Can We Spot Energy Regressions using Developers Tests?

Benjamin Danglot, Jean-Rémy Falleri, Romain Rouvoy

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

Can We Spot Energy Regressions using Developers Tests?

Benjamin Danglot · Jean-Rémy Falleri · Romain Rouvoy

Received: date / Accepted: date

Abstract Background
Software Energy Consumption (SEC) is gaining more and more attention. In this paper, we tackle the problem of warning developers about the increase of SEC of their programs during *Continuous Integration* (CI).
Objective
In this study, we investigate if the CI can leverage developers' tests to perform *energy regression testing*. Energy regression is similar to performance regression but focuses on the energy consumption of the program instead of standard performance indicators, like execution time or memory consumption.
Method
We perform an exploratory study of the usage of developers' tests for energy regression testing. We first investigate if developers' tests can be used to obtain stable SEC indicators. Then, we evaluate if comparing the SEC of developers' tests between two versions can pinpoint energy regressions introduced by automated program mutations. Finally, we manually evaluate several real commits pinpointed by our approach.
Impact
Our study will pave the way for automated SEC regression tools that can be readily deployed inside an existing CI infrastructure to raise awareness of SEC issues among practitioners.

Benjamin Danglot
Research & Development, Davidson consulting, Paris, France
E-mail: bdanglot@

Jean-Rémy Falleri
Univ. Bordeaux, Bordeaux INP, CNRS, UMR 5800 LaBRI, F-33400 Talence, France / IUF
E-mail: falleri@labri.fr

Romain Rouvoy
Univ. Lille, Inria, CNRS, UMR 9189 CRIStAL, F-59000 Lille, France / IUF
E-mail: romain.rouvoy@univ-lille.fr

Keywords Continuous Integration ; Energy Regression ; Sustainable Software

1 Introduction

Software Energy Consumption (SEC) is gaining more and more attention from both researchers and practitioners [1–4].[1] However, the awareness of SEC among developers is not yet widely spread and is just starting to be considered a key indicator of a project quality [5, 6].

In this article, we tackle the problem of warning developers about the increase of SEC of their programs. More specifically, this study takes place in the context of *Continuous Integration* (CI), widely adopted by the software industry. In particular, our overarching objective is to flag—or to classify as breaking—all CI commits that negatively impact the SEC of an application under development or maintenance.

Whenever a developer pushes changes as a commit, the CI is responsible for building and checking if the submitted changes are "correct". Traditionally, CI performs regression testing—*i.e.*, it verifies the absence of regression bug [7]. A regression bug is an unintended loss of features introduced by code changes not intended to alter this part of the program behavior. As code changes may have side effects that introduce an undesired behavior change (or bug), these regression bugs can be spotted by the tests triggered by the CI. The most straightforward regression detection technique is executing all the tests, no matter the submitted changes. If any test fails, this means that there is a regression, and the associated code changes are then labeled as breaking. Then, the developer who authored the code changes must fix the regression bug(s) and make all tests pass again.

In this study, which executes a pre-registered protocol [8], we investigate if the developers' tests [9] can be leveraged to perform an additional verification as part of the CI: *Energy Regression Testing* (ERT). Our idea is inspired by the work of Ding et al. [10], who showed that such tests could spot performance regressions in a program. Energy regression is similar to performance regression but focuses on the energy consumption of the program instead of its execution time or memory consumption. In other words, we aim to integrate into the CI an *energy regression oracle* that ensures that the applied code changes do not severely degrade the SEC. In particular, the CI would label a code change as breaking if it severely increases the energy consumption of the program.

Similarly to performance, energy consumption is a dynamic property: one must execute the program to collect SEC indicators. As we aim to detect the negative impact on energy consumption, we need to compare the energy consumption of two versions of a program: the version before and the one after applying a given commit. To do so, developers' tests can be considered a candidate workload—as they are already executed for each version—that can

[1] `https://greensoftware.foundation`

be used to monitor dynamic properties' variations, as long as the commit does not modify these tests.

In this article, we take a first step towards this vision by performing an exploratory study of the applicability of such an approach (which is detailed in Section 2). We first investigate if stable SEC indicators can be computed from developers' tests (see Section 4). Second, we investigate if comparing SEC indicators of two versions can pinpoint energy regressions introduced by controlled program mutations. Finally, we study the difference of SEC indicators along the commit history of several major *Open Source Software* (OSS) repositories and perform a manual evaluation of several real-world commits pinpointed by our approach (see Section 5).

We find out that it is possible to measure the energy consumption in a stable way using the number of cycles and that we can use the developers' tests to assess the variation of energy consumption on a ratio of commits ranging from 5% to 40% in the projects of our corpus. We also show that our approach can be tuned to adjust its sensibility. Furthermore, some manually analyzed breaking commits had signs of modifications that could induce an increase in SEC, while others seemed to be false positives, emphasizing the need for tunability. An important limitation of our approach is that we do not have any ground truth workload to put our results in perspective.

2 Overview & Research Questions

This section first provides an overview of our approach to performing energy regression testing (see Section 2.1). Then, it presents the two research questions investigated in our exploratory study (see Section 2.2).

2.1 *Energy Regression Testing* (ERT)

ERT refers to the capability of detecting energy consumption drifts in software systems that are induced by changes observed at the source code level. ERT differs from state-of-the-art regression testing by considering energy consumption as a non-functional concern that can cause a software test to break. *Test breakage* in the context of ERT, therefore, refers to a software test that is passing from a functional perspective, but reports on an increase of the energy consumption. This increase is observed by comparing the execution of tests on two consecutive versions of a *System Undert Test* (SUT). Figure 1 depicts an overview of the key phases involved in ERT.

ERT Selection. The first step of our approach aims at selecting the tests that should be executed to detect energy regressions. This step takes as input a project commit with its associated current snapshot (denoted by `HEAD` in Figure 1 and v_2 in the remainder) and parent snapshot (denoted by `HEAD~1` in Figure 1 and v_1 in the remainder). It computes a textual diff between the two

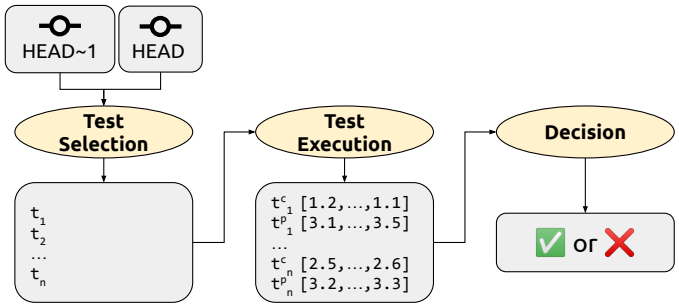

Fig. 1 Overview of key steps involved in ERT.

snapshots, resulting in a set of deleted lines in v_1 and a set of added lines in v_2. Then, it computes the code coverage of all tests (which test executes which line of the program) for both versions. We select the tests from v_1 that cover the lines deleted by the commit, and we select the tests from v_2 that cover the lines added by the commit. From this selection, we discard all the modified tests as the energy measurement of the underlying test might be affected by the modification, introducing noise in our measurements. We also discard the deleted (resp. added tests), as we cannot be sure that we can execute them on the current (resp. parent) version. We also discard the tests that fail on any of both versions. Then, we take the union of both test subsets (v_1 and v_2) as our candidate workload for energy measurement.

