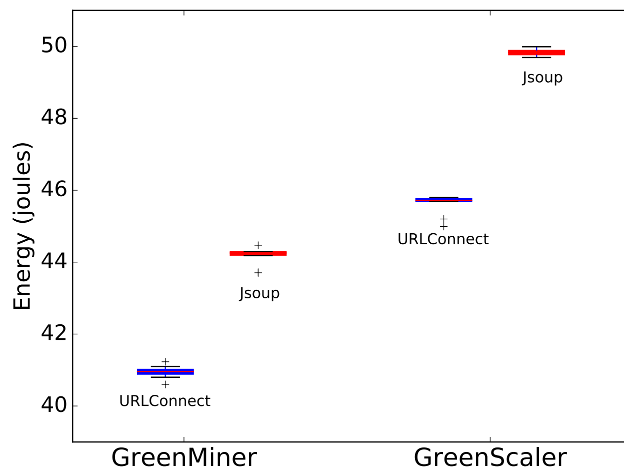


Fig. 10: Comparing Java download libraries: `Jsoup` and `URLConnect`. For the executed test, *GreenScaler* suggests that version with `Jsoup` consumes around 4 joules more than the version with `URLConnect`. This is very similar to the actual measurements from *GreenMiner*.

GreenScaler's ability to detect energy regressive API changes is significant for energy-aware app developers. Previous studies [80, 81] have found that energy-aware app developers should take extra care while selecting and utilizing APIs that are energy hungry—e.g., APIs for making `HTTP` requests. It is evident that *GreenScaler* can help developers for selecting energy efficient APIs. However, to perform accurate comparison between two APIs, developers should write concise test cases that exercise other irrelevant components the least. A very recent work by Song *et al.* [82] discusses about writing such test cases.

7.6 Evaluating *GreenScaler* Tool from Developers' Perspectives

One of the end products of this research is the *GreenScaler* tool [14], a model-based tool support that Android developers should be able to use to estimate their apps' energy consumption. We apply a qualitative evaluation of *GreenScaler* as follows.

1) *Accurate and reliable:* Previous research [44] shows that Android tools' evaluation are biased by the apps used for evaluation. As a result, very different

performance might be observed when those tools are evaluated on a different set of apps. *GreenScaler* is trained and tested on 472 real-world Android apps. In addition, 24 Android apps with 984 versions were tested with *GreenScaler*. To the best of our knowledge, no previous energy model was tested on such a wide variety of apps.

2) *Easy to use: GreenScaler* works without any need of expensive hardware instrumentation. Developers do not even need to instrument their apps to run with *GreenScaler*. The only required tools to run *GreenScaler*, Android Debug Bridge (`adb`) and `aapt`, come with the Android development framework. *GreenScaler* does not suffer from Android framework compatibility issues. As *GreenScaler* works without the need of source code of an app, it works on both native and non-native Android apps. Previous studies observed that these issues make many of the Android tools unusable [44, 83].

3) *Regression detection: GreenScaler* is accurate on detecting energy regression; a developer can immediately verify if an updated app's version is more energy expensive than the previous one. In case of energy regression, developers can make a trade-off between energy efficiency and other functionalities. Energy-aware developers are willing to sacrifice some other features if that helps in reducing energy consumption [13]. In addition to evaluating source code changes, developers can employ *GreenScaler* model with manually created benchmarks to compare energy consumption of different third-party libraries to select the most efficient one.

8 The Importance of More Apps in Training

Will more apps help? Figure 11 shows the reduction in error as more apps are used in training. From 472, 50 apps were sampled randomly as test instances. Using the rest of the apps, accuracy of *GreenScaler* is tested using different training sets with different number of training apps (50, 100, 150 and so on). The accuracy can vary for the same training size based on the selected apps—some apps capture more system calls than others. This is why for each training size x, we repeated each test 100 times with 100 different combinations of x number of apps. Figure 11 shows the combined error distribution for each training size.

Apps with high estimation errors (outliers) exist for all the training sizes. This high error, however, dwindles continuously as we add more apps in our training. In fact, with 400 apps in training the upper error-bound becomes very close to 10%. The dotted line shows the average of the 5 worst estimations with each training size. Although the decay of error rate becomes slow after adding 300 apps in training set, the least number of outliers with 400 apps suggest the possible improvement of *GreenScaler* with adding even more number of apps. Evidently, adding more apps in training improves the upper-error bound of *GreenScaler*. With a large number of apps, we can evaluate the performance of more complex approaches like deep learning. This is why

automatic test generation is so useful. *GreenTestGen* enables a process that allows *GreenScaler* to improve continuously.

