Definition, Implementation and Validation of Energy Code Smells: an Exploratory Study on an Embedded System

Antonio Vetro', Luca Ardito, Giuseppe Procaccianti, Maurizio Morisio
Dipartimento di Automatica ed Informatica
Politecnico di Torino
Torino, Italy
E-mail: name.surname@polito.it

***Abstract*—Optimizing software in terms of energy efficiency is one of the challenges that both research and industry will have to face in the next few years. We consider energy efficiency as a software product quality characteristic, to be improved through the refactoring of appropriate code pattern: the aim of this work is identifying those code patterns, hereby defined as Energy Code Smells, that might increase the impact of software over power consumption. For our purposes, we perform an experiment consisting in the execution of several code patterns on an embedded system. These code patterns are executed in two versions: the first one contains a code issue that could negatively impact power consumption, the other one is refactored removing the issue. We measure the power consumption of the embedded device during the execution of each code pattern. We also track the execution time to investigate whether Energy Code Smells are also Performance Smells. Our results show that some Energy Code Smells actually have an impact over power consumption in the magnitude order of micro Watts. Moreover, those Smells did not introduce a performance decrease.**

Keywords-Code Smells; Energy Code Smells; Green Software; Software Metrics

I. Introduction

The issue of sustainability is starting to be addressed among the software engineering community. Although, it is still unclear how to design sustainable software. While for common quality characteristics (reliability, performance, security, etc.) processes and metrics have been proposed and widely investigated by the Software Engineering (SE) community, as regards sustainability the discussion is still in its initial phase.

Among the kaleidoscope of aspects related to software sustainability, one of the most visible is the energy (or, alternatively, power) consumption of software systems. Indeed, software does not consume energy directly, however it has a direct influence on the energy consumption of the hardware underneath. In fact, applications and operating systems indicate how the information is processed and, consequently, drive the hardware behaviour: previous work [1] suggested that software can increase the total power consumption of a computer system up to 10%. This and other initial findings [2] open investigation spaces on the optimization of energy and power consumption of IT devices acting on the software instead of the hardware. Moreover, nowadays the same software runs on multiple devices, thus it might be more productive and feasible for software houses to green the single software rather then relying on the greening of all the hardware implementations underneath (that could require competences commonly not owned by software houses). Optimizing a software product in terms of energy efficiency has also some issues. The absence of a standard procedure, or a benchmark, to compare systems is the most prominent one. This is because software is intangible and it is deployed on devices with their own specifications and features. This makes really difficult to standardize a transparent, platform-independent measuring system for every software system.

Another consideration must be done regarding software architectures. During the last years, software engineers always tried to increase the number of software layers - that is, for improving interoperability, abstraction, decoupling, etc. However, the steep increase of software layers directed the optimization efforts only on each layer ("horizontal" optimization) and not across them ("vertical" optimization). Since energy efficiency directly relates with hardware technologies, a more intense communication flow between hardware and software is needed to achieve significant optimizations. In this sense, embedded systems make a perfect case study, because their architecture is simplified by design, and also because power consumption issues acquire a peculiar importance, for operational reasons (most embedded systems are battery-powered). For this reason our work uses an embedded system as the testbed to validate a new approach for the design and implementation of sustainable software. We investigate, and here we also introduce the goal and main contribution of this study, how software can be optimized by identifying code patterns that use in a sub-optimal way the hardware resources. These code patterns ought to be refactored in order to improve the energy efficiency of the software at run time. We define and name the code patterns Energy Code Smells, inspired by the well-known book of Fowler and Beck [3].

This study empirically validates the impact of Energy Code Smells over power consumption. We provide background in Section II, then in Section III we describe the used approach for the validation of the concept of Energy Code Smell. In Section IV we describe the experimental setup of our analysis: results (V), discussion (VI) and threats to the validity (VII) follow. Finally we mention the related works (VIII) and we expose our conclusions and future research (IX).

II. Energy Code Smells: background and definition

The term "code smells" was coined by Fowler and Beck [3] referring to poor implementation choices that make the software difficult to maintain. These bad implementation practices can be characterized as patterns in source code. For instance, the smell "*Long Method*" refers to a method that has grown too large: typically, the longer is the method the more difficult is to maintain it. One or more refactoring actions are associated to code smells: for example all you have to do to refactor a Long Method is to extract parts of the method that seem to go nicely together and make a new method. As a result the original method is shorter and easier to maintain. Refactoring code smells might have an effect not onaly on maintainability but also on other properties of the software, such as portability, testability or, as in the case of this work, the energy efficiency. As a consequence, we take inspiration by the original work of Fowler and Beck and we introduce the concept of smells into the Green IT community, introducing the Energy Smells:

A Energy Smell is an implementation choice that makes the software execution less energy efficient.