ERT Execution. The second step of our approach is to instrument the tests selected in the previous step to capture our measurements of interest. This is done by injecting probes into the selected tests. We inject two probes: one to start the monitoring at the beginning of the test and one to stop the monitoring at the end of the test. These start and stop probes will be automatically triggered when running the tests and will collect energy and performance metrics (these metrics will be detailed in 4). As explained in [11] energy measurement is inherently subject to variations. We execute the selected tests multiple times to compute a representative value of the actual energy consumption. We arbitrarily decided to execute 100×, as it limits the overall cost of the experiment while ensuring a confidence interval of 95% and a margin of error of 10%, according to Cochran's formula [12]. Therefore, for each test, we collect a sample of 100 measurements for v_1 and another sample of 100 measurements for v_2.

ERT Oracle. Finally, using the paired samples of 100 measurement computed in the previous step, we label the commit as *passing* (no energy regression detected, denoted using a green check) or *breaking* (an energy regression has been detected, denoted by a red cross). To make this decision, we adopt a two-step oracle. First, we use a *test filtering* process to filter out the tests exhibiting similar energy measurements between the two versions, which could introduce noise in our decision process. And finally, we apply a *decision ora-*

cle to classify the commit. The test filtering process and decision oracle are detailed in Section 5.

2.2 Research Questions

When it comes to reasoning on energy, the literature has shown that the measurement of energy consumption can be subject to large variations [11], which is mainly due to complex interactions of hardware and software features. In this context, the study of the energy impact of code changes should ensure that the measurements reported by the CI are stable and reproducible enough to avoid false positives. This is particularly challenging as short executions of developers' tests may affect the stability of measurements.

This observation, therefore, leads us to formulate the following two research questions:

- RQ_1: *Can the energy consumption of developers' tests be measured stably?*
  - H_1: Measuring the energy consumption of developers' tests delivers stable and reproducible measurements.
  - H_2: Measuring dynamic performance indicators, such as duration or executed instructions of developers' tests, can approximate the energy consumption and deliver stable and reproducible measurements.

  This research question assesses whether developers' tests can be leveraged to measure the energy consumption of a small snippet of code stably. It will deliver recommendations on which measurements can be considered by studying their stability across several measurements. H_1 focuses on considering energy consumption, usually reported in Joules, as a direct indicator to monitor a software system. Additionally, we establish H_2 as an alternative hypothesis to investigate if other—indirect—indicators can be used, if H_1 does not hold.
- RQ_2: *Can developers' tests be used to detect potential energy regression introduced by code changes?*
  - H_3: A potential energy regression introduced by code changes can be detected from the developers' tests by computing the *energy delta* of unmodified tests executed before and after the code changes.

3 Implementation & Dataset

3.1 Approach Implementation

We focus on Java projects managed using Maven. We restrict our study to this technological stack because we can leverage tools developed in previous works. While this choice reduces the time spent preparing our experiments, we believe it is possible to replicate our approach in other technological stacks, as long as the tools we need in our experiment (a code coverage framework and a source code manipulation framework, see below) are available. Our implementation

is available as an OSS repository on GitHub[2]. The data produced by our experiments are available on Zenodo[3]. In the remainder of this section, we describe the main components we use in our implementation.

Test Selection. To perform test selection, we enhanced the prototype diff-test-selection[4] that has been developed by Danglot et al. [13] to select the tests that execute the code changes and amplify them. We make the artifact more accurate by implementing our test selection algorithm, described in Section 2.1, within diff-test-selection. In our experimentation, we decided to select the tests that execute the code that has been changed to save computation time.

Energy Measurement. We use TLPC-sensor[5] to collect metrics about energy consumption and *Hardware Performance Counters* (HwPC) metrics. It uses *Running Average Power Limit* (RAPL) [14] to collect measurements on the energy consumption metrics of supported components, such as CPU and DRAM. It leverages the standard Linux library perf to collect HwPC metrics, such as the number of executed instructions, number of cycles, etc.

Test Instrumentation & Execution. Regarding test instrumentation—*i.e.*, above-mentioned probes to measure the energy consumption and performance metrics—we rely on the library of Java code analysis and transformation Spoon [15]. For test execution, we adopt the test-runner,[6] which delivers an API to run the tests in an isolated *Java Virtual Machine* (JVM), avoiding dependencies clashes.

3.2 Dataset

As explained in 3.1, our implementation requires Java projects managed using Maven with their commits. We rely on an existing dataset built by Danglot et al. [13], as they meet these criteria.

Our approach cannot be applied on commits with no Java code changes or on commits with an empty set of selected tests (because tests were modified along with the code or because no test covers the added or deleted lines). To that extent, for each project, we browse the commits' history backward, starting from the head of the main branch until we find 50 commits where we can successfully apply our approach.

Table 1 shows the list of OSS projects in our dataset. Most of our projects are Java libraries, except for `xwiki`. Their sizes range from small (10,000 LOC) to large (1,000,000 LOC). Interestingly, we note that the ratios of commits

[2] `https://github.com/davidson-consulting/diff-jjoules`

[3] `https://zenodo.org/record/6528917`

[4] `https://github.com/STAMP-project/dspot/tree/master/dspot-diff-test-selection`

[5] `https://github.com/davidson-consulting/tlpc-sensor`

[6] `https://github.com/STAMP-project/test-runner`

compatible with our approach range from 40% in the case of `jsoup` to 4% for `xwiki`. It indicates that many commits cannot be analyzed using our approach.