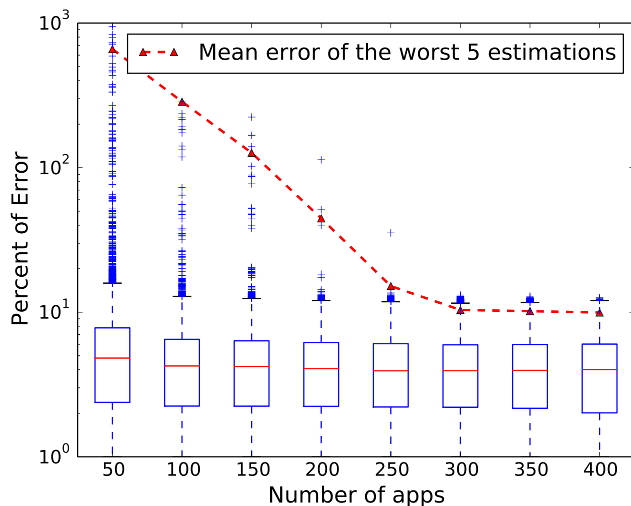


Fig. 11: Model's accuracy against the number of apps used in training. The accuracy improves with more apps in training. This suggests that we can continuously improve the model by adding more apps with the random test generation process.

9 Research Directions for the Software Energy Research Community

We provided empirical evidence on the success of automatic test generation for building software energy models. This observation encourages more research avenues to explore. We provide two such examples.

9.0.1 Domain Specific Energy Models and Deep Learning

In this paper, we focused on building a generic single energy model that can be used to estimate energy of Android apps from any category. With random test generation process aided by test heuristics, we can collect measurements for more apps and build domain specific models. Although the AndroZoo database does not contain app categories, we can collect apps from repositories (e.g., F-droid or Google App Store) where categories are available. Instead of building a single model, a cluster of models (separate models for games, communication, utility and other categories) can be built.

Do domain specific models offer better accuracy than a one-for-all model like *GreenScaler*? We do not know the answer, and that is why it could be an interesting future work. Similarly, how about models that are built on resource usage similarity rather than similarity in app category? With more and more apps in training, should we employ deep learning for building energy models? Deep learning usually requires large training data. While techniques like early stopping and dropout layers [84] can help alleviate the problem, the *GreenScalar* methodology provides a method of continually generating and adding more measurements of more apps to achieve an appropriate amount of measurements for deep learning. This leads to a question about how many apps we need to measure for building a deep learning based model?

9.0.2 Building API Recommendation Systems for Energy-aware Developers

Previous research [85, 86] focused on building API recommender systems for developers. Developers can select an API based on their requirements when multiple options are available. Different metrics can be used while ranking different APIs: documentation, performance, usability, number of users, and number of reported bugs. These recommender systems do not consider energy efficiency of APIs, which might be crucial for the energy-aware developers. Researcher can use *GreenScaler* to build an API recommender system that includes energy efficiency as one of the performance metrics. Results from section 7.5.1 clearly show that *GreenScaler* is adept for such studies.

10 Dataset

For future researchers, we share our dataset publicly [14]. This dataset contains all the selected tests for the 472 AndroZoo apps with the CPU-utilization based heuristic and *GreenOracle* based heuristic—total 944 tests. This tests can be run on GreenMiner in order to explore more research questions and to reproduce our results.

The dataset also contains resource usage and energy consumption for all the apps used for *GreenScaler* and *GreenOracle* models. As we mentioned earlier, each test was run multiple times: 10 times for energy measurements, 10 times for capturing system calls, 10 times for CPU and related measurements, and 5 times for capturing screen shots. The dataset, however, only contains the mean values of each measurements. We have four types of data for the Androzoo apps: 1) AndroZoo apps with resource usage and energy consumption for tests selected with CPU-utilization heuristic that includes all the captured system calls (472 data points), 2) AndroZoo apps with resource usage and energy consumption for tests selected with CPU-utilization heuristic that groups similar system calls together (472 data points). Similarly, there are two more sets of measurements from tests with model-based heuristic (in

total, $472 \cdot 2 = 944$ data points from model-based heuristics). We also shared the *GreenOracle* dataset with and without grouping system calls. This dataset can be used not only to reproduce our results, but also to investigate other machine learning and feature selection techniques for building better models.

Data points: A single data point (as used for model building) represents resource usage and energy consumption of an APK (for an app or for a version). So a single data point is:
Number of cpu jiffies, number of context switches,.., test duration, Red, Green, Blue, number of sento syscalls,.., number of dup system calls,.., Energy consumption.
Here, energy consumption is the dependent variable and all others are the independent variables.

11 Threats to Validity

This section describes the threats to validity of our experiments and results: conclusion validity, construct validity, internal validity, and external validity.

11.1 Conclusion Validity

Some of our conclusions might not be accurate due to the statistical tests we used. Although the tests we used do not assume anything about the distribution of the data, some of them still have their own set of assumptions. For example, the Kendall's τ test assume that there exists no tied rank in the data. Similarly, the Wilcoxon p assumes that the two distributions under test observe similar variance. The results would be inaccurate in case these assumptions are wrong. We mitigated this threat by comparing the actual error distributions with cumulative distribution functions (CDF). Like previous studies [18, 19, 87], we also evaluated our model's accuracy by calculating the percentage of error relative to the ground truths. This method, however, is asymmetric—the error limit in case of under-estimation is 100% whereas for over-estimation it does not have a boundary [88]. This threat was mitigated by showing the actual ground truths and estimated joules when we evaluated *GreenScaler*'s accuracy in detecting energy regression (section 7.4).

11.2 Construct Validity

Modeling software energy consumption is difficult [13]. For example, a CPU can operate at different frequencies and use different power in these states. Consequently, two different apps in spite of utilizing the CPU for the same

duration, might consume different amount of energy based on the triggered CPU frequency level. In our model building process, we relied on the number of CPU jiffies (the time between two clock ticks that can vary) instead of the CPU time directly. Our assumption is that a CPU in higher frequency state would have different number of CPU jiffies than a CPU in lower frequency state. We do not have direct empirical evidence for the accuracy of such assumption. Instead, we relied on the accuracy of the model built on such assumption.

Similarly, our approach for capturing resource usage by tracing system calls can be criticized. Although capturing different system calls usually indicate the types and amount of resources accessed by an app during a test run, there can be some exceptions—direct memory access (DMA) for example.

We also relied on the claim that system call based models do not suffer from tail energy phenomenon [11, 15, 19]. Although the good accuracy of *GreenScaler* across a large number of apps suggest that such a claim might be true, we do not have direct empirical evidence for that. We do not know if the tests we exercised (including *GreenOracle* dataset) were able to observe any tail energy leaks. For that matter, we do not have evidence that any of our subject apps has an execution path that can produce tail energy leaks. Finding if an app actually has tail energy leak requires manually investigating the app's source code to see if it is not doing batch processing (e.g., not sending a number of packets in a single batch) although batch processing option was available.

11.3 Internal Validity

The resource usage we collected are mostly related to a process, such as the number of CPU jiffies used by an app, and system call traces. However, we also used global resource usage, such as the number of context switches, the number of global CPU jiffies. These global resources can be affected by other background processes than the process we are interested in. We mitigated this threat by uninstalling all other optional apps that can impact the global resources. *GreenMiner* also uninstalls an app immediately after running it, even when the next run is scheduled for the same app. This ensures that the current run does not use any stored data from a previous run.

The mapping mechanism—average from system call traces, CPU jiffies, and energy consumption—might not be 100% accurate as little variation between different measurements is observed. Modern mobile devices and their software are not as deterministic as we would hope. There was no direct control over the laboratory temperature that might be harmful for measuring accurate energy consumption. However, INA219's specification [89] suggests that measurements would not be significantly different over the expected laboratory temperature range. We mitigated these two threats by running each scenario 10 times.

The best feature set for modeling Android apps' energy consumption is obtained from a recursive elimination process. There are other feature selection algorithms [90] that might produce a different set of features. However, the selected features with the followed procedure complements earlier findings that CPU, screen, test duration, file operations and network transmissions are the main sources of energy consumption [9,19,62,65,81]. Similarly, energy model from the CPU-utilization based test generation is compared against the model built on tests based on the *GreenOracle* model. Other energy models might produce better tests.

11.4 External Validity

External validity can be criticized for using a single version of Android phone and OS. Architecture independent energy models, however, still remain as open research problem, but there is preliminary work on converting energy models between platforms [91].

The three apps—Storyhoard, Klaxxon, and Password Hash—we used for evaluating code coverage are comparatively older than the AndroZoo apps, and might not cover the latest Android coding features and style. The *GreenScaler* model is trained and tested on 472 AndroZoo apps, with the leave-one-out approach. Although it is possible that this model might fail to estimate the energy consumption of a new app, the chance is low. We also mitigated this threat by testing *GreenScaler* model on the *GreenOracle* dataset. However, from our subject apps, we do not know if any of them was using GPS. Even if we had such an app, our phones were immobile, thus would not reflect the actual usage scenario of apps with GPS usage. Also, we only experimented with Wi-Fi, and do not know how a system call based model would perform for other technologies like 3G, and 4G.

Our model building test generation tool created test cases with random events. Such a tool would fail to exercise app's components where human intervention is required—such as providing correct *id* and *password*. Also, the generated tests might not be meaningful—the tests might drive an app in a very different way than a human user. Our objective, however, was not to develop a tool that can exercise every functionalities of every Android apps. We also did not target to produce meaningful tests. Rather, we investigated if an energy model built on randomly generated test cases can accurately estimate energy consumption of human written meaningful tests. By achieving high accuracy on human written tests (i.e., *GreenOracle* dataset), we believe that our objective is indeed achieved.

12 Related Work

We divide the related previous studies into three areas: modeling software energy consumption, techniques to optimize software energy consumption, and studies related to software energy testing.

12.1 Modeling Energy Consumption

Instruction-based modeling is estimating energy consumption using program instruction cost [18,26]. The basic problem of these approaches is their rigidness to one particular programming language. Energy estimation for apps without source code is not possible with such approaches. *GreenScaler* applies black-box testing and does not require source code. Instruction-based modeling might also require per-instruction power profile, which is not available for all devices [92]. In contrast, *GreenScaler* relies on features that can be accessed from any Linux-based systems.

The most commonly used approach for modeling software energy consumption is the utilization-based approach [27–29,87,93,94]. The basic philosophy is that capturing the usage time of a component with its energy consumption allows modeling its energy profile. Such approaches, however, cannot model tail energy leaks [11,15,19]—energy consumed by a component even after completing its task before becoming inactive (transition time energy consumption). In our models, however, we did not model energy consumption using active usage period of hardware components. Instead, the cumulative counts of different system calls, CPU jiffy, and other OS-level statistics were used. This automatically alleviated the intricacy of separately modeling tail energy for every hardware components. As a result, in contrast to up to 200% error in estimating *joules* with utilization-based approach [8], our count-based model exhibits only ≈15% error in extreme cases.

Pathak *et al.* [19] proposed a complex Finite State Machine (FSM) based model using system call traces. Aggarwal *et al.* [15,20] applied system call counts to predict if energy consumption of different versions differ from each other based on the number of changed system call counts. This model, however, does not offer the actual energy consumption, and thus the developers would not be sure how bad the energy regression incurred from a change in source code is. None of these models consider screen colour and may profile other components inaccurately. The number of apps used for learning and validation was also very small compared to our dataset.

Nucci *et al.* [64] proposed PETrA, an energy estimation tool that leverages Android tools such as `dmtracedump`. As PETrA is the state-of-the-art for estimating energy consumption of Android systems, we wanted to compare *GreenScaler*'s accuracy with PETrA. Unfortunately, PETrA relies on measurements that are not supported by all Android devices. For example, the `batterystats`

program to collect which components were active during an app run, is not supported by the version of Android running on the GreenMiner's Galaxy Nexus phones. We found that the same file could be accessed by using the `batteryinfo` program, but again the provided data was very different than what PETrA expects. We had the same issue with Galaxy Nexus hardware and OS while trying to run other components of PETrA—e.g., `dmtracedump`. We also tried to run PETrA with LG Nexus 5, a close relative of LG Nexus 4 used by PETrA, but failed to produce any results. Again, it was because of the different `batterystats` file. We contacted one of the PETrA authors, and came to know that in order to run PETrA on a different device than LG Nexus 4, we need to re-implement PETrA. The authors are also thinking to make PETrA open source so that such implementation is possible. In contrast to PETrA, *GreenScaler* is already open source and relies on information that are available on any Linux-based systems. Moreover, PETrA heavily relies on the built-in *power_profile.xml* file for getting the current draw for components like CPU, which is not the same as the current from the battery where the voltage is measured. *GreenScaler*, on the other hand, is built on real energy measurements that does not involve any battery information and does not need a battery to run or estimate energy consumption. In worst cases, PETrA's estimation error is more than 50%, especially for apps with high network usage, which is much higher than *GreenScaler*. PETrA also requires manual app instrumentation, which makes it hard to work with hundreds of apps for research purpose. App instrumentation also makes it hard to adopt a tool in a continuous integration system. Also, in contrast to *GreenScaler*, PETrA's performance on detecting energy regression is unknown.

12.2 Energy Optimization

A process of app recommendation based on energy usage is proposed by Saborido *et al.* [95]. A user can select an energy efficient app when multiple apps with the same functionalities are available. With the availability of such recommendation systems, developers would be forced to develop energy efficient apps. In order to help developers optimize their apps' energy consumption, a significant number of research was dedicated on energy optimization techniques and guidelines.

Wake locks are frequently used by Android developers to continue operations even when a device goes to sleep status [96]. Unfortunately, programmers may write code to acquire wake lock that never releases the lock [97]. Pathak *et al.* [98] observed that 70% of energy bugs are related to wake locks. Much research [8,96,97,99–101] has been conducted to characterize, detect, and minimize wake lock bugs.

In a previous work [17], we observed that employing `HTTP/2` server can help significantly in reducing clients' energy consumption. For energy efficient

logging, we showed in a separate study that developers can combine small log messages and write them together to save energy [25].

As screen colour is very sensitive for OLED screen's energy consumption, tools for automatic colour transformation have been developed [65, 102]. In case of video streaming, pre-fetching has been found helpful to save energy [103]. Job off-loading to a server to save energy was also studied [78, 104]. The overhead associated to data off-loading can be so expensive that it might even worsen energy consumption [78].

For reducing tail energy, bundling I/O operations can be effective [17, 19, 62]. Ad-blockers help reducing energy consumption [23], as advertisements are source of significant energy drains [105]. Some studies concentrated on writing energy efficient code during the development phase [81]. For example, energy profiles of the frequently used Java collection framework were studied [9, 12]. Manotas *et al.* [106] developed a framework for automated selection of energy efficient Java collections.

12.3 Energy Testing

Research on software energy testing focused on reporting well-known energy-hungry APIs, and locating energy bugs in a system.

Linares-Vásquez *et al.* [80] reported a list of energy-greedy APIs by studying 55 Android apps. The authors concluded that careful selection and application of these selected APIs can lead to more energy efficient apps. This list of energy-greedy APIs are, however, obtained only from 55 apps and might not be complete in listing all energy-hungry APIs. Moreover, energy efficiency is not only effected by the energy greedy APIs. There are other factors (e.g., tail energy [19], code obfuscation [107], code refactoring [108]), that can impact energy consumption. Our *GreenScaler* model does not estimate energy consumption based on counting energy hungry APIs, and thus do not have such limitations.

Jabbarvand *et al.* [109] proposed a test suite minimization approach for energy testing. The authors hypothesized that tests that covers energy-greedy APIs (using the API list from Linares-Vásquez *et al.*) should be enough to locate energy bugs. In our case, however, we needed to generate test cases from the scratch with no existing test suite. Moreover, our objective was to generate test cases for building energy models, not to locate energy bugs. Finally, in contrast to merely stating a hypothesis, we provided empirical evidence that code coverage is not a good heuristic for generating energy model building tests.

13 Conclusion and Future Work

In this paper, we proposed and showed the value of continuous software energy consumption model building through automatic test generation. This process built *GreenScaler*, an ever improving software energy model. The success of random test generation for building energy models is encouraging. More software energy research can be conducted with our simplistic approach. Our model building approach uses measurements of resource usages that are accessible from any Android systems, and is reproducible for other Android devices.

We demonstrated code coverage's irrelevance to power usage. In fact, code coverage correlates more with test run-time than with power usage. Instead of code coverage, we built energy models using automatic test generation by two resource-utilization heuristics: CPU-utilization and E-heuristic (software energy model estimation). We found that simple CPU-utilization heuristic exhibits similar performance to a more complex model based heuristic in generating tests to produce energy models.

There is a clear relationship between the number of apps measured and the upper error-bound of count-based software energy consumption models. By automating formerly manual-labour intensive testing work, we can continuously produce ever more accurate models that can be used by developers with no hardware-based instrumentation. We also demonstrated that these models work well in the relative case whereby version to version the model successfully predicts changes in energy consumption of an app undergoing modification. We shared our *GreenScaler* tool so that developers can have direct feedback on energy consumption without dealing with hardware instrumentation [14].

Future work includes scaling up this app measurement approach even further, so that approaches like deep learning and domain specific energy modelling can be studied. We hope that the idea of energy consumption test heuristics excites other researchers as well, as there is a need for more investigation into test generation heuristics that are good for energy modelling—perhaps other energy models serve as better heuristics than CPU-time heuristics. We used random search, other forms of search such as genetic algorithms might prove fruitful. We do not yet know the bounds of this model, perhaps there is a true saturation point. Questions left unanswered include: "what is the effect of more tests per app on a model", and "what are other features we should be measuring?"

Acknowledgements

Shaiful Chowdhury is grateful to the Alberta Innovates - Technology Futures (AITF) to support his PhD research. Abram Hindle is supported by an NSERC Discovery Grant. Stephanie Borle was supported by an NSERC Undergraduate Student Research Award.

References

1. C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about the energy consumption of software?," *IEEE Software*, pp. 83–89, 2015.
2. Hern and a. Alex, "Smartphone now most popular way to browse internet ofcom report." https://www.theguardian.com/technology/2015/aug/06/smartphones-most-popular-way-to-browse-internet-ofcom/, 2015. (last accessed: 2016-Jul-29).
3. V. Woollaston, "customers really want better battery life." http://www.dailymail.co.uk/sciencetech/article-2715860/, 2014. (last accessed: 2015-APR-22).
4. B. Jones, "Microsoft has found the source of recent surface pro 3 battery woes." http://www.digitaltrends.com/computing/microsoft-surface-pro-3-battery-getting-patch/, 2016. (last accessed: 2016-Jul-30).
5. H. Malik, P. Zhao, and M. Godfrey, "Going green: An exploratory analysis of energy-related questions," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pp. 418–421, 2015.
6. G. Pinto, F. Castor, and Y. D. Liu, "Mining Questions About Software Energy Consumption," in *MSR 2014*, pp. 22–31, 2014.
7. S. A. Chowdhury and A. Hindle, "Characterizing energy-aware software projects: Are they different?," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pp. 508–511, 2016.
8. A. Pathak, Y. C. Hu, and M. Zhang, "Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof," in *EuroSys '12*, (Bern, Switzerland), pp. 29–42, April 2012.
9. S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 225–236, 2016.
10. D. Li and W. G. J. Halfond, "Optimizing energy of http requests in android applications," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*, DeMobile 2015, pp. 25–28, 2015.
11. S. A. Chowdhury and A. Hindle, "Greenoracle: Estimating software energy consumption with energy measurement corpora," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pp. 49–60, 2016.
12. R. Pereira, M. Couto, J. a. Saraiva, J. Cunha, and J. a. P. Fernandes, "The influence of the java collection framework on overall energy consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, pp. 15–21, 2016.
13. I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, and J. Clause, "An empirical study of practitioners' perspectives on green software engineering," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 237–248, 2016.
14. S. Chowdhury, S. Gil, S. Romansky, and A. Hindle, "Greenscaler-tools-and-data." https://github.com/shaifulcse/GreenScaler-Tools-and-Data, 2017.
15. K. Aggarwal, C. Zhang, J. C. Campbell, A. Hindle, and E. Stroulia, "The Power of System Call Traces: Predicting the Software Energy Consumption Impact of Changes," in *CASCON '14*, 2014.
16. S. Chowdhury, K. Luke, J. Toukir, Imam Mohomed, S. Varun, K. Aggarwal, A. Hindle, and G. Russell, "A System-call based Model of Software Energy Consumption without Hardware Instrumentation," in *IGSC '15*, (Las Vegas, US), December 2015.
17. S. Chowdhury, S. Varun, and A. Hindle, "Client-side Energy Efficiency of HTTP/2 for Web and Mobile App Developers," in *SANER '16*, (Osaka, Japan), March 2016.
18. S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption Using Program Analysis," in *ICSE '13*, pp. 92–101, 2013.
19. A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained Power Modeling for Smartphones Using System Call Tracing," in *EuroSys '11*, (Salzburg, Austria), pp. 153–168, April 2011.

20. K. Aggarwal, A. Hindle, and E. Stroulia, "Greenadvisor: A tool for analyzing the impact of software evolution on energy consumption," in *2015 IEEE ICSME*, (Bremen, Germany), pp. 311–320, Sept 2015.
21. Jiffy, "Linux Man Page." `http://man7.org/linux/man-pages/man7/time.7.html`, 2016. (last accessed: 2016-Jan-10).
22. A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework," in *MSR 2014*, (Hyderabad, India), pp. 12–21, May 2014.
23. K. Rasmussen, A. Wilson, and A. Hindle, "Green Mining: Energy Consumption of Advertisement Blocking Methods," in *GREENS 2014*, (Hyderabad, India), pp. 38–45, June 2014.
24. A. Hindle, "Green Mining: Investigating Power Consumption Across Versions," in *ICSE '12*, pp. 1301–1304, June 2012.
25. S. A. Chowdhury, S. Nardo, A. Hindle, and Z. Jiang, "An exploratory study on assessing the energy impact of logging on android applications," *Accepted in Empirical Software Engineering Journal*, 2017.
26. C. Seo, S. Malek, and N. Medvidovic, "Component-level energy consumption estimation for distributed java-based software systems," in *Lecture Notes in Computer Science*, vol. 5282 of *Lecture Notes in Computer Science*, pp. 97–113, Springer Berlin Heidelberg, 2008.
27. A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *Proceedings of the USENIXATC'10*, 2010.
28. A. Shye, B. Scholbrock, and G. Memik, "Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures," in *IEEE/ACM MICRO 42*, (New York, NY, USA), pp. 168–178, December 2009.
29. S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, "Using Complete Machine Simulation for Software Power Estimation: The SoftWatt Approach," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pp. 141–150, 2002.
30. S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. Mcminn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Softw.*, vol. 86, pp. 1978–2001, Aug. 2013.
31. P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pp. 165–176, 2016.
32. M. Harman, Y. Jia, and Y. Zhang, "Achievements, open problems and challenges for search based software testing," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–12, April 2015.
33. M. Pradel, M. Huggler, and T. R. Gross, "Performance regression testing of concurrent classes," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pp. 13–25, 2014.
34. C. Zhang and A. Hindle, "A green miner's dataset: Mining the impact of software change on energy consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, pp. 400–403, 2014.
35. R. Yandrapally, S. Thummalapenta, S. Sinha, and S. Chandra, "Robust test automation using contextual clues," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pp. 304–314, 2014.
36. S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 201–211, Nov 2015.
37. A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pp. 224–234, 2013.
38. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pp. 258–261, 2012.

39. K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, (New York, NY, USA), pp. 94–105, ACM, 2016.
40. R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pp. 599–609, 2014.
41. N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in gui testing of android applications," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 559–570, 2016.
42. Monkey, "UI/Application Exerciser Monkey." `https://developer.android.com/studio/test/monkey.html`. (last accessed: 2016-May-11).
43. R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 72–82, 2014.
44. S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?," in *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, (Washington, DC, USA), pp. 429–440, 2015.
45. H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proceedings of International Conference on Advances in Mobile Computing & Multimedia*, MoMM '13, pp. 68:68–68:74, 2013.
46. R. Sasnauskas and J. Regehr, "Intent fuzzer: Crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, pp. 1–5, 2014.
47. M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "Mining android app usages for generating actionable gui-based execution scenarios," in *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pp. 111–122, 2015.
48. S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pp. 59:1–59:11, 2012.
49. T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pp. 641–660, 2013.
50. W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pp. 623–640, 2013.
51. W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, FASE'13, pp. 250–265, 2013.
52. S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pp. 204–217, 2014.
53. Y. D. Lin, J. F. Rojas, E. T. H. Chu, and Y. C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for android devices," *IEEE Transactions on Software Engineering*, vol. 40, pp. 957–970, Oct 2014.
54. C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proceedings of the ISSTA 2015*, ISSTA 2015, pp. 83–93, 2015.
55. H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using java pathfinder," *SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, Nov. 2012.
56. K. Moran, M. Linares-Vsquez, C. Bernal-Crdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 33–44, April 2016.

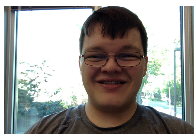
57. P. Boonstoppel, C. Cadar, and D. Engler, "Rwset: Attacking path explosion in constraint-based test generation," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 351–366, Springer, 2008.
58. S. Anand, P. Godefroid, and N. Tillmann, "Demand-driven compositional symbolic execution," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 367–381, Springer, 2008.
59. A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the ISSTA '09*, pp. 57–68, 2009.
60. L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 435–445, 2014.
61. Emma, "EMMA: a free Java code coverage tool." `http://emma.sourceforge.net/`, 2006. (last accessed: 2016-JUL-22).
62. D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated energy optimization of http requests for mobile applications," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pp. 249–260, 2016.
63. K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pp. 468–471, 2016.
64. D. D. Nucci, F. Palomba, A. Prota, A. Panichella, A. Zaidman, and A. D. Lucia, "Software-based energy profiling of android apps: Simple, efficient and reliable?," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 103–114, Feb 2017.
65. D. Li, A. H. Tran, and W. G. J. Halfond, "Making Web Applications More Energy Efficient for OLED Smartphones," in *ICSE 2014*, (Hyderabad, India), pp. 527–538, June 2014.
66. M. Dong, Y.-S. K. Choi, and L. Zhong, "Power modeling of graphical user interfaces on oled displays," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pp. 652–657, 2009.
67. T. Hastie, R. Tibshirani, and J. Friedman, "Linear methods for regression," in *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics, 2001.
68. V. Vapnik, *The nature of statistical learning theory.* Springer, 2000.
69. Linux man-pages project, "Intro linux man page." `http://linux.die.net/man/2/intro`, 2016.
70. C. wei Hsu, C. chung Chang, and C. jen Lin, "A practical guide to support vector classification," 2010.
71. T. Hastie, R. Tibshirani, and J. Friedman, "Model assessment and selection," in *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer Series in Statistics, 2001.
72. H. C. Thode, *Testing for normality*, vol. 164. CRC press, 2002.
73. M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods.* John Wiley & Sons, 2013.
74. D. F. Bauer, "Constructing confidence sets using rank statistics," *Journal of the American Statistical Association*, vol. 67, no. 339, pp. 687–690, 1972.
75. N. Cliff, *Ordinal methods for behavioral data analysis.* Psychology Press, 2014.
76. S. Romansky, S. A. Chowdhury, A. Hindle, N. Borle, and R. Greiner, ""deep green: modelling time-series of software energy consumption," in *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME) (accepted)*, 2017.
77. A. Miranskyy, Z. Al-zanbouri, D. Godwin, and B. Bener, "Database engines: Evolution of greenness," *Journal of Software: Evolution and Process*, vol. 30, no. 4, 2018.
78. A. P. Miettinen and J. K. Nurminen, "Energy Efficiency of Mobile Clients in Cloud Computing," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, (Boston, MA, USA), June 2010.
79. StackOverflow, "Why are synchronize expensive in Java?." `http://stackoverflow.com/questions/1671089/why-are-synchronize-expensive-in-java`, 2009. (last accessed: 2016-Jul-22).

80. M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: An empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pp. 2–11, 2014.
81. D. Li, S. Hao, J. Gui, and W. G. J. Halfond, "An Empirical Study of the Energy Consumption of Android Applications," in *Proceedings of the 2014 IEEE ICSME*, (Victoria, BC, Canada), pp. 121–130, September 2014.
82. W. Song, X. Qian, and J. Huang, "Ehbdroid: Beyond gui testing for android applications," in *ASE 2017*, pp. 27–37, 2017.
83. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, (New York, NY, USA), pp. 259–269, ACM, 2014.
84. I. Goodfellow, Y. Bengio, and A. Courville, *Regularization for Deep Learning.* MIT Press, 2016. http://www.deeplearningbook.org.
85. G. Uddin and F. Khomh, "Automatic summarization of api reviews," in *ASE 2017*, pp. 159–170, 2017.
86. Y. M. Mileva, V. Dallmeier, and A. Zeller, "Mining api popularity," in *Testing – Practice and Research Techniques*, pp. 173–180, 2010.
87. L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones," in *Proceedings of the 8th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, 2010.
88. M. Shepperd, M. Cartwright, and G. Kadoda, "On building prediction systems for software engineers," *Empirical Software Engineering*, vol. 5, pp. 175–182, Nov 2000.
89. Texas Instruments, Dallas, USA, *INA219 Zer-Drift, Bidirectional Current/Power Monitor With I2C Interface*, December 2015. http://www.ti.com/lit/ds/symlink/ina219.pdf.
90. M. Karagiannopoulos, D. Anyfantis, S. B. Kotsiantis, and P. E. Pintelas, "Feature Selection for Regression Problems." http://www.math.upatras.gr/~dany/Downloads/hercma07.pdf. (last accessed: 2015-Oct-22).
91. C. Zhang, "The impact of user choice on energy consumption," *MSc. thesis, University of Alberta*, 2013.
92. R. W. Ahmad, A. Gani, S. H. A. Hamid, F. Xia, and M. Shiraz, "A review on mobile application energy profiling: Taxonomy, state-of-the-art, and open research issues," *Journal of Network and Computer Applications*, vol. 58, pp. 42 – 59, 2015.
93. J. Flinn and M. Satyanarayanan, "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications," in *WMCSA '99*, (New Orleans, Louisiana, USA), pp. 2–10, February 1999.
94. M. Dong and L. Zhong, "Self-constructive High-rate System Energy Modeling for Battery-powered Mobile Systems," in *Proceedings of the MobiSys '11*, pp. 335–348, June 2011.
95. R. Saborido, G. Beltrame, F. Khomh, E. Alba, and G. Antoniol, "Optimizing user experience in choosing android applications," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 438–448, March 2016.
96. Y. Liu, C. Xu, S. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *FSE 2014*, (Seattle, WA, USA), Nov 2016.
97. F. Alam, P. R. Panda, N. Tripathi, N. Sharma, and S. Narayan, "Energy optimization in android applications through wakelock placement," in *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1–4, March 2014.
98. A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, pp. 5:1–5:6, 2011.
99. Banerjee, Abhijeet and Chong, Lee Kee and Chattopadhyay, Sudipta and Roychoudhury, Abhik, "Detecting Energy Bugs and Hotspots in Mobile Apps," in *FSE 2014*, (Hong Kong, China), pp. 588–598, Novemeber 2014.

100. X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "How Speedy is SPDY?," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, (Seattle, WA, USA), pp. 387–399, April 2014.
101. P. S. Patil, J. Doshi, and D. Ambawade, "Reducing power consumption of smart device by proper management of wakelocks," in *Advance Computing Conference (IACC), 2015 IEEE International*, pp. 883–887, June 2015.
102. M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of guis in android apps: A multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pp. 143–154, 2015.
103. N. Gautam, H. Petander, and J. Noel, "A Comparison of the Cost and Energy Efficiency of Prefetching and Streaming of Mobile Video," in *Proceedings of the 5th Workshop on Mobile Video*, MoVid '13, (Oslo, Norway), pp. 7–12, February 2013.
104. M. Othman and S. Hailes, "Power Conservation Strategy for Mobile Computers Using Load Sharing," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 2, pp. 44–51, January 1998.
105. J. Gui, D. Li, M. Wan, and W. G. J. Halfond, "Lightweight measurement and estimation of mobile ad energy consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*, GREENS '16, pp. 1–7, 2016.
106. I. Manotas, L. Pollock, and J. Clause, "Seeds: A software engineer's energy-optimization decision support framework," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pp. 503–514, 2014.
107. C. Sahin, P. Tornquist, R. McKenna, Z. Pearson, and J. Clause, "How Does Code Obfuscation Impact Energy Usage?," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014.
108. C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2014.
109. R. Jabbarvand, A. Sadeghi, H. Bagheri, and S. Malek, "Energy-aware test-suite minimization for android apps," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pp. 425–436, 2016.

Shaiful Chowdhury is a PhD candidate in Computing Science at the University of Alberta, Canada. Previously, he received his MSc and BSc degrees in Computer Science from University of Saskatchewan, Canada and University of Chittagong, Bangladesh respectively. Shaifuls research interest includes software energy modeling and efficiency, mining software repositories, and applications of machine learning in software engineering. Shaiful won the Early Achievement Award in PhD (Computing Science) at the University of Alberta. He also received the mining challenge paper award at MSR 2015.

Stephanie Borle received her BSc in Computing Science from the University of Alberta. During this time she was awarded an NSERC USRA which gave her the opportunity to research software engineering topics. Currently, Stephanie is working toward an MSc degree in Speech-Language Pathology at the University of Alberta.



Stephen Romansky is a Computing Science MSc student at the University of Alberta where he completed his BSc and will start a PhD. In his MSc, Stephen applied machine learning to the task of predicting software energy consumption. Stephen's research interests include Software Engineering.

Abram Hindle is an associate professor of Computing Science at the University of Alberta. His research focuses on problems relating to mining software repositories, improving software engineering-oriented information retrieval with contextual information, the impact of software maintenance on software energy consumption (Green Mining), and how software engineering informs computer music. He likes applying machine learning in music, art, and science. Sadly Abram has no taste in music and produces reprehensible sounding noise using his software development abilities. He has published several papers in international conferences and journals, including EMSE, ICSE, FSE, ICSM, MSR, and SANER. He has served on the program committees of several international conferences, and has program co-chaired MSR and SCAM. Abram received a PhD in computer science from the University of Waterloo, and Masters and Bachelors in Computer Science from the University of Victoria. `http://softwareprocess.ca`