Since software has different levels of abstractions and organizations, Energy Smells can be located at code, design or architectural level. Therefore, Energy Code Smells are implementation choices *at source code level* (code patterns) that make a sub-optimal usage of the hardware resources underneath. As a consequence, they provoke a higher energy (or alternatively, power) consumption.

III. Validation of Energy Code Smells

The aim of our research is to identify Energy Code Smells. In addition to that, we are also interested in understanding whether the Energy Code Smells also degrade the performances of the application in terms of execution time. We set up two research questions for our investigation:

RQ1. Which code patterns have an effect on power consumption (i.e. which code patterns are Energy Code Smells)?

RQ2. Code smells that have an effect on execution time do also have effect on energy consumption (i.e. are Energy Code Smells also Performance Smells) ?

The epistemological approach adopted for this research is the empirical one. We set up an experiment observing two dependent variables: power consumption (W) for RQ1 and execution time (ms) for RQ2. The two dependent variables are measured on the execution of C++ functions running on an embedded device. The choice of the embedded device has several advantages, the main two being:

- it has no operating system and thus confounding factors in the experiment are minimized;
- it runs on a battery and it really needs energy efficient code.

In other terms, refactoring Energy Code Smells in such an environment might lengthen the life of the battery.

The potential Energy Code Smells selected for the experiment are code patterns used by two popular static analysis tools. For each code pattern selected for the experiment, we set up a C++ function with two implementations, one that violates the code pattern (thus contains a potential Energy Code Smell) and the refactored one without the violation. Therefore the treatment is the refactoring of the smell and it is possible to observe an effect on the two variables by comparing the measurements on the two versions of the code.

A. Potential Energy Code Smells selection

As introduced above, the software that runs on the selected device is C++ code. In order to identify Energy Code Smells on C++ code we look at already existing code patterns. In particular, we examined patterns implemented by Automatic Static Analysis (ASA) tools. ASA tools examine source and compiled code and check it against good programming practices and possible bug patterns. The advantage of using ASA tools is the speed of the verification and the applicability before testing or production phase.

The two tools selected for this study are Cpp-Check and Findbugs. CppCheck is a well-known static analysis tool for C/C++ which contains many patterns regarding a variety of desired software properties: safety, portability, performance, etc . An example of C/C++ pattern on portability is "*64 bits portability*", i.e. assign address to int or long. An example of checked pattern on performance is instead "*Address not taken*" of the category "Memory leaks", which detects when the address to allocated memory is not taken. In order to identify which patterns can be considered relevant for energy efficiency, two of the authors carefully read all patterns and selected independently which ones could cause a higher power consumption of the Waspmote. All conflicts (a pattern selected by only one expert) were resolved in a reconciliation meeting, where patterns were discussed and a final decision taken. In addition to the Cpp-Check patterns, we also reviewed the patterns of another static analysis tool, Findbugs. It is similar to Cpp-Check, but it analyzes Java code. The same two authors reviewed all FindBugs patterns and decided firstly if they can be applied to C++ code, then whether they might be related to energy efficiency.

The selection process ended up with the patterns shown in TABLE I.

TABLE I
POTENTIAL ENERGY CODE SMELLS SELECTED FOR VALIDATION.

Pattern Name	Pattern Description	Tool
Parameter By Value	Passing a parameter by value to a function	CppCheck
Self Assignment	Assignment of a variable to itself. (e.g., x=x).	CppCheck
Mutual Exclusion OR	OR operator between two mutually exclusive conditions (thus always evaluating to true).	CppCheck
Switch Redundant Assignment	Redundant assignment in a switch statement: for example, assigning a value to a variable in a case block without a following break instruction, then re-assigning another value to the same variable in the subsequent case block.	CppCheck
Dead Local Store	A statement assigning a value to a local variable, which is not read or used in any subsequent instruction.	FindBugs
Dead Local Store Return	A return statement assigning a value to a local variable, which is not read or used in any subsequent instruction. (i.e. return(x=1);)	FindBugs
Repeated Conditionals	A condition evaluated twice (e.g., x==0 —— x==0).	FindBugs
Non Short Circuit	Code using non-short-circuit logic boolean operators (e.g., & or ‖) rather than short-circuit logic ones (&& or ‖‖). Non-short-circuit logic causes both sides of the expression to be evaluated even when the result can be inferred from knowing the left-hand side.	FindBugs
Useless Control Flow	Control flow constructs which do not modify the flow of the program, regardless of whether or not the branch is taken (e.g., an if statement with an empty body).	FindBugs

Subsequently, we wrote for each of the patterns a pair of C++ functions, one containing a potential smell and another one refactored without that smell. For example, the "Non-Short Circuit Logic" pattern, shown in Listing 1 has the following two functions:

```
void NonShortCircuit_With(){
        int count = 0;
        int total = 345;
        if ( count > 0 & total / count > 80 )
                count=0;
}

void NonShortCircuit_Without() {
    int count = 0;
        int total = 345;
        if ( count > 0 && total / count > 80 )
                count=0;
}
```

Listing 1. Non Short Circuit Code Pattern

The function NonShortCircuit With() is the one with the potential smell "*Non-short circuit logic*". The smell is in the line $if(count > 0 \ \& \ total/count > 80)$ because the AND operator is single & and so both predicates in the expressions will be evaluated at run-time. In the function, NonShortCircuit Without() the code is refactored replacing & with &&. All functions are available online [4] for the sake of replication.

IV. EXPERIMENT

A. Context: the WASP

The device used for the experiment is the Waspmote V1.1 (Libelium Comunicaciones Distribuidas S.L. Esso). The hardware architecture is based on a ATmega 1281 microcontroller with a CPU frequency of 8 MHz and 8KB of SRAM. It has no operating system: programs are directly loaded on a FLASH memory of 128 K. This architecture well suites our experiment because no other threads run in parallel with the chosen program, thus eliminating any software noise for the energy measurement. The device is basically a motherboard with connectors to plug in other elements such as sensors, wireless modules (ZigBee, XBee, Bluetooth), GSM/GPRS modules and a GPS (Global Positioning System) module. For this reason it is used in different fields, such as Smart Metering, Building Automation, Agriculture etc. It runs on a lithium battery (3.7V and 1150mAh), so the energy consumption of software has a key role here. To compile and load the C++ programs it is sufficient to use the IDE provided by the manufacturer and connect it to a computer via USB cable.

B. Experiment setup

The objective of the experiment is to measure power consumption and execution time on each function pair, in order to evaluate if the potential smell affects the two dependent variables. We divide the experiment in two parts: one for measuring power consumption, and another one for the execution time.

Measuring power consumption and execution time for a single function is a challenging task because usually execution is too fast to get reliable data. We control this threat repeating each function 1 million of times, that makes one sample. We collect 50 sample in order to reach statistical significance. Each function pair is loaded on the Waspmote and evaluated two times: the first one for the execution time, the latter one for the power consumption.

No specific instrumentation was needed to obtain the execution time, because the Waspmote embeds a Real-Time Clock (RTC) with a millisecond accuracy. We measure the execution time of every loop (i.e. 50 measurements).

On the other side, analyzing power consumption is more complicated. The only way to obtain a precise measure of the power consumption is using a power meter. The RTC is powered by an auxiliary battery, which makes it completely independent from the main power supply. Therefore it is possible to power the Waspmote with a constant voltage

(V_G = 3.7 V) by means of a generator and use a shunt resistor to measure the current intensity. An analog to digital converter (ADC) connected to the PC reads the voltage drop across a resistor R of 1 Ω. The current flowing in the circuit can be computed by measuring the voltage drop on the resistor ($I = V_{ADC}/R$). The instant power consumption value can be computed as:

$$P = V_L \cdot I = (V_G - V_{ADC})\frac{V_{ADC}}{R} = \frac{V_G V_{ADC} - V_{ADC}^2}{R} \quad (1)$$

Fig. 1 represents the circuit described.

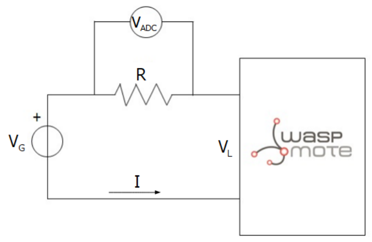

Figure 1. Circuit built to measure the power consumption.

The device used to measure the power consumption has a frequency of 49KHz, i.e. it gets 49000 measurements each second. In order to precisely measure the power consumption relative to the execution of the function pairs, we inserted a sleep interval at the beginning of the data acquisition to exclude the peak of device power on, and we filtered out, through a threshold, all the measurements corresponding to the idle consumption between the iterations of the function execution. As shown in Fig. 2, the threshold filters out the transient and includes only the peaks corresponding to the actual execution of the function.

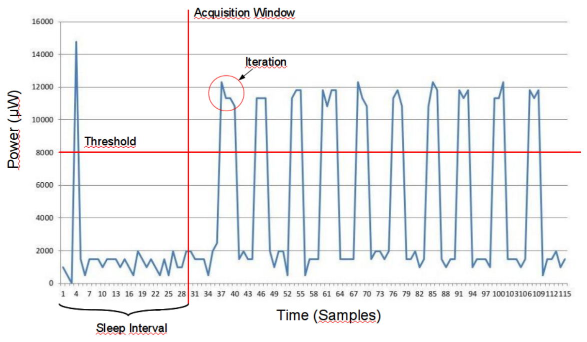

Figure 2. Sampling current intensity: an example.

C. Analysis methodology

For each research question we derived a pair of null and alternative hypotheses to test.

RQ1:

$$H1_0 \quad : P_{with} \leq P_{without}$$

TABLE II
RESULTS OF POWER CONSUMPTION.

Smell name	Mean with smell (μW)	Mean w/o smell (μW)	Diff. Means (μW)	P-val	Impact%
Dead Local Store Return	41241	41278	-37	1	-0.09
Dead Local Store	40249	40205	44	$\leq$ **0.01**	0.11
Mutual Exclusion OR	40758	40772	-14	1	-0.03
Non Short Circuit	41113	41043	70	$\leq$ **0.01**	0.17
Parameter By Value	40967	40723	244	**0**	0.60
Repeated Conditionals	41155	41126	29	$\leq$ **0.01**	0.07
Self Assignment	40952	40879	73	$\leq$ **0.01**	0.18
Switch Redundant Assignment	40724	40756	-32	1	-0.08
Useless Control Flow	41051	41142	-91	1	-0.22

$$H1_a \quad : P_{with} > P_{without}$$

where P is the power consumption of the function, with and without the potential smell. If the refactored version of the function consumes less than the function with the smell, the null hypothesis is rejected in favor of the alternative one. As a consequence we consider the pattern a Energy Code Smell. The hypothesis is tested with the Mann-Whitney test, given $\alpha = 0.05$.

RQ2:

$$H2_0 \quad : T_{with} \leq T_{without}$$
$$H2_a \quad : T_{with} > T_{without}$$

where T is the execution time of the two functions. If the smell has a negative impact on performance, the refactored function will be faster and the null hypothesis is rejected. In that case, we consider the pattern a Performance Smell. In order to answer RQ2, we compare which Energy Code Smells are also Performance Smells. We also use Mann-Whitney and $\alpha = 0.05$ to test the hypotheses.

At the end of the experiment each function has 50 measurements of execution time and about 25 millions of power measures. Then, after filtering out values below the idle threshold (8mW), we obtained about 8 millions values for power measurement, on which we ran the analysis.

V. RESULTS

We report results on the power consumption and execution time respectively in TABLE II and III. The two tables report the name of the smell, the means and their difference for both the dependent variables, the p-value of the Mann Whitney test [5] and the difference in percentage of the power consumption (or execution time) between the execution of the code with the smell and the execution with the refactored code.

TABLE III
RESULTS OF EXECUTION TIME.

Smell name	**Mean with smell** (ms)	**Mean w/o smell** (ms)	**Diff. Means** (ms)	**P-val**	**Impact%**
Dead Local Store Return	3288.76	3288.74	0.02	0.41	6.08e-04
Dead Local Store	17707.34	17707.38	-0.04	0.66	-2.26e-04
Mutual Exclusion OR	3540.76	3540.60	0.16	**0.04**	4.52e-03
Non Short Circuit	3288.74	3288.80	-0.06	0.76	-1.82e-03
Parameter By Value	3288.76	3288.74	0.02	0.41	6.08e-04
Repeated Conditionals	3288.80	3288.74	0.06	0.24	1.82e-03
Self Assignment	3288.66	3288.78	-0.12	0.90	-3.64e-03
Switch Redundant Assignment	3540.58	3540.62	-0.04	0.65	-1.13e-03
Useless Control Flow	3288.80	3288.74	0.06	0.24	1.82e-03

We observe from TABLE II that all power consumptions ranged from 40mW to about 42mW. Five code patterns over nine have a p-value ¡ 0.05 (in bold) and therefore the null hypothesis is rejected for them. The code patterns are:

- DeadLocalStore
- NonShortCircuit
- ParameterByValue
- RepeatedConditionals
- SelfAssignment

Overall the saved power consumption is less than 1%.

The answer to RQ1 is: *five code patterns (DeadLocalStore, NonShortCircuit, ParameterByValue, RepeatedConditionals, SelfAssignment) are Energy Code Smells, and their impact is in the order of* μW.

Focusing on performance, from TABLE III becomes evident that there is no difference in execution time. The null hypothesis is rejected only for MutualExclusionOr, however the magnitude order is μ seconds. We also notice that DeadLocalStores are about 5 times slower.

Thus, our answer to RQ2 is: *Energy Code Smells are not Performance Smells*.

VI. DISCUSSION

We identified five smells which provoked a higher power consumption of the Waspmote in the use cases prepared for the experimentation. However, we observe that the saved power is less than 1 %. A first motivation resides in the implementation choices: the function pairs executed only differ in a single instruction, and the operations are done with primitive types (e.g., integer). The motivation of such implementation was the exclusion of any possible confounding factor in the analysis, but the drawback of such a choice is a very small achievement in energy efficiency improvements. Let us take dead stores as example: the smell *DeadLocalStore* is implemented with an integer (we save a value on a variable and immediately overwrites it with another integer). Using a struct with several members is totally different and might lead to a higher impact, because the resulting compiled code requires the CPU to produce more instructions and interact more intensively with the memory. If increasing the complexity of the data structure will result in still negligible power consumption saving, the next step is to increase the logical complexity of the function, i.e. comparing complete algorithms that are functionally equivalent but differ in the implementation. A further step is to move the focus towards the comparison of functionally equivalent design choices. Understanding the impact of Energy Code Smells over real power consumption could also contribute to build more precise models of the power consumption of software. As a matter of fact, it may be possible to categorize software instructions beforehand in terms of energy efficiency, then subsequently use this information in order to predict the resulting energy efficiency of a complete software product.

Yet another research direction that is suggested by this first leap is: can the impact of Energy Code Smells be higher in code that drives an hardware resource with higher energy needs? For instance the impact on the code that handles the GPS transmitter is expected to be very different from the one used in this experiment, where the small functions use only CPU and RAM, besides in a not intensive way. The same investigation approaches can be applied to the domain of execution time. As can be noticed from the results, all the execution times are equal, exception given for the *DeadLocalStore* function pairs. We have observed that Energy Code Smells do not degrade the performances, but we cannot generalize the findings for more complex code structures and usage scenarios, with different hardware resources involved (e.g, sensors).

VII. THREATS TO VALIDITY

In this section, we expose the threats to validity that might affect our study.

As regards construct validity, our main threat regards instrumentation. We carefully evaluated the precision of our measures, comparing them with the specifications from Waspmote manufacturers. During our experimentation, the difference between actual and expected values was negligible and inside the specified ranges. As far as conclusion threats are concerned, in order to increase the statistical reliability of the results, we collected a relevant amount of values (e.g., every function is looped 1 million times for power consumption measurement resulting in 25 millions of samples). Internal validity is represented by confounding factors such as other processes running during execution. However, the Waspmote does not have an operating system and the only thread in execution during the tests is the code loaded. As regards external validity, we do not aim at generalizing our results to a family of embedded devices.

This study aims at assessing the existence of the Energy Code Smells in a single context: other empirical validations are necessary for other environments or devices.

VIII. Related Work

We did not find in the literature similar approaches for energy efficiency optmization. However, we found techniques that rely on algorithmic and data optimizations. The algorithmic optimization has a high potential, but it is also a hard and time-consuming task, with no guaranteed results. Data optimization is based upon the efficient use of the system architecture. For example, as regards embedded systems, often software libraries are used for emulating floating-point hardware components. Those libraries do not take into account the architecture of a specific system, thus their usage often leads to a high power consumption and low performance. Šimunić et al. [6] show that by removing those libraries and optimizing the source code, it is possible to significantly reduce power consumption (up to 77%).

In terms of benchmarks, SPECpower [7] is an initiative to extend existing SPEC benchmarks to power and energy measurement. SPECpower ssj2008 reports the energy efficiency in terms of overall $ssj\_ops/watt$. This metric represents the sum of the performance measured at each target load level (in ssj_ops) divided by the sum of the average power (in watts) at each target load including active idle.

In battery-powered systems, it is not enough to analyze algorithms based only on time and space complexity. Several research proposed energy aware algorithms for specific functionalities, such as supporting randomness [8] or focusing on cryptographic [9].

Previous work by Bunse et al. [10] addresses the relationship between energy and performance optimizations, which is one of the research questions of the present work. Authors analyzed different implementations of several sorting algorithms, showing that implementations optimized for energy performed differently with respect to those optimized for performance. This findings holds in our work, since we found that Energy Code Smells are not Performance Smells.

IX. Conclusions

This is an exploratory study: we defined for the first time the concept of Energy Code Smells and we performed a first validation to understand not only the impact, but also the boundaries of the concept. We identified some Energy Code Smells starting from code patterns implemented by two common Automatic Static Analysis tools - namely, CppCheck and FindBugs. We performed an experiment, on an embedded system, in order to assess the energetic impact of those code patterns and determine whether Energy Code Smells are also performance smells. Our experimental results showed that some of the code patterns actually have an impact over power consumption. This impact, however, is in the magnitude order of μW. Our future research works will be devoted to analyzing more complex data structures and using hardware resources which could increase this impact with respect to the overall power consumption. As regards time analysis, only one pattern had an actual impact over execution time (a few μ seconds), and it is not identified as a Energy Code Smell. Thus, we conclude that Energy Code Smells are not Performance Smells. Results suggest that the target and applicability of Energy Code Smells should be refined with further investigations. The lessons learned in this exploratory study let us identify several research threads that the research community might address, such as the identification of Energy Code Smells that are higher-level constructs with more complex data structures, the identification of Green Design Smells and the use of more complex systems as test beds. Finally, the experimental results that will be collected might be also used to build more precise models of the power consumption of software.

References

[1] G. Procaccianti, A. Vetro', L. Ardito, and M. Morisio, "Profiling power consumption on desktop computer systems," in Information and Communication on Technology for the Fight against Global Warming, ser. Lecture Notes in Computer Science, D. Kranzlmüller and A. Toja, Eds. Springer Berlin / Heidelberg, 2011, vol. 6868, pp. 110–123.

[2] A. Vetro', L. Ardito, M. Morisio, and G. Procaccianti, "Monitoring it power consumption in a research center: Seven facts," in Proceedings of The First International Conference on Smart Grids, Green Communications and IT Energy-aware Technologies, ser. ENERGY 2011, 2011, pp. 64–69.

[3] M. Fowler and K. Beck, Refactoring: improving the design of existing code. Addison-Wesley Professional, 1999.

[4] A. Vetro', L. Ardito, G. Procaccianti, and M. Morisio, "Energy code smells," January 2013, retrieved January 2013. [Online]. Available: http://softeng.polito.it/greensmells/

[5] L. Sachs, Applied Statistics–A Handbook of Techniques, S. S. in Statistics, Ed. Springer-Verlag, 1984.

[6] T. Šimunić, L. Benini, G. De Micheli, and M. Hans, "Source code optimization and profiling of energy consumption in embedded systems," in Proceedings of the 13th international symposium on System synthesis. IEEE Computer Society, 2000, pp. 193–198.

[7] K.-D. Lange, "Identifying shades of green: The specpower benchmarks," Computer, vol. 42, no. 3, pp. 95–97, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.84

[8] R. Jain, D. Molnar, and Z. Ramzan, "Towards understanding algorithmic factors affecting energy consumption: switching complexity, randomness, and preliminary experiments," in Proceedings of the 2005 joint workshop on Foundations of mobile computing. ACM, 2005, pp. 70–79.

[9] N. Potlapally, S. Ravi, A. Raghunathan, and N. Jha, "A study of the energy consumption characteristics of cryptographic algorithms and security protocols," Mobile Computing, IEEE Transactions on, vol. 5, no. 2, pp. 128–143, 2006.

[10] C. Bunse, H. Hopfner, E. Mansour, and S. Roychoudhury, "Exploring the energy consumption of data sorting algorithms in embedded and mobile environments," in Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on. IEEE, 2009, pp. 600–607.