Table 1 Considered projects and period for selected commits.

project	type	LOC	start date	end date	latest commit	#total commits	#selected commits	%selected commits
jsoup	Library	24,586	17/01/10	20/10/21	`3bd4d79`	123	50	40.65%
mustache	Library	8,586	03/05/10	07/12/21	`2d814a7`	147	50	34.01%
commons -io	Library	42,343	25/01/02	28/10/21	`8827b4e`	194	50	25.77%
gson	Library	22,724	01/09/08	18/09/21	`aa5554e`	293	50	17.06%
commons -lang	Library	88,000	19/07/02	27/03/22	`4b9dfa2`	421	50	11.88%
xwiki	Application	100,114	13/10/06	05/04/22	`0b10941`	1,144	50	4.37%

Table 1 shows the main descriptive statistics of the benchmark dataset. The first column is the project's name, and the second is its type. The third column is the number of lines of java code computed with *cloc*. The fourth (resp. fifth) column is the date of the project's oldest (resp. latest) considered commit. The sixth column is the SHA of the latest considered commit. The seventh column is the number of commits we analyzed until we obtained 50 commits on which we could apply our approach. The eighth column is the number of commits we selected. The ninth column is the ratio of selected commits over the considered commits.

4 Can the energy consumption of developers' tests be measured stably?

In this section, we investigate our first research question.We first explain our experimental protocol, and we then describe our results.

4.1 Experimental Protocol

To answer RQ_1, we are interested in investigating the stability of measurements of interest for several executions of a given test. More precisely, we consider the energy consumption and HwPC of the commits selected in Section 3.2 as our measurements of interest. This results in the following variables:

- The *energy consumed* by the CPU during the execution of a test on a given version of the program under study reported in *Joules* (J);
- Several HwPC monitored along the execution of individual tests on a given version of the targeted program, most notably number of instructions, number of cycles. Both are positive integers.

The energy consumed by the execution of a test is the key metric of our study. Nevertheless, we believe that these two other performance metrics can offer relevant candidate metrics to approximate the energy consumption of a test.

We argue that these indicators might be used as an alternative whenever the energy consumption is too unstable. These variables will be used to study the energy variations of a SUT.

A limitation of our approach, leveraging developers' tests, is that the most common developer's tests available are unit tests, which are usually small and fast to execute. This strengthens the challenge of stable measurement of energy consumption.

To assess the stability of our measurements of interest, we compute their *Standard Deviation* (σ) and *Coefficient of Variation* (CV). These indicators deliver insights about the stability of the collected measurements and thus validate or invalidate the hypotheses H_1 and H_2. Having several measurements of energy consumption and performance metrics will provide us a way to compute statistical indicators to assess if the selected metrics are stable enough:

1. σ measures the variation among a set of values. A *low* σ indicates that the values x_i tend to be close to the mean μ of the set, while a *high* σ indicates that the values are spread out over a broader range. The σ of a population of size N is computed by the following formula:

$$\sigma = \sqrt{\frac{\sum_{i \in N}(x_i - \mu)^2}{N}} \tag{1}$$

2. CV measures the dispersion of a probability distribution. The CV is computed by the following formula:

$$CV = \frac{\sigma}{\mu} \tag{2}$$

For each measurement of interest, the lower these values (σ and CV), the more stable.

4.2 Experimental Results

The experimental results are reported in Table 2. The first column gives the name of the project, the second, third, and fourth columns report on the medians of the CV for the software energy consumption ($\tilde{CV_E}$), the instructions ($\tilde{CV_I}$) and the cycles ($\tilde{CV_C}$) over all the tests executions; For the fifth and sixth columns, we computed the CV of the number of instructions (CV_I), of the number of cycles (CV_C) and the software energy consumption (CV_E) for each test. Then, we count the number of times that CV_E is greater than CV_I (column 3) and greater than CV_C (column 4); The eight and ninth columns are the Pearson correlation ratio (ρ) computed using medians for each test for the number of instructions (ρ_I) and the number of cycles (ρ_C), respectively.

For all the projects, the $\tilde{CV}$ of the software energy consumption ($\tilde{CV_E}$) is higher than both $\tilde{CV}$ of instructions ($\tilde{CV_I}$) and cycles ($\tilde{CV_C}$). The minimum is 0.02 for the $\tilde{CV_I}$ of xwiki and the maximum is 0.28 for the $\tilde{CV_E}$ of jsoup.

In a very large proportion, 87.51% is the minimum for jsoup, CV_E is greater than CV_I and CV_C.

Project	$C\tilde{V}_E$	$C\tilde{V}_I$	$C\tilde{V}_C$	$\%CV_E > CV_I$	$\%CV_E > CV_C$	ρ_I	ρ_C
jsoup	0.28	0.12	**0.10**	87.51%	95.52%	0.95	0.97
mustache.java	0.16	0.05	**0.05**	89.43%	95.05%	0.65	0.66
commons-io	0.25	**0.03**	0.05	95.93%	95.42%	0.70	0.89
gson	0.15	**0.04**	0.06	94.45%	95.35%	0.84	0.89
commons-lang	0.25	**0.03**	0.04	99.07%	99.15%	0.89	0.87
xwiki	0.09	**0.02**	**0.02**	91.75%	95.00%	0.87	0.87

Table 2 Summary of the ratio of commits on which the approach could be applied, the stability of the measured performance indicators per project and the correlation factor between energy consumption and performance indicators.

From the two last observations, we conclude that performance metrics—*i.e.*, number of instructions and number of cycles—are more stable than energy measurements both in terms of dispersion, estimated by the median of the CV and quantitative aspects, showed by the number of CV computed with energy measurements that is greater than the CV computed for both the number of instructions and the number cycles.

Furthermore, for all projects, the Pearson correlation ratios (ρ) are very high: the minimum is 0.65 (instr) for mustache.java, and the maximum is 0.97 (cycles) for jsoup. These high values of the Pearson correlation ratios show that the number of instructions and the number of cycles is highly correlated to the energy consumption of the tests. More precisely, the Pearson correlation ratio for the cycles (ρ_C) is slightly higher for the number of cycles than for the number of instructions (ρ_I). To illustrate these observations, we plot the distributions of the CV for the 3 measurements for each project in Figure 2.

Answer to RQ_1: we show that we can measure the energy consumption of tests among projects selected from GitHub. We also show that the measure of the number of instructions and the number of cycles is slightly more stable than the measure of energy consumption. In addition to this, the cycles and the instructions are highly correlated to energy consumption. Therefore, one can adopt these performance metrics to approximate energy consumption in a more stable and reproducible way. We particularly recommend using the number of cycles as a stable approximation of the energy consumption, hence validating H_2 over H_1.

In the remainder of this article, we adopt the number of cycles to detect regressions in the energy consumed by software tests.

5 Can developers' tests be used to detect potential energy regression introduced by code changes?

In this section, we investigate our second research question.We first explain our experimental protocol, and we then describe our results.

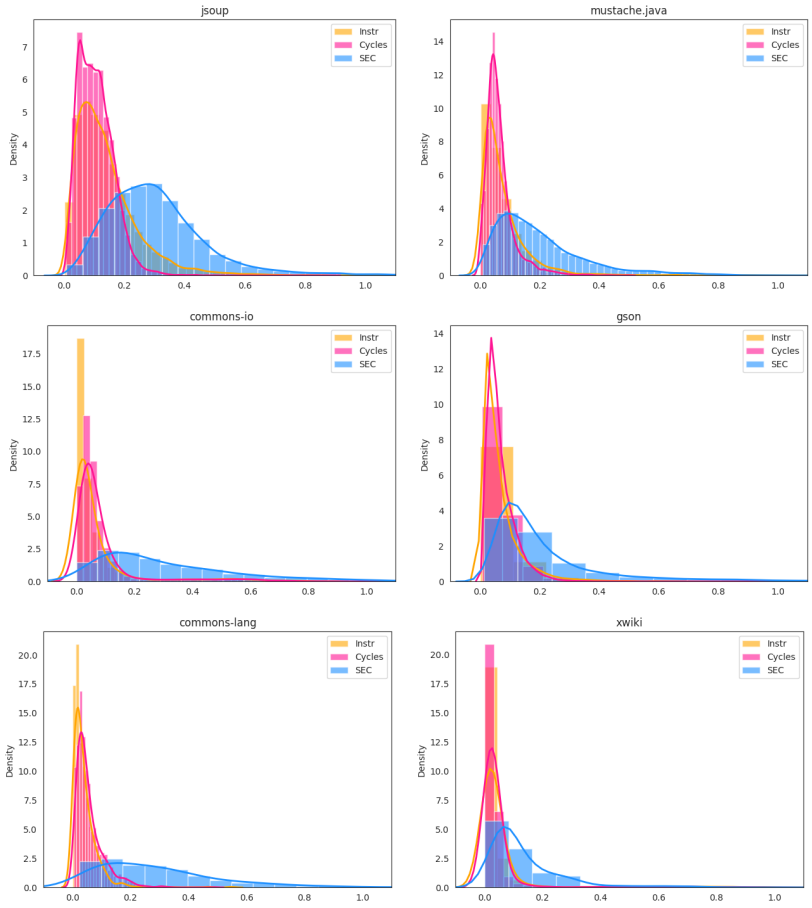

Fig. 2 Distributions of CV_E, CV_I, and CV_C for the 6 Java/Maven projects under study.

5.1 Experimental Protocol

To investigate H_3, we start by running a controlled experiment to assess the capability to detect regressions in the energy consumed by the test suite of a CI. To do so, we inject source code mutations for each selected repository to generate artificially new versions of the projects under study. The new versions are generated so that we control the level of regression in energy consumption. In practice, we inject energy-consuming statements at the beginning of some methods of the project's source code. These energy-consuming statements are side-effect-free instructions that are executed as long as a target energy consumption is not reached. Note that this way of mutating the code differs from the traditional mutation operators used in mutation-based test analysis. Then, to select the mutated methods, we rely on their code coverage and hit count—*i.e.*, the number of times the methods are executed during a run of the whole

test suite of the version to mutate. For each project, we select 5 methods with a low number of hits and 5 methods with a median number of hits. To determine what low and median mean, we compute the number of hits per method, sort the methods according to the number of hits and take the 5th percentile (± 2) for the low number of hits and the median (± 2) for the median number of hits. We did not consider methods with a larger number of hits, as it would lead to a very long experimentation time, given that the mutations—that increase the method execution time—would have to be executed many times.

Then, we consider two intensities of mutation: First, a *zero* mutation intensity that does not increase the energy consumption—*i.e.*, it is neutral regarding the energy consumption. Second, a *max* mutation intensity that drastically increases the energy consumption project-wise. To compute this max mutation intensity, we rely on the results of RQ_1, where we take the 95th percentile of energy consumption delta over all tests considered. Finally, we create $2 \times (5 + 5) = 20$ mutants representing each combination using these methods and intensities, leading to a total of 120 mutants for all the projects under study. Then, for each mutant, we measure the energy consumption of the tests selected on the version before (v_1) and after (v_2) the mutation.

As mentioned above, every source code mutation injects an energy payload as an invocation of method `consumeEnergy`, described in Algorithm 1, within each selected method.

Algorithm 1 Method `consumeEnergy` injecting a synthetic energy payload in mutated programs.

Require: Energy consumption probe: P
Require: Energy payload to inject (μJ): $payload$
Require: Seeded random generator: $R()$
Require: Current time: $t()$
1: $threshold \leftarrow P.startMonitoring() + payload$
2: $random \leftarrow R(t())$
3: **while** $P.getCurrentEnergyConsumed() < threshold$ **do**
4: $\quad random \leftarrow random + R(t())$
5: **end while**
6: $P.stopMonitoring(random)$

The method `consumeEnergy` first starts the monitoring of the energy consumption (line 1) and estimates the threshold consumption `threshold` by summing the energy payload `payload` (in our protocol, it corresponds to either a zero or a max increase). Then, it initializes a variable, called `random`, with a random value—using the seeded random generator with the current time as seed (line 2). It keeps looping as long as the current energy consumed is below the energy threshold (line 3). At each iteration of the loop, it accumulates the variable `random` with a new random value, using the seeded random generator with the current value of the `random` variable as seed (line 4). When the loop's condition is met, the method `consumeEnergy` stops the monitoring and returns. The `random` value is given as a parameter of method `consumeEn-`

ergy to escape *Just-in-Time* (JIT) optimisations of the JVM. This method allows us to *1)* artificially increase the energy consumption of a program without altering its functional behavior, *2)* configure the exact amount of energy to be consumed by the injected mutation, and *3)* bypass runtime optimizations, as the value of the random number cannot be predicted.

As described in Section 2.1, we apply a two-steps process to decide if the energy regression test suite of a commit is breaking or passing:

1. we discard the results of tests when the energy consumption measurement is too similar between the two versions using a test filter,
2. we aggregate the results of the remaining tests to take a decision (breaking or passing) by applying a decision oracle.

We apply different configurations for each step of this process when investigating RQ_2, as follows.

Test Filters. We consider three alternative test filters:

- all: this baseline filter does not remove any test.
- empty intersection: this aggressive filter removes all tests for which the intervals of values, defined by the real interval between the minimum and maximum values among the 100 computed values, have a non-empty intersection between the two versions.
- student's t-test: this filter uses the p-value of a t-test between the two sets of 100 measures for each version, as well as Cohen's d [16] effect size to determinate if a test has a significant variation of energy consumption. Any test yielding a p-value greater than 0.05 (no significance) or an effect size lesser than given thresholds (values too similar) are filtered out. We use four classical effect size thresholds [16]: 0.2, 0.5, 0.8, and 1.2.

Decision Oracle. Note that when the test filter discards all the tests, it indicates that the energy measurements of all the selected tests between the two versions are very similar. Therefore, we classify the commit as passing. When this is not the case, we consider 4 oracles to decide if the selected tests T of a given commit should be classified as breaking, as follows:

- strict: this very sensitive oracle labels a commit as breaking if there is at least one test-wise delta that is greater than zero: $\exists t \in T | \Delta(t) > 0$,
- aggregate: this oracle labels a commit as breaking if the sum of the deltas of the tests t is greater than zero, $\sum_{t \in T} \Delta(t) > 0$,
- code coverage: this oracle labels a commit as breaking if the sum of the deltas, weighted by the code coverage of the test t, is greater than zero, $\sum_{t \in T} \omega_{cocov}(t) \cdot \Delta(t) > 0$,
- diff coverage: this oracle labels a commit as breaking if the sum of the deltas, weighted by the diff coverage of the test t, is greater than zero, $\sum_{t \in T} \omega_{dicov}(t) \cdot \Delta(t) > 0$.

The delta of energy consumption, $\Delta(t)$, is defined as the difference of median energy consumptions measured for t between v_2 and v_1. Any positive $\Delta(t)$ detects a regression in energy consumption.

In the above formulas, $\omega_{cocov}(t)$ is defined as the ratio of lines covered by the test t over the total number of lines of the program. $\omega_{dicov}(t)$ is the ratio of added or deleted lines covered by the test t over the number of added and deleted lines, respectively. In these formulas, we pessimistically set $\epsilon = 0$ as the threshold value ϵ to detect an energy regression between two versions. However, it might turn out to be a too restrictive value. In these cases, it is possible to relax the threshold $\epsilon > 0$ to reduce the number of false positive commits considered as breaking. For instance, with such a threshold ϵ, the strict oracle would use the formula $\exists t \in T | \Delta(t) > \epsilon$

To validate our hypothesis H_3, we observe the ratio of mutants classified as breaking and passing. This protocol allows us to confront our approach against a change that does not increase the energy consumption, in which case we expect the change to be labeled as passing, and against a change that does increase the energy consumption, expecting the change to be labeled as breaking.

To further investigate H_3 on realistic code changes, we also apply this approach on the commits previously selected, as described in Section 4, and we report on the ratio of passing and breaking commits. Finally, we manually examine one arbitrary breaking and passing commit per project (picked from all the tested configurations and by trying to favor small commits that are easier to analyze) to gain more insights into the type of changes that lead to such decisions.

5.2 Experimental Results

The results are summarized in Table 3. For the sake of readability, the results are aggregated for all the projects under study. Note that we consider the number of cycles to compute the deltas in this research question, as our experimental results showed it to be a stable measurement highly correlated to energy consumption.

The lines represent the different combinations of decision oracles aggregated by test filters. Columns two to five represent the various mutants we generated. The second and third column report on the result for the zero mutation on methods that have few executions and on methods that have a median number of executions, respectively. The fourth and fifth column report on the result for the max mutation on methods that have few executions and on methods that have a median number of executions, respectively. Finally, the last column represents the commits selected in our dataset (see Section 3.2). The resulting columns respect the following layout: on the first row, the symbol ✓ is followed by the number of changes (for the mutations) or commits (for the real changes) labeled as passing, followed by the number of changes/-commits between parenthesis that were considered as passing directly after

Changes	Zero Low	Zero Med	Max Low	Max Med	Real
all					
strict	✓ 11 (0) ✗ 19 / 30	✓ 0 (0) ✗ 30 / 30	✓ 0 (0) ✗ 30 / 30	✓ 0 (0) ✗ 30 / 30	✓ 27 (0) ✗ 273 / 300
agg	✓ 16 (0) ✗ 14 / 30	✓ 8 (0) ✗ 22 / 30	✓ 1 (0) ✗ 29 / 30	✓ 1 (0) ✗ 29 / 30	✓ 137 (0) ✗ 163 / 300
cocov	✓ 16 (0) ✗ 14 / 30	✓ 9 (0) ✗ 21 / 30	✓ 1 (0) ✗ 29 / 30	✓ 1 (0) ✗ 29 / 30	✓ 154 (0) ✗ 146 / 300
dicov	✓ 22 (0) ✗ 8 / 30	✓ 15 (0) ✗ 15 / 30	✓ 14 (0) ✗ 16 / 30	✓ 10 (0) ✗ 20 / 30	✓ 217 (0) ✗ 83 / 300
∅ intersection					
strict	✓ 30 (30) ✗ 0 / 30	✓ 30 (29) ✗ 0 / 30	✓ 14 (13) ✗ 16 / 30	✓ 3 (3) ✗ 27 / 30	✓ 261 (239) ✗ 39 / 300
agg	✓ 30 (30) ✗ 0 / 30	✓ 30 (29) ✗ 0 / 30	✓ 14 (13) ✗ 16 / 30	✓ 3 (3) ✗ 27 / 30	✓ 269 (239) ✗ 31 / 300
cocov	✓ 30 (30) ✗ 0 / 30	✓ 30 (29) ✗ 0 / 30	✓ 14 (13) ✗ 16 / 30	✓ 3 (3) ✗ 27 / 30	✓ 272 (239) ✗ 28 / 300
dicov	✓ 30 (30) ✗ 0 / 30	✓ 30 (29) ✗ 0 / 30	✓ 19 (13) ✗ 11 / 30	✓ 10 (3) ✗ 20 / 30	✓ 290 (239) ✗ 10 / 300
t-test 0.20					
strict	✓ 24 (24) ✗ 6 / 30	✓ 13 (13) ✗ 17 / 30	✓ 2 (2) ✗ 28 / 30	✓ 1 (1) ✗ 29 / 30	✓ 75 (74) ✗ 225 / 300
agg	✓ 24 (24) ✗ 6 / 30	✓ 13 (13) ✗ 17 / 30	✓ 2 (2) ✗ 28 / 30	✓ 1 (1) ✗ 29 / 30	✓ 75 (74) ✗ 225 / 300
cocov	✓ 24 (24) ✗ 6 / 30	✓ 14 (13) ✗ 16 / 30	✓ 2 (2) ✗ 28 / 30	✓ 1 (1) ✗ 29 / 30	✓ 92 (74) ✗ 208 / 300
dicov	✓ 29 (24) ✗ 1 / 30	✓ 18 (13) ✗ 12 / 30	✓ 15 (2) ✗ 15 / 30	✓ 10 (1) ✗ 20 / 30	✓ 203 (74) ✗ 97 / 300
t-test 0.50					
strict	✓ 29 (29) ✗ 1 / 30	✓ 29 (29) ✗ 1 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 162 (162) ✗ 138 / 300
agg	✓ 29 (29) ✗ 1 / 30	✓ 29 (29) ✗ 1 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 162 (162) ✗ 138 / 300
cocov	✓ 29 (29) ✗ 1 / 30	✓ 29 (29) ✗ 1 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 169 (162) ✗ 131 / 300
dicov	✓ 30 (29) ✗ 0 / 30	✓ 29 (29) ✗ 1 / 30	✓ 17 (4) ✗ 13 / 30	✓ 10 (2) ✗ 20 / 30	✓ 237 (162) ✗ 63 / 300
t-test 0.80					
strict	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 183 (183) ✗ 117 / 300
agg	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 183 (183) ✗ 117 / 300
cocov	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 188 (183) ✗ 112 / 300
dicov	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 17 (4) ✗ 13 / 30	✓ 10 (2) ✗ 20 / 30	✓ 250 (183) ✗ 50 / 300
t-test 1.20					
strict	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 204 (204) ✗ 96 / 300
agg	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 204 (204) ✗ 96 / 300
cocov	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 4 (4) ✗ 26 / 30	✓ 2 (2) ✗ 28 / 30	✓ 207 (204) ✗ 93 / 300
dicov	✓ 30 (30) ✗ 0 / 30	✓ 30 (30) ✗ 0 / 30	✓ 17 (4) ✗ 13 / 30	✓ 10 (2) ✗ 20 / 30	✓ 262 (204) ✗ 38 / 300

Table 3 Summary of the number of injected mutations or commits considered (–), labeled as breaking (✗) or passing (✓). The total number is reported after the symbol "/".

the test filtering step (meaning that for this change/commit, the test filter discarded all the tests because the energy measurement was too similar). On

the second row, the symbol ✗ is followed by the number of changes labeled as breaking. Then, the symbol / is followed by the total number of changes for the given combination of test filter and decision oracle.

We start by focusing on the results for the zero mutants (second and third columns), as it will enable us to identify configurations that are too sensitive. Indeed, any zero mutant that is identified as breaking is a false positive. Any configuration labeling a zero mutant as breaking is likely too sensitive in practice. It turns out that it is the case for the following test filters: all, t-test 0.20, and, to a lesser extent, t-test 0.50 (that only yields one false positive). Interestingly, even by changing the decision oracles, these configurations consistently label at least one zero mutant as breaking.

Then, we focus on the max mutants for the remaining configurations (fourth and fifth columns), as we expect all these mutants to be labeled as breaking. Any max mutant labeled as passing can be considered a false negative. However, as we do not have a ground truth workload, we cannot ensure that it impacts energy consumption meaningfully. Interestingly, when considering the strict, agg or cocov decision oracles, the results are identical for t-test 0.80 and t-test 1.20 filters. In this controlled experiment, these combinations seem to be the best-performing ones, reporting no false positives and only 6 false negatives. The dicov decision oracle appears as less sensitive than the others since it augments the number of false positives for t-test 0.80 and t-test 1.20. Finally, the empty-intersection test filter is clearly the least sensitive since it yields many more false negatives, especially for the low mutants. In combination with the least sensitive decision oracle (dicov), empty-intersection yields more about 63% false negatives.

The results for our selected commits (last column) confirm the trend we observed on the mutants. Our very sensitive decision oracles (all, t-test 0.20 and t-test 0.50) label the majority of commits as breaking (unless used with the dicov decision oracle). In practice, these configurations would probably raise too many alerts making them very tedious to use. The t-test 0.80 and t-test 1.20 that yielded identical results for the mutants exhibit differences on the selected commits. As expected, the t-test 1.20 filter is less sensitive than the t-test 0.80 filter. Finally, the empty-intersection remains the least sensitive one. Even using the very sensitive all decision oracle, it labels only 39 commits out of 300 as breaking. Using the least sensitive dicov decision oracle, it labels only 10 commits out of 300 as breaking.

Answer to RQ_2: According to our experimental results, we observe that our test filters and commit decision strategies allow us to define configurations that range from very sensitive (very prone to classify commit as breaking) to conservative (rarely classifying commits as breaking). The all, t-test 0.20, and t-test 0.50 seem too sensitive as, whatever the decision oracles, they yield at least one false positive on the zero mutants. As expected, the t-test 0.80 and t-test 1.20 are less sensitive and yield the fewest false negatives on the max mutants when used with the strict, agg or cocov decision oracles. However, they label as breaking from half to one-third of the selected commits (unless used with the dicov decision oracle). In practice, they would therefore yield a high number of alerts. The most conservative configuration is achieved using empty-intersection filter and *dicov* oracle, which labels as breaking only 10 out of the 300 selected commits.

5.3 Manual Assessment

In this section, we manually analyze an arbitrary sample of commits classified as "breaking' or passing by the different configurations explored in our experiments.

5.3.1 JSoup

Breaking Commit. Listing 1 shows the diff introduced by commit `cc2363e`.[7] In this commit, a loop that looks up to parents in a hierarchy of elements is refactored. To look up the parents, the implementation now uses a `while` loop that calls to `parent()` at each iteration instead of relying on an object of type `Elements`, which is a subtype of `ArrayList`) that is precomputed by a `parents()` method (inserted code starting on Line 22). It is not obvious to explain why there would be an increase in energy consumption in this case, as using the `while` loop should avoid the creation of an `ArrayList`. This could be the occurence of a false positive yielded by our approach.

Passing Commit. Listing 2 shows the diff introduced by commit `011e83f`.[8] In this commit, a boolean expression is updated, and the new version uses a comparison to a variable instead of a constant (cf. Line 6). This commit is probably a bug fix, as the previous version seemed to assume that the last iteration covers a `pos` equals 0. This is only a tiny modification that does not involve any memory allocation. Therefore, there is no evidence to support a potential energy regression, explaining why the commit is classified as passing.

[7] `https://github.com/jhy/jsoup/commit/cc2363e`
[8] `https://github.com/jhy/jsoup/commit/011e83f`

```
--- src/main/java/org/jsoup/nodes/Element.java
+++ src/main/java/org/jsoup/nodes/Element.java
@@ -258,7 +258,7 @@
         return attributes().dataset();
     }

-    @Override
+    @Override @Nullable
     public final Element parent() {
         return (Element) parentNode;
     }
--- src/main/java/org/jsoup/parser/HtmlTreeBuilder.java
+++ src/main/java/org/jsoup/parser/HtmlTreeBuilder.java
@@ -138,13 +138,13 @@

             // setup form element to nearest form on context (up ancestor chain). ensures form controls are associated
             // with form correctly
-            Elements contextChain = context.parents();
-            contextChain.add(0, context);
-            for (Element parent: contextChain) {
-                if (parent instanceof FormElement) {
-                    formElement = (FormElement) parent;
+            Element formSearch = context;
+            while (formSearch != null) {
+                if (formSearch instanceof FormElement) {
+                    formElement = (FormElement) formSearch;
                     break;
                 }
+                formSearch = formSearch.parent();
             }
         }
```

Listing 1 Diff of the selected breaking commit for jsoup.

```
--- src/main/java/org/jsoup/parser/HtmlTreeBuilder.java
+++ src/main/java/org/jsoup/parser/HtmlTreeBuilder.java
        LOOP: for (int pos = bottom; pos >= upper; pos--) {
            Element node = stack.get(pos);
-           if (pos == 0) {
+           if (pos == upper) {
                last = true;
                if (fragmentParsing)
                    node = contextElement;
```

Listing 2 Diff of the selected passing commit for jsoup.

5.3.2 mustache.java

Breaking Commit. Listing 3 shows the diff introduced by commit `909fc58`.[9] This commit modifies a method that appends a string, stored in the `appended` attribute) to a writer supplied as a parameter. After the commit, the `appended` attribute is cached into an array of chars inside the introduced `appendedChars` attribute, which is supplied to the `Writer`, avoiding the mandatory conversion from `String` to `char[]` that is made by the writer inside the standard library code. According to the commit message (*no reason to keep converting this*

[9] `https://github.com/spullara/mustache.java/commit/909fc58`

```
--- src/main/java/com/github/mustachejava/codes/DefaultCode.java
+++ src/main/java/com/github/mustachejava/codes/DefaultCode.java
@@ -192,10 +192,16 @@
      writer.write(tc.endChars());
    }

+  private char[] appendedChars;
+
    protected Writer appendText(Writer writer) {
      if (appended != null) {
        try {
-         writer.write(appended);
+         // Avoid allocations at runtime
+         if (appendedChars == null) {
+           appendedChars = appended.toCharArray();
+         }
+         writer.write(appendedChars);
        } catch (IOException e) {
          throw new MustacheException(e);
        }
```

Listing 3 Diff of the selected breaking commit for mustache.java.

string), we conjecture this is a commit aiming to improve performance. With the test's workload, this expected performance improvement increases energy consumption, as the overhead of the new attribute might not outweigh the gain of using the cache. However, with a realistic production workload, the result could be different. This case demonstrates the usefulness of performing regular energy regression testing, as some expected optimizations can be detrimental in practice.

Passing Commit. Listing 4 shows the diff introduced by commit `f7a0c86`.[10] This commit adds a check before executing the body of a method. According to the comment, this check prevents executing code for a particular combination of input parameters. Typically, under a production workload, one would expect to detect a slight energy regression due to the additional check. However, with a test workload, the edge case path may also be exercised, resulting in a stable or energy regression, thanks to the early return in this case.

5.3.3 commons-io

Breaking Commit. Listing 5 shows the diff introduced by commit `f281d13`.[11] In this commit, an ad-hoc implementation to read a file as an array of bytes is replaced by a call to a standard library's method introduced in JDK 11. Additionally, a null-check on the method's input parameter is introduced via a call to `requireNonNull`. The observed regression in energy consumption seems legit for two reasons: first, the `requireNonNull` call increases the energy consumption as the developer added this null-check. Second, we conjecture

[10] `https://github.com/spullara/mustache.java/commit/f7a0c86`
[11] `https://github.com/apache/commons-io/commit/f281d13`

```
1  --- src/m/j/c/g/mustachejava/reflect/BaseObjectHandler.java
2  +++ src/m/j/c/g/mustachejava/reflect/BaseObjectHandler.java
3  @@ -144,6 +144,9 @@
4      }
5
6      protected AccessibleObject findMember(Class sClass, String name) {
7  +     // under java11 it would return a wrapper we don't want
8  +     if (String.class == sClass && \"value\".equals(name)) {
9  +       return null;
10 +     }
11       AccessibleObject ao;
12       try {
13         ao = getMethod(sClass, name);
```

Listing 4 Diff of the selected passing commit for mustache.java.

```
1  --- src/main/java/org/apache/commons/io/FileUtils.java
2  +++ src/main/java/org/apache/commons/io/FileUtils.java
3  @@ -2608,11 +2608,8 @@
4       public static byte[] readFileToByteArray(final File file) throws
            IOException {
5  -         try (InputStream inputStream = openInputStream(file)) {
6  -             final long fileLength = file.length();
7  -
    // file.length() may return 0 for system-dependent entities,
8  -                 treat 0 as unknown length - see IO-453
9  -             return fileLength > 0 ?
10 -                 IOUtils.toByteArray(inputStream, fileLength) :
11 -                 IOUtils.toByteArray(inputStream);
12 -         }
13 +         Objects.requireNonNull(file, \"file\");
14 +         return Files.readAllBytes(file.toPath());
15       }
```

Listing 5 Diff of the selected breaking commit for commons-io.

that using a standard library's method instead of an ad-hoc implementation might increase energy consumption, as such methods usually cover all possible edge cases.

Passing Commit. For commons-io, only one commit has been labelled as passing, which is the same as the analyzed breaking one, but obtained with a different configuration. This highlights that configuring the sensitivity of the approach is very important.

5.3.4 GSON

Breaking Commit. Listing 6 shows the diff introduced by commit `d9cc7bc`. [12] In this commit, a copy of a list is introduced to avoid working directly on the original list (cf. Line 10). In that case, we can effectively guess that copying an array consumes more energy than not doing it, hence explaining the energy regression.

[12] `https://github.com/google/gson/commit/d9cc7bc`

```
1  --- src/main/java/com/google/gson/GsonBuilder.java
2  +++ src/main/java/com/google/gson/GsonBuilder.java
3  @@ -562,8 +562,11 @@
4        List<TypeAdapterFactory> factories = new ArrayList<
           TypeAdapterFactory>(this.factories.size() + this.
           hierarchyFactories.size() + 3);
5        factories.addAll(this.factories);
6        Collections.reverse(factories);
7  -     Collections.reverse(this.hierarchyFactories);
8  -     factories.addAll(this.hierarchyFactories);
9  +
10 +     List<TypeAdapterFactory> hierarchyFactories =
11 +         new ArrayList<TypeAdapterFactory>(this.hierarchyFactories);
12 +     Collections.reverse(hierarchyFactories);
13 +     factories.addAll(hierarchyFactories);
14 +
15       addTypeAdaptersForDate(datePattern, dateStyle, timeStyle,
           factories);
```

Listing 6 Diff of the selected breaking commit for GSON.

```
1  --- src/main/java/com/google/gson/stream/JsonReader.java
2  +++ src/main/java/com/google/gson/stream/JsonReader.java
3  @@ -1006,13 +1006,12 @@
4          } else if (c == '\\') {
5            pos = p;
6            int len = p - start - 1;
7  -          char escapeChar = readEscapeCharacter();
8            if (builder == null) {
9  -           int estimatedLength = (len + pos - p) * 2;
10 +           int estimatedLength = (len + 1) * 2;
11             builder = new StringBuilder(Math.max(estimatedLength, 16)
                 );
12           }
13           builder.append(buffer, start, len);
14 -         builder.append(escapeChar);
15 +         builder.append(readEscapeCharacter());
16           p = pos;
17           l = limit;
18           start = p;
```

Listing 7 Diff of the selected passing commit for GSON.

Passing Commit. Listing 7 shows the diff introduced by commit `4644837`.[13]

In this commit, there are two modifications. First, a temporary variable is deleted, and its access is replaced by a direct call (cf. Lines 7 and 15). Second, the computation of a value is modified and notably uses fewer variables than the previous version (cf. Line 10). Therefore, the fact that there is no noticeable increase in energy consumption is expected, as the new version of the code allocates and reads fewer variables.

[13] `https://github.com/google/gson/commit/4644837`

```
1  --- src/main/java/org/apache/commons/lang3/RegExUtils.java
2  +++ src/main/java/org/apache/commons/lang3/RegExUtils.java
3  @@ -256,7 +256,7 @@
4         * @see java.util.regex.Pattern
5         */
6        public static String replaceAll(final String text, final Pattern
             regex, final String replacement) {
7  -         if (text == null || regex == null || replacement == null) {
8  +         if (ObjectUtils.anyNull(text, regex, replacement)) {
9                return text;
10           }
11           return regex.matcher(text).replaceAll(replacement);
12 @@ -310,7 +310,7 @@
13        * @see java.util.regex.Pattern#DOTALL
14        */
15       public static String replaceAll(final String text, final String
             regex, final String replacement) {
16 -         if (text == null || regex == null || replacement == null) {
17 +         if (ObjectUtils.anyNull(text, regex, replacement)) {
18               return text;
19           }
20           return text.replaceAll(regex, replacement);
21 @@ -449,7 +449,7 @@
22        * @see Pattern#DOTALL
23        */
24       public static String replacePattern(final String text, final
             String regex, final String replacement) {
25 -         if (text == null || regex == null || replacement == null) {
26 +         if (ObjectUtils.anyNull(text, regex, replacement)) {
27               return text;
28           }
29           return Pattern.compile(regex, Pattern.DOTALL).matcher(text).
                 replaceAll(replacement);
```

Listing 8 Diff of the selected breaking commit for commons-lang.

5.3.5 commons-lang

Breaking Commit. Listing 8 shows the diff introduced by commit `615c1da`.[14] In this commit, the content of three boolean expressions is refactored similarly. In these expressions, it is ensured that the three parameters supplied to their containing methods are not null, using the `||` logical operator. These ad-hoc expressions are replaced by a call to an equivalent helper method `ObjectUtils.anyNull()`. The implementation of `anyNull(Object... values)` uses a variadic parameter. While arguably more readable, it has the overhead of creating an array to group the parameters supplied to the method, which is not the case in the replaced ad-hoc expression. This could explain the energy regression.

Passing Commit. Listing 9 reports on the diff introduced by commit `bfbf729`.[15] In this commit, an ad-hoc expression that computes the size of an array, using 0 in case of a null value, is replaced by a call to an equivalent helper method

[14] `https://github.com/apache/commons-lang/commit/615c1da`
[15] `https://github.com/apache/commons-lang/commit/bfbf729`

```
1  --- src/main/java/org/apache/commons/lang3/StringUtils.java
2  +++ src/main/java/org/apache/commons/lang3/StringUtils.java
3  @@ -6865,10 +6865,7 @@
4   * @since 2.4
5   */
6   public static String replaceEachRepeatedly(final String text, final
        String[] searchList, final String[] replacementList) {
7  -    // timeToLive should be 0 if not used or nothing to replace,
8  -    // else it's the length of the replace array
9  -    final int timeToLive = searchList == null ? 0 : searchList.length;
10     return replaceEach(
11 -        text, searchList, replacementList, true, timeToLive
12 +        text, searchList, replacementList, true,
13 +        ArrayUtils.getLength(searchList)
14     );
15  }
```

Listing 9 Diff of the selected passing commit for commons-lang.

`ArrayUtils.getLenght(Object)`. This helper method defers part of the computation to the standard library method `Array.getLenght(Object)` from the `reflect` package. The main difference with the original code is that there are several subtype tests inside the standard library code to prevent supplying a non-array type and also to consider the possibility of having an array from a primitive type. These introduced subtype tests could explain the increase in energy consumption.

5.3.6 xwiki

Breaking Commit. Listing 10 shows the diff introduced by commit `9464cc7`.[16] In this commit, the chain of calls to create an object of type `Reflections` is updated to comply with the upgrade to the new version of the eponymous library. Since the incriminated code is not inside the project, it is difficult to explain the observed energy regression, but it highlights the fact that upgrading dependencies does not necessarily have a neutral effect.

Passing Commit. Listing 11 shows the diff introduced by commit `5f85426`.[17] The most notable change is the removal of a temporary `parameters` variable (cf. Line 4). The expression that was evaluated to give the value of the variable is now directly passed to the `configure()` method (cf. Line 15). In this case, there is no evidence of any additional computation or allocation, therefore the decision of classifying the commit as passing seems logical.

Manual Analysis: Our manual review of a sample of commits labeled as breaking and passing indicated that, in most cases, we could understand the reason behind the decision. However, it also shows that the approach is likely to be prone to yield false positives or negatives.

[16] `https://github.com/xwiki/xwiki-commons/commