The Power of System Call Traces: Predicting the Software Energy Consumption Impact of Changes

Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia

Department of Computing Science
University of Alberta
Edmonton, Canada
{kaggarwa, chenlei.zhang, joshua2, abram.hindle, stroulia}@ualberta.ca

Abstract

Battery is a critical resource for smartphones. Software developers as the builders and maintainers of applications, are responsible for updating and deploying energy efficient applications to end users. Unfortunately, the impact of software change on energy consumption is still unclear. Estimation based on software metrics has proved difficult. As energy consumption profiling requires special infrastructure, developers have difficulty assessing the impact of their actions on energy consumption. System calls are the interface between applications and the OS kernel and provide insight into how software utilizes hardware and software resources. As profiling system calls requires no specialized infrastructure, unlike energy consumption, it is much easier for the developers to track changes to system calls. Thus we relate software change to energy consumption by tracing the changes in an application's pattern of system call invocations. We find that significant changes to system call profiles often induce significant changes in energy consumption.

1 Introduction

Software tends to undergo constant development. Even during maintenance periods, bug fixes and new features are committed to a project. Developers understand that with change comes risk, especially the risk of performance regressions. One kind of performance that is difficult for developers to measure or predict is software energy consumption [1].

Due to this difficulty there has been much effort put into studying, explaining, and estimating software power use, especially on mobile platforms. Researchers have tried to model energy consumption on smartphones by creating energy consumption models based on hardware components [2], system run-time statistics [3], finite state machines [4], and byte code instruction usage [5]. However, none of the studies listed above have investigated the impact of software change on application energy consumption based on actual commit histories of software projects. Hindle *et. al.* [6] has made the first step toward revealing the relationship between software change and energy consumption. Much is to be learned by measuring energy consumption of multiple software versions and describing potential correlations between software metrics and energy consumption. Hindle *et. al.* [1] also created a dedicated test bed called *Green Miner*, and demonstrated that one requires multiple test runs to reliably calculate the energy consumption of an application on smartphones. This kind of testing methodology is useful but it requires specialized equipment and large amounts of time to obtain reliable data.

We want to help developers estimate if their source code changes cause changes in their soft-

Copyright ©2014 Karan Aggarwal, Chenlei Zhang, Joshua Charles Campbell, Abram Hindle, and Eleni Stroulia. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

ware's energy profile: the profile of energy consumption during runtime. Thus we attempt to rely on dynamic analysis of test cases to estimate power use changes caused by software change. In other words, we gather metrics while software is undergoing a use case test.

The dynamic analysis used for energy consumption is tracing (recording) system calls during application execution. Software running on modern *Operating Systems* (OSes) uses system calls to interact with the services provided by an OS's kernel. Examples of such services include: access to hardware devices, network communication, communication with other applications, requesting and releasing system resources, and receiving notifications of system events. We use records of system calls to model software energy consumption over multiple versions of two applications. Our results point towards a relationship between the change in energy profile and change in the system call invocation profile.

The benefit of estimating changes in power use based on system call profiles is that system calls are easy for developers to measure. A system call profiler *strace* is available on Linux platforms, including Android. Similar software is available on all major OSes: Apple's OSX provides *dtrace* and Microsoft provides *NT Kernel Logger* for Windows. It is easy to run and profile an application with `strace`. One can simply invoke `strace` and their application from the command line: `strace -c yourapp`. Using `strace` takes less labour than developing a hardware test bed instrumented with a power monitor and running tests multiple times to get a statistically reliable estimate of power use. Thus, if we can predict changes in power use based on system call profiles then developers may avoid the expensive hassle of measuring, remeasuring, and estimating the energy consumption of their applications directly. Developers can detect possible energy consumption regressions with simple system call profiling of their application.

This study makes four important contributions:

- First, it demonstrates that system call profiles can be used to model changes in energy consumption profiles for Android applications.
- Second, it lays out a dynamic analysis method of relating software change to energy consumption by tracing system calls.
- Third, an experiment is performed to contrast the different energy consumption behaviours in terms of multiple software versions and different test cases.
- Fourth, a simple rule of thumb is proposed, tested and verified: *a significant change in system call counts often implies a significant change in power use*.

strace, `http://sourceforge.net/projects/strace/`

2 Background and Related Work

This section reviews system calls and relevant work. The related work is organized into three categories consisting of attempts to build models of energy consumption of software applications: utilization-based power models, instruction-based power models, and system-call-based power models. Furthermore, research about combining software energy performance with mining software repositories techniques is discussed. Energy refers to the cost or ability to do work while power is the rate of energy use. Power is measured in watts (W) while joules (J) are the cumulative watt-seconds measured to do work. If a process uses 4W of power for 10 seconds it consumes 40J of energy. Mean-watts are often used when tests are of the same length of time. Mean-watts is the average power used for any moment of a task. Mean-watts multiplied by seconds produces the energy consumed in unites of joules during a test.

2.1 System Calls

System calls form an API for user-space applications to access the services, abstractions and devices that are managed by the kernel and the rest of the OS. System calls are standard functions provided by the OS kernel to user processes.

Usually such system calls are provided for: communicating with the hardware (for example, accessing the hard disk), creating and executing new processes, managing memory use, sending and receiving data to and from other processes, and receiving notifications for events [7]. Figure 1 presents the relationships among user applications,

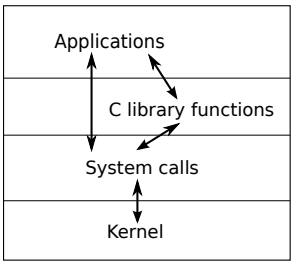

Figure 1: This diagram shows how applications, C library functions, system calls, and the kernel interact with each other.

C library functions, system calls and the OS kernel [7]. System calls are typically called by a library, such as the C standard library, graphics libraries, network libraries, inter-process communication (IPC) libraries, and occasionally by the application itself.

One can expect different versions of software to invoke different system calls during their execution if they differ from each other in terms of the services they use and how they access these services. As an essential interface sitting between the application and the OS kernel that triggers hardware utilization and other kernel services, we ask *can system calls provide a set of features for making predictions about software energy consumption?*

2.2 Utilization-Based Power Models

Utilization-based power models model full system energy consumption by profiling each individual component's energy consumption in two steps. The first step is to collect utilization statistics and the corresponding energy consumption when running a list of applications under a sample of scenarios. The second step is to apply regression analysis, usually linear regression, to train a model for software energy consumption based on utilization data. This model is then used to predict energy consumption.

Flinn *et al.* [8] model the energy consumption of each component in a software system by designing and implementing a tool called PowerScope. By combining the current measurement and each process in the system, PowerScope generates the profile of power use of each component in an application. PowerScope is mostly based on the CPU profile and the power model is device specific.

Gurumurthi *et al.* [9] built a complete system power simulator, which is called SoftWatt. It was implemented on the SimOS infrastructure and is able to model the CPU, memory, and disk use of the targeted application. SoftWatt uses resource usage statistics from SimOS to estimate power statistics.

Shye *et al.* [10] studied the energy consumption behaviour in an Android smartphone, the HTC Dream, with data from real user activity. The authors deploy a logger application that is able to collect system performance metrics and user activity on an Android phone, and report it back to a central server.

Carroll *et al.* [11] analyzed the energy consumption of an Android smartphone, the Openmoko Neo Freerunner. Six typical usage scenarios of a smartphone, audio playback, video playback, text messaging, voice calls, emailing, and web browsing, were tested on the smartphone to gather the distribution of energy consumption in each component. Their analysis was very limited to the Android smartphone they used.

Zhang *et al.* [2] generated two online power models for Android smartphones, PowerTutor and PowerBooter. PowerTutor is based on the combination of power models for components in Android smartphones. The authors measured the energy consumption of selected hardware components on an Android smartphone and built linear regression models for each of them. As the power models generated by PowerTutor vary significantly between different modules of Android smartphones, they also proposed a general power model: PowerBooter. PowerBooter is based on the discharge curve of the lithium-ion battery in a specific Android smartphone. PowerTutor has better accuracy compared with PowerBooter, however they both have shortcomings. PowerTutor is specific to Android smartphone modules and PowerBooter needs the discharge curve of the battery on each Android smartphone.

Dong *et al.* [3] implemented a self-constructive energy model for Linux-based mobile systems. The energy model, called Sesame, generates energy models for mobile systems without external power measurement. Sesame collects system statistics and applies the Advanced Configuration and Power Interface (ACPI) to gather the predictors for the power model. Energy readings are collected through the smart battery interface, and linear re-

gression power models are generated based on the collected data.

Mittal *et al.* [12] proposed and implemented a power model, WattsOn, for the mobile device emulator on the Windows Phone platform. It builds upon a set of power models that focus on individual components, such as cellular network (3G), WiFi network, display, and CPU. Application developers are shown the energy consumption of each component in order to make better decisions while implementing an application. They also apply resource scaling to generalize WattsOn for real phones to overcome the measurement differences between the emulator and phone.

As mentioned in Balasubramanian *et al.* [13] and Pathak *et al.* [4], some of the components (NIC, 3G, and SD Card) exhibit a tail energy phenomenon. Tail energy usage occurs when components stay in a high power state after the completion of an operation even though they are no longer being used. Thus, utilization-based power models are unable to model the tail energy phenomenon. Compared with utilization-based power models, our methodology is based on tracing system calls, which is able to overcome the problem posed by the tail energy phenomenon [4].

2.3 Instruction-Based Power Models

For applications running in a JVM, a list of research papers have taken a different approach: utilizing the Java bytecode instructions to build energy models for software systems.

Seo *et al.* [14, 15] have implemented an energy consumption model for Java-based software systems running on distributed devices. This power model consists of three components, computational energy cost, communication energy cost, and infrastructure energy overhead. This energy consumption model makes accurate estimates which fall within 5% of the actual energy cost for an application. However, the model is highly dependent on the hardware and JVM.

Hao *et al.* [16] built an energy consumption model, *eCalc*, for Android application CPU energy consumption at two levels: the whole program and the method. The approach in *eCalc* is similar to Seo *et al.* [15].

An extension of *eCalc* implemented by Hao *et al.* [5], called *eLens*, combined program analysis with instruction-based power modelling. *eLens* is able to estimate the energy consumption of hardware components besides the CPU and has fine-grained energy profiling on multiple levels. Li *et. al.* [17] extended this work by profiling bytecode instructions, and estimating the energy consumption of each line of source code. Their profiler application, *vLens*, is able to estimate energy consumption with high accuracy.

Instruction-based power models are often designed for software running in a JVM. Some of the applications in this paper do not run in a JVM.

2.4 System-Call-Based Power Model

A system-call-based power model was proposed by Pathak *et al.* [4]. They applied system call tracing to model the energy consumption of applications running on smartphones. First, they studied the power behaviour of components in a smartphone and showed that: 1) several components have tail power states (a component stays in high power state for a period of time after use); 2) system calls that do not imply utilization can change power states; and 3) several components do not have quantitative utilization. They conclude that energy linear models based on correlating utilization with energy consumption are not accurate. Second, they built energy models by tracing system calls using three steps. The first step was to model the power states and generate finite state machines (FSMs) of each system call for each component in a smartphone. The second step was to integrate all the FSMs of system calls to model a FSM for each individual component. In the final step the FSM model of the smartphone was developed based on the FSMs in second step. Finally, based on the FSM of a certain smartphone, they can identify the state that the system is currently in and estimate the energy consumption of an application. The authors have implemented FSMs for several Windows and Android smartphones. Their results show improved accuracy compared to an approach [10] based on linear regression modelling. Pathak *et al.* [18] extended their prior work and implemented a fine-grained energy profiler for smartphone using FSMs. This energy profiler, *eprof*, can work on both Android

and Windows Mobile phones to estimate the energy consumption of smartphone apps.

Similar to this system-call-based power model, we also trace system calls and correlate them with software energy consumption. However, this work relies on the aggregate count of system call invocations and studies system call profiles across multiple versions of existing products.

2.5 Mining Software Repositories

Mining software repositories (MSR) research seeks to enable software engineers and researchers to base their decisions on data mined from software repositories such as version control systems, bug trackers, and project documentation. MSR research applies statistical analysis, data mining, machine learning, and other automated techniques to extract rich data from software repositories in order to discover interesting and actionable information about software systems [19]. With the help of this historical information, knowledge can be acquired about software development processes and characteristics and used to inform decisions. MSR techniques have been successfully applied to many areas such as predicting software defects and validating the effectiveness of development behaviours. Only a few papers have tried to leverage MSR techniques to understand software energy performance.

Gupta *et al.* [20] studied the energy consumption of a Windows phone. They combined power traces and execution logs to build power models. Specifically, a power trace is the measurement of power use on a phone during a test session by a power meter. The execution log is a sequence of active executable files and shared libraries during a test session. Using the combined data set, they created linear regression models and predicted the energy consumption of application running on a Windows phone. They also applied techniques such as decision trees to detect energy patterns within the data set.

Hindle [6] provided a detailed methodology, called *Green Mining*, to collect energy consumption data for applications over multiple versions on Linux-based systems. Based on power measurements and software metrics, the author studied the correlation between software change and energy consumption over multiple application versions. Although the correlation between software metrics and energy consumption is very low, *Green Mining* points to a promising research direction: combining power measurement with MSR techniques.

Hindle *et al.* [1] also created a test bed called *Green Miner*, which has a dedicated hardware infrastructure to measure the energy consumption of Android applications across multiple versions. They also demonstrated that multiple test runs are required to reliably calculate the energy consumption of an application on a smartphone. *Green Miner* can measure up to 50 power readings every second. A constant voltage of 4.1V is provided to the phone which draws varying current according to phone's requirements. This test bed was used for the case studies presented in this paper.

The case studies presented here are based on the methodology of *Green Mining*, extending this methodology to find the correlation between software change and energy consumption via system call tracing.

This paper applies the approach of Pathak *et al.* [4, 18] and Hindle [6] to find the relationship between system call traces and energy consumption across multiple versions of applications. Here, instead of using FSMs, the number of system call invocations are used to build various types of models. Counting system call invocations is easier than making FSMs.

3 Methodology

This section describes the procedure for creating test scripts and collecting the energy and system call statistics for multiple versions of two applications. Data was collected using the following procedure:

1. Select the application, and build multiple versions;
2. write the test script to drive the application;
3. configure the Green Miner; and
4. collect and analyze the results.

3.1 Selecting the Application and Building Multiple Versions

Two open source applications with multiple versions were chosen for these tests: `Calculator`

and Firefox for Android. Both applications are widely used, Calculator for calculations and Firefox for browsing the web. 101 Calculator versions were retrieved from its GitHub repository. These versions were commited between January 1, 2013 and February 5, 2013. Using the Android NDK tools 101 Calculator APKs (Android Packages) were built. APKs are like Java jar packages or Debian deb OS packages, except that APKs are used only in Android.

Similarly, Firefox application versions were obtained by building the nightly commits in the Firefox repository from March 7, 2011 to November 17, 2011. 156 versions of Firefox were built, each separated by one commit.

3.2 Devising the Test Sequence

The test scripts were created to simulate realistic usage of the application by an average user. The Calculator application is used for calculations that a user might execute daily such as unit conversions or tax calculations. The Calculator application test does some simple conversion calculations such as converting miles to kilometres, gallons to litres, calculating tax amounts and solving an equation using the quadratic formula. This test has a duration of 125 seconds.

The primary use of Firefox is to browse web pages full of multimedia such as images. To simulate an average user reading a web page with the Firefox application, the script opens the Wikipedia page about American Idol, and emulates reading action for 4 minutes. The script scrolls the page by swiping the page down and clicks on the screen at regular intervals of 10 seconds to prevent the phone from sleeping. The webpage is stored on a local server to be loaded by Firefox. This prevents the webpage from changing and also prevents varying server response times and network congestion from affecting the data collected.

3.3 Configuring the Green Miner

The *Green Miner* test bed [1] was used for testing the applications. *Green Miner* is a test bed consisting of four Android clients. Each client consists of a *Galaxy Nexus* phone controlled by a Raspberry Pi that starts the tests, collects and uploads the data onto a centralized server and an Arduino for measuring energy consumption. In order to emulate a user using the phones in the test bed, the screen pixel positions of each of the touch events is recorded. For example, to emulate a tap action on the phone, the pixel position on the screen for the tap is recorded. Similarly, for a swipe action, the starting and ending pixels on the screen are recorded. The emulator provides the option to record the pixel positions at each tap or swipe on the emulator screen. The Android emulator is available with the standard Android Studio platform.

Green Miner takes this sequence in the form of a script that is used to replay the sequence of taps and swipes on an actual phone. It runs the sequence on the test bed phone by injecting touch events into the phone's software, from the script file. It creates touch events on the phone, swiping actions on the phone screen and text entry on the phone's on-screen keypad. For system call tracing, strace is employed, and cross-compiled for Android. The strace -c option is used to obtain system call counts.

3.4 Running the Tests

As there is a variation in power measurements due to factors unrelated to the software being tested, multiple tests for the same version need to be run. In these experiments, 10 tests per version were run. From these tests the mean energy consumption is calculated for each version. Because strace adds its own energy overhead during the tests, separate test runs are required for collecting the system call data. 10 runs per version are run with the strace tool in the background to obtain the system call profiles. There were a total of 2020 and 3120 test runs for the Calculator and Firefox applications, respectively. A total time of 400 hours was needed to run these tests.

4 Results and Analysis

In this section, the results from the experiments on Calculator and Firefox are presented and analyzed.

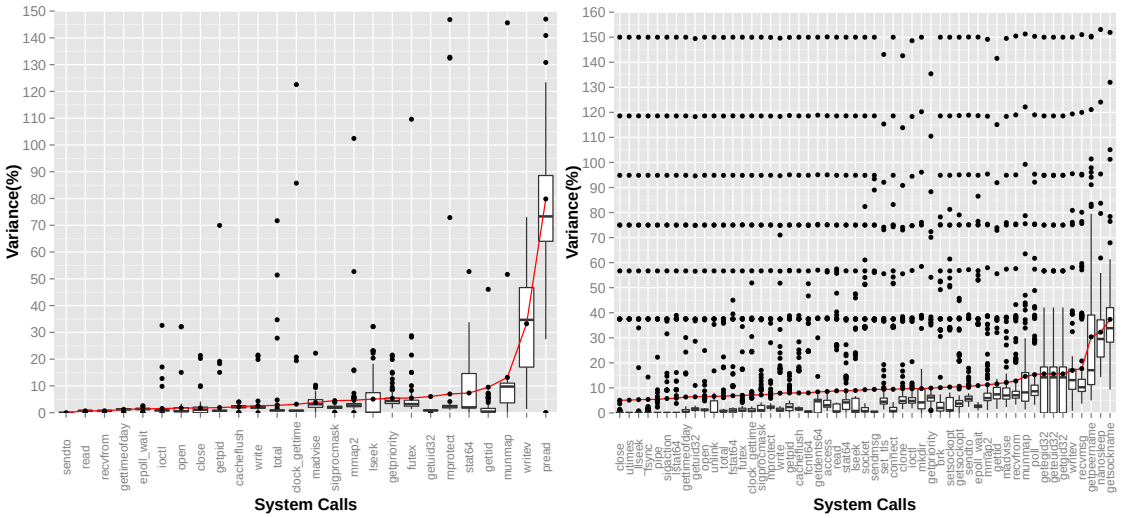

Figure 2: **Variance as a percentage of mean value across 10 runs per system call per version for the `Calculator` (left) and `Firefox` (right) applications. The X axis refers to the system calls, while the Y axis refers to variance in percentage.**

4.1 System Call Profile Stability

In our tests, the `Calculator` application invoked 46 different system calls and `Firefox` invoked 91 different system calls. The system calls that had on an average less than 10 calls per version were filtered out, leaving 25 system calls for `Calculator` and 53 for `Firefox`. Though it is generally expected that system call counts are stable across different runs, the number of system call invocations showed some variation. Different system calls have different variances. Figure 2 shows the average of the variances of the 10 counts for each version over all versions of `Calculator` and `Firefox`, relative to the average of the means.

Calulator 14 of the 25 system calls had an average variance $\leq$ 3% between versions. However, three system calls, `munmap`, `pread`, and `writeev` have average variances of 13%, 79%, and 33%, respectively, across all versions. 4 system calls had an average variance between 3% and 5%. Finally, 4 system calls had an average variance between 5% and 10%. The total system call count, the sum total of all system calls, has a mean variance of 2% across all versions.

Firefox 33 of the 52 system calls had a mean variance between 5% and 10% and the remaining 19 had a mean variance $\geq$ 10% when variances were averaged over all versions. The `Firefox` application shows much higher variability than the `Calculator` application.

4.2 Energy Consumption

The power distribution as shown in figure 3 and figure 4 were obtained by running the tests on *Green Miner*. In order to determine whether the differences between any two versions were statistically significant, pairwise Student's *t*-tests were performed for each pair of `Calculator` and `Firefox` versions, as shown in figure 5.

Calculator Figure 3 depicts a sudden drop in energy consumption at versions 73, 74, 82, and 84. Figure 5 shows that for versions after version 45, many versions are statistically different than the versions before version 45. The authors looked within the code repository and determined that this is because the screen display density was reduced in version 45, leading to a slight difference in the energy consumed. Versions 73, 74, 82 and 84 are versions that failed during testing and form their own cluster. These four versions are significantly different from all the other versions in terms of energy consumption. The errors in these four versions could have been the result of two large, incomplete refactoring efforts, one which started at version 73 and another which started at version 82.

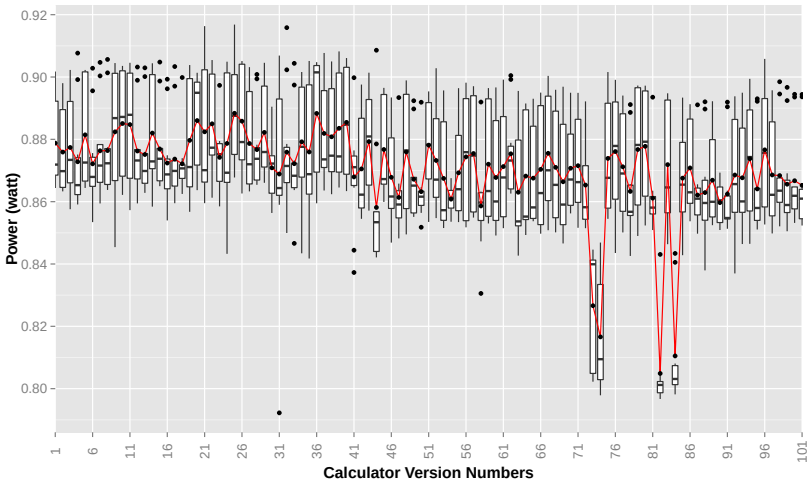

Figure 3: **Box plots of the mean watts used for each version of the `Calculator` application. Each of the 101 versions was measured 10 times. The X axis represents the version numbers and the Y axis represents the energy consumption.**

Firefox Versions after version 56 are significantly different from the versions before version 56 as shown in figure 5 and 4. Also, version 46 is significantly different from versions 1 to 56 in terms of power use. In version 56, a bug was fixed which added `.net` at the end of the domain name whenever the active tab was switched before loading the website by removing the buggy code. This contributed to the reduced energy consumption in versions 56 onwards. Power use increases again after version 70, when functionality to check for plugin crashes was added. Versions 77 to 86 contained a bug that was introduced during refactoring and was finally fixed in version 87. Versions 98 to 100 contained bugs that crashed `Firefox` completely.

4.3 Linear Regression

In order to test the relationship between energy consumption and system call counts, two models were used. First, a linear model was created using the ordinary least-squares error (OLS) regression method for individual system calls, relating the mean energy consumption and mean system call counts for that system call.

Calculator A large number of the 25 system calls are highly correlated with each other: most system calls have a high Spearman's correlation coefficient ($|\rho| \geq 0.8$) with at least one other system call. Hence, for 25 system calls, models were constructed relating energy consumption to the number of invocations of that system call. A linear model of the form $y = b_1 \cdot x + b_0$ was created, where y is the mean power use of a version, x is the mean system call count for a particular system call for that version, b_0 is the intercept, and b_1 is the slope coefficient. 10 such models are depicted in table 1. Table 1 also shows the mean number of invocations of the system call across the versions as $\bar{x}$, and mean power contribution of the system call, $b_1 \cdot \bar{x}$, as estimated from the linear models in the last column.

Selected system calls are shown in table 1 and their descriptions in table 2. 12 out of 25 of the system calls correlate with (Spearman's $|\rho| \geq 0.3$) the energy consumption, while the other 12 have $|\rho|$ values between 0 and 0.3 with 1 being negatively related to energy consumption. However, all 25 system calls have high R^2 values. 20 of the 25 models had $R^2 \geq 0.6$. A model relating the mean sum total of all system call counts to power was also constructed, as shown in Table 1. Sum count of all system calls correlates weakly ($|\rho| = 0.37$) but its linear relationship has a higher R^2 value of 0.74. Though the linear model coefficient b_1 for almost all the system calls is small, they are statistically significantly different from zero with p-value ($\leq 1 \times 10^{-12}$). The coefficients are small because the number of call invocations are high and the units used (mean watts) are large. As the linear models have robust R^2 values, they capture a rela-

tionship between the system call counts and energy consumption of the `Calculator` application.

Firefox Only 10 out of 53 of the system calls are mildly correlated (Spearman's $|\rho| \geq 0.3$) to energy consumption, while 36 have $|\rho|$ values between 0 and 0.3 and the remaining 7 are negatively correlated. However, most of the linear models have low R^2 values, most of them with R^2 less than 0.3. A linear model based on the sum total of all system calls has a correlation of 0.606 and R^2 value of 0.31. The coefficient, b_1 is statistically significant with very small p-values($\leq 1\times10^{-12}$).

10 selected linear models are shown in table 3 and the system calls used are described in table 4. Table 3 also shows the mean number of invocations of each system call across all versions as $\bar{x}$ and the mean power contribution of each system call estimated by the linear models ($b_1 \cdot \bar{x}$) in the last column.

4.4 Logistic Regression

Second, a logistic regression model was constructed by relating a significant change in power (compared to a reference) to a significant change in system call counts (compared to the same reference).

Developers are concerned about the effect that their code changes have on their application's energy consumption. Logistic regression models were created to address the question, "*can we predict whether our application's energy consumption changes or not by using changes in system call counts?*"

To use logistic regression a binary classification is required. Thus each application version's energy consumption is characterized as *high* or *low*. High energy consumption is defined as being more than the mean, and low energy consumption is defined as being less than the mean. Using the mean system call counts from all versions as a reference, the difference between the system call count for each version and the reference is calculated. This difference is used as the independent variable in the logistic regression.

Most system calls are highly correlated with other system calls, thus a model with a reduced number of system call counts as features can be constructed. An iterative approach was used to choose the appropriate independent variables (system call counts) for the model. This approach is to add system calls as independent variables one by one. If all system calls in the model were significant the new system call was kept as an independent variable. If not all system calls were significant to the predictions of the model, the insignificant system calls were removed. The final model is the one with the largest number of significant system calls.

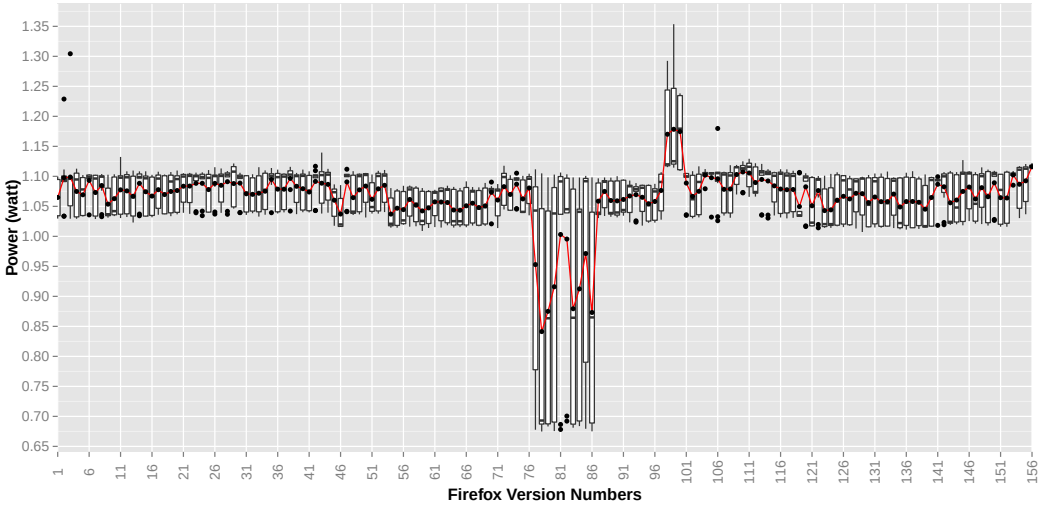

Figure 4: **Distributions of the mean watts consumed per version of `Firefox` application over 10 tests for 156 versions. The X axis represents the version numbers and the Y axis is average power use.**

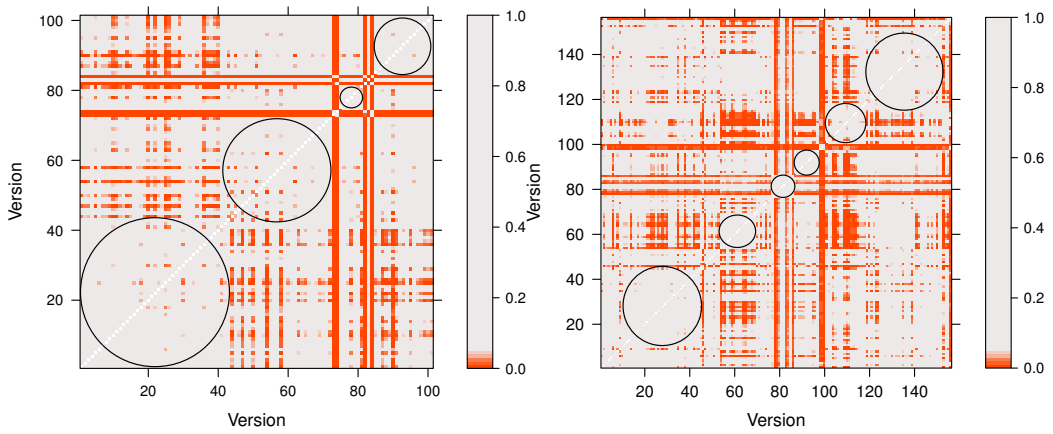

Figure 5: **Pairwise Student's *t*-test for power distribution of each version. Version pairs for `Calculator` are shown on the left, and version pairs for `Firefox` are shown on the right. The X axis and the Y axis represent the version numbers. Orange squares indicate pair of versions are significantly different (p-value $\leq$ 0.05) after correction for multiple hypotheses. Light Grey circles represent the clusters of versions having similar energy profile.**

System Call	Spearman's ρ	R^2	Coefficient b_1	$\bar{x}$	$b_1 \cdot \bar{x}$
`mmap2`	0.670	0.764	8.87×10^{-5}	511.30	0.0453
`open`	0.651	0.819	5.03×10^{-4}	115.73	0.0582
`close`	0.048	0.695	1.44×10^{-3}	32.72	0.0470
`epollwait`	0.031	0.684	1.06×10^{-5}	5113.56	0.0543
`mprotect`	0.613	0.756	5.83×10^{-6}	8438.49	0.0491
`recvfrom`	0.060	0.698	7.64×10^{-6}	7213.38	0.0550
`writev`	0.645	0.391	6.57×10^{-4}	29.03	0.0190
`cacheflush`	0.498	0.745	1.27×10^{-5}	3954.34	0.0500
`ioctl`	0.201	0.740	3.79×10^{-6}	14885.60	0.0560
`total calls`	0.373	0.740	5.64×10^{-7}	98155.94	0.0550

Table 1: Linear model summary for selected system calls for `Calculator` application

Calculator 257 models using 5 system calls were found. No model was found where all 6 or more system calls were still significant. In order to select the best model out of these models, mean accuracy under 10-fold cross validation was used. The dataset is quite small and the variability in the cross validation error is high. To counter this variability, 10-fold cross validations were run 100 times and their accuracies were averaged. The model using `recvfrom`, `munmap`, `cacheflush`, `gettid`, and `epoll_wait` had the highest average accuracy of 87.7%. The model with the lowest accuracy using only significant system calls had an accuracy of 75.6%. The top 3 most common system calls in all 257 models were `write`, `gettimeofday`, and `getpid`. Using the most accurate model, *it is possible to predict the direction of the change in energy consumption relative to the reference energy consumption with very high accuracy.*

Firefox 128 models containing 9 system calls were obtained. No model had 10 or more system calls that were statistically significant. Each of the 9 features is the change in the system call count of a version with respect to the mean call count of that system call across all versions. The highest average accuracy was 80.4% from the model with `recvmsg`, `lseek`, `read`, `gettimeofday`, `futex`, `epoll_wait`, `clock_gettime`, `cacheflush` and `access` as features. The lowest accuracy was 68.07%. The three most common system calls among the models were `ioctl`, `close` and `lseek`.

System Call	Description [21]
mmap2	map files or devices into memory (called to allocate memory and load libraries)
open	open a file descriptor
close	close a file descriptor
epollwait	Wait for events on the epoll file descriptor
mprotect	set protection on a region of memory
recvfrom	receive messages from a socket
writev	write data into multiple buffers
cacheflush	flush contents of instruction and/or data cache
ioctl	performs device-specific I/O operations

Table 2: Selected system calls with their descriptions from the Calculator application test case.

4.5 Rule of Thumb

Developers should not be required to use expensive and specialized power testing equipment in order to make predictions about the changes in energy usage caused by their changes to the application source code. One of the goals of this paper is to provide developers with a "rule of thumb" in order to make this prediction quickly. To provide this rule, the Student's *t*-test was used to examine the relationship between significant changes in system call usage and significant changes in power consumption.

In order to create a rule useful in real-world development processes, only consecutive versions were considered. 100 Student's *t*-tests were performed for Calculator and 155 Student's *t*-tests were performed for Firefox. Each test was done for a consecutive pair of versions to determine whether the version succeeding the change was significantly different from its predecessor. For each version, 10 power measurements were taken and compared with the 10 power measurements for the next version.

t-tests were also performed for each of the system calls, using the 10 call invocation counts to determine whether the usage of that system call changed significantly between any two consecutive versions. An α of 0.05 was chosen as a p-value threshold for establishing whether consecutive versions are significantly different or not. Since consecutive versions are tested, the control group is different for each t-test. Therefore, multiple test correction is not required.

The purpose of this procedure is to determine whether a significant change in system call use and a significant change in energy profile usually occur together. In order to establish the usefulness of the rule of thumb, precision, recall and specificity were calculated:

$$\begin{aligned} Precision &= \frac{SS}{SS+SN} \\ Recall &= \frac{SS}{SS+NS} \\ Specificity &= \frac{NN}{NN+SN} \end{aligned}$$

SS is the number of times the power difference as well as the system call count difference are significantly different; NS is the number of times the power is significantly different but the system call profile is not ; SN is the number of times the system call profile is significantly different but the power is not; and NN is the number of times that neither are significantly different.

All three statistics, precision, recall, and specificity, range from 0 to 1 with 0 being the worst and 1 being the best possible value. High precision indicates that positive results from our rule of thumb indicate a significant change in power consumption. High recall indicates that there were few significant power changes that the rule of thumb missed. High specificity indicates that our rule of thumb generates few false positives.

In other words, if a developer used our rule of thumb to decide when to perform power tests, with high precision, more of the time they spent doing power testing would yield significant results. With high recall, they would find more of the significant results that could be found. With high specificity, they would spend less time testing insignificant results. Table 5 summarizes these statistics for our rule of thumb on selected system calls on both applications.

Calculator The highest recall of 0.909 was obtained using the system call cacheflush while the lowest was obtained by using sendto which had a constant number of invocations, 182, in all versions. The highest precision of 1 was obtained using the system calls pread and stat64

System Call	Spearman's ρ	R^2	Coefficient b_1	$\bar{x}$	$b_1 \cdot \bar{x}$
_llseek	0.100	0.040	1.30×10^{-4}	208.45	0.029
brk	-0.170	0.001	-8.07×10^{-5}	72.175	-0.005
mmap2	0.360	0.257	3.50×10^{-4}	546.95	0.190
ioctl	0.407	0.534	1.77×10^{-4}	1413.16	0.250
epollwait	0.088	0.274	3.41×10^{-5}	6293.50	0.210
ftruncate	0.001	0.269	2.64×10^{-7}	10.23	0.000
fsync	0.044	0.126	1.00×10^{-3}	78.46	0.120
close	-0.030	0.016	3.10×10^{-6}	8117.55	0.020
fstat64	0.060	0.014	1.57×10^{-5}	1517.78	0.023
dup2	0.235	0.008	1.40×10^{-3}	11.43	0.002
write	0.199	0.072	5.63×10^{-6}	7313.99	0.041

Table 3: Linear model summary for selected system calls for Firefox application

System Call	Description [21]
_llseek	reposition read/write file offset
brk	change data segment size (used to allocate memory, for example by malloc)
mmap2	map files or devices into memory (used to load libraries and allocate memory, for example by malloc)
ftruncate	truncate a file to a specified length
ioctl	performs device-specific I/O operations
epollwait	Wait for events on the epoll file descriptor
fsync	synchronize a file's in-core state with storage device
close	close a file descriptor
fstat64	get file status
dup2	duplicate a file descriptor

Table 4: Selected system calls with their descriptions from the Firefox application test case.

while the lowest was obtained by using sendto. The highest specificity was obtained by using cacheflush while the lowest specificity was obtained by using sendto. For some system calls, changes in system call usage predicted changes in energy usage with high precision and recall while having high specificity as shown in Table 5.

Firefox The highest recall, 0.6, is obtained using the system calls lstat64, pipe, and utimes. The lowest recall of 0.1 was obtained using seven different system calls. The highest precision, 0.263, was obtained using ioctl while the lowest was obtained using stat64. The highest specificity of 0.96 was obtained using ioctl, while using cacheflush had the lowest at 0.93. Values for precision and recall are lower than those obtained for the Calculator application, however specificity is higher.

Sum of Calls in table 5 refers to sum total of counts of all system calls, and can be used in a situation where the developer has no information about what system call will be best to use.

Given our threshold of $\alpha = 0.05$, Calculator's power use changed significantly 11 out of 100 times, or 11% and Firefox's power use changed significantly 10 out of 155 times, or 6.4%. Thus, by randomly guessing version pairs we would get a precision of 0.11 for Calculator and 0.06 for Firefox. Because the number of commits that change energy consumption significantly is low and energy testing is expensive, we expect a much higher specificity than recall. The unbiased coin flip detects on an average of 50% of the significant change power versions, giving a recall of 0.5. Random guess gives equal number of false negatives and true negatives, giving a specificity of 0.5 also. Regardless of data, while randomly guessing, recall and specificity must sum to 1. An unbiased coin flip usually performs worse than our rule of thumb with a randomly selected system call. System calls that are constant between versions perform worse than a random guess. In practice, the random guess should be biased to predict lesser number of significant changes, thereby reducing recall and increasing specificity to limit testing cost. Even

Calculator application			
	Precision	Recall	Specificity
sendto	0	0	0.89
stat64	**1.00**	0.55	0.98
cacheflush	0.34	**0.91**	**0.98**
Sum of calls	0.35	0.72	0.96
Coin flip	0.11	0.50	0.50
Firefox application			
	Precision	Recall	Specificity
fcntl64	0.04	0.10	0.94
ioctl	**0.26**	0.50	**0.96**
lstat64	0.08	**0.60**	0.95
Sum of calls	0.18	0.60	0.97
Coin flip	0.06	0.50	0.50

Table 5: Rule of Thumb — Precision, recall and specificity for the rule of thumb, using best and worst system calls for the Calculator and Firefox applications.

sum of system calls outperforms the unbiased coin flip on all three statistics. Even if a biased coin flip were to be used, the rule of thumb using the sum total would perform better.

These observations can be used as a rule of the thumb by developers: If the system call profile changes significantly from the previous version, it is probable that the application's power usage has changed as well.

5 Discussion

The results in the previous sections show that system call counts are somewhat stable but still have variance. This observation implies that repeated measurements are required in order to address the high variability of some of the system calls counts. The linear models obtained show that many system calls correlate with energy consumption. This was demonstrated by high R^2 values obtained with the Calculator application models. However, in the case of Firefox, the linear models do not perform as well, having both lower correlations and lower R^2 values.

The logistic regression models successfully determined if a change in the system call profile would lead to a significant change in power consumption with accuracies between 75.6% and 87.7% for the Calculator application and accuracies of 68.07% to 80.04% for Firefox. Our logistic model is able to predict whether power usage will change significantly with high accuracy using the changes in system call usage.

While it might be the case that an application is using a lot of power doing CPU-only computations while not making system calls, this is rare because usually CPU-heavy computations involve memory management system calls such as sbrk. Even in the case that they do not, this type of application behavior will still be evident in the system call profile. Waiting for system call completion is usually the only way that an application can allow the CPU to idle. Thus, we expect system calls like epollwait, which is specifically used to idle the application, to have a positive correlation with power use. This is because in order for the application to sleep many times, it must wake up and perform computations many times. And in fact, this is what was observed in Table 3.

I/O system calls can also be an indicator that an app is using power-hungry peripherals. For example, an app might use write to send data to a remote computer over the network, activating the phone's WiFi transmitter, which requires power. Thus, a network application like Firefox should show a positive correlation between write calls and power consumption, which is what was observed.

Some system calls have a negative correlation with power use. This correlation is most likely caused by an application using alternate methods which use less power to achieve the same result. Consider the case of Firefox, which, like all browsers, has a cache. If Firefox can retrieve data from its cache it doesn't need to use WiFi as much, saving power.

The rule of thumb, "energy profile usually changes significantly when the system call profile changes significantly," is supported by both datasets, however it achieves higher precision, recall and specificity on Calculator than on Firefox. The rule of thumb outperforms the random guess, providing developers an insight to review their last change with the application's energy consumption perspective, resulting in large time savings when instrumentation is not available or not feasible.

Thus, developers who keep track of system call profiles for their application can make an informed decision on when more expensive power testing may be useful. Profiling system calls is less resource intensive than setting up special hardware test bed and using power instrumentation to profile power consumption. The rule of thumb requires only that developers track system call profiles, which is very simple with the help of `strace`, a free software instrument.

6 Threats to Validity

Internal validity is constrained by our choice of applications and their versions. Also, variation in power from version to version is not very high in the `Calculator` versions tested. Additionally, the larger linear models can suffer from multicolinearity.

The external validity is constrained by the test construction and application choice. It is possible that our test has limited coverage of the applications' features. However, time constraints limit the number of features that can be tested 1010 or 1560 times per application. High coverage tests do not guarantee a realistic test or effective usage, however, because high-coverage tests will often exercise features that are rarely used in practice.

7 Conclusion

This study investigated the relationship between system call invocations and energy consumption across multiple versions of two Android applications, `Calculator` and `Firefox`. System calls were found to suffer from limited variability. By relying on averages, linear models of the relationship between system calls and energy consumption became possible. Most system calls are mildly correlated to energy consumption.

This work proposes a change model using logistic regressions to predict if a new version has significantly different energy consumption compared to an older version based on the difference in system call invocations. Our model achieves a maximum accuracy of 87.7% for the `Calculator` and 80.04% for `Firefox`.

The hypothesis that a significant change in an application's system call profile predicts a change in the application's energy consumption profile, is supported by the results obtained. This study demonstrates a relationship between energy consumption and system call profiles, providing a promising research direction, and a practical method for developers to estimate when running expensive energy measurement tests is worthwhile by making inexpensive measurements of their change's impact on system calls usage.

This work demonstrates that system call profiles can be used to model changes in energy consumption profiles for Android applications. These observations are easy to leverage by developers. Developers concerned about their software's energy consumption can estimate the impact of a revision by comparing system call counts between revisions using simple tools like `strace` combined with student's t-test or a logistic regression built with the power profiles of previous versions.

A simple tool could perform system call tracing and the Student's t-test. If power profiles from previous versions of the same software were available, it could also use a model produced by logistic regressions. Such a tool would be useful to determine if a change might have introduced a bug which increases power consumption, as occurred during the development of `Firefox`. Such bugs could be introduced during refactoring or the addition of new features, and may not be detected by either user or automated testing.

References

[1] A. Hindle, A. Wilson, K. Rasmussen, J. Barlow, J. Campbell, and S. Romansky, "GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework," in *Mining Software Repositories (MSR), 2014 11th IEEE Working Conference on*. ACM, 2014.

[2] 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 *CODES/ISSS '10*. New York, NY, USA: ACM, 2010, pp. 105–114.

[3] M. Dong and L. Zhong, "Self-Constructive, High-Rate Energy Modeling for Battery-Powered Mo-

bile Systems," in *MobiSys '11*. New York, NY, USA: ACM, 2011, pp. 335–348.

[4] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-Grained Power Modeling for Smartphones using System Call Tracing," in *Proceedings of the sixth conference on Computer systems*, ser. EuroSys '11. New York, NY, USA: ACM, 2011, pp. 153–168.

[5] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating Mobile Application Energy Consumption using Program Analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, 2013, pp. 92–101.

[6] A. Hindle, "Green Mining: A Methodology of Relating Software Change to Power Consumption," in *MSR*, 2012, pp. 78–87.

[7] R. W. Stevens and S. A. Rago, *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[8] J. Flinn and M. Satyanarayanan, "PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications," in *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications*, ser. WMCSA '99, 1999.

[9] 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*, ser. HPCA '02, 2002.

[10] A. Shye, B. Scholbrock, and G. Memik, "Into the Wild: Studying Real User Activity Patterns to Guide Power Optimizations for Mobile Architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 168–178.

[11] A. Carroll and G. Heiser, "An Analysis of Power Consumption in a Smartphone," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, ser. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 21–21.

[12] R. Mittal, A. Kansal, and R. Chandra, "Empowering Eevelopers to Estimate App Energy Consumption," in *Proceedings of the 18th annual international conference on Mobile computing and networking*, ser. Mobicom '12. New York, NY, USA: ACM, 2012, pp. 317–328.

[13] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy Consumption in Mobile Phones: A Measurement Study and Implications for Network Applications," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, ser. IMC '09. New York, NY, USA: ACM, 2009, pp. 280–293.

[14] C. Seo, S. Malek, and N. Medvidovic, "An Energy Consumption Framework for Distributed Java-Based Systems," in *ASE '07*, 2007, pp. 421–424.

[15] ——, "Component-level energy consumption estimation for distributed java-based software systems," in *Component-Based Software Engineering*. Springer, 2008, pp. 97–113.

[16] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating Android Applications' CPU Energy Usage via Bytecode Profiling," in *First International Workshop on Green and Sustainable Software (GREENS), in conjunction with ICSE 2012*, June 2012.

[17] D. Li, S. Hao, W. G. Halfond, and R. Govindan, "Calculating source line level energy information for android applications," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 2013, pp. 78–89.

[18] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the Energy Spent inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*, ser. EuroSys '12. New York, NY, USA: ACM, 2012, pp. 29–42.

[19] A. E. Hassan, "The Road Ahead for Mining Software Repositories," in *Proceedings of the Future of Software Maintenance (FoSM) at the 24th IEEE International Conference on Software Maintenance*, 2008, pp. 48–57.

[20] A. Gupta, T. Zimmermann, C. Bird, N. Naggapan, T. Bhat, and S. Emran, "Detecting Energy Patterns in Software Development ," Microsoft Research, Tech. Rep. MSR-TR-2011-106, 2011.

[21] The Linux man-pages project, "Linux Man Pages Online," http://man7.org/linux/man-pages/, 2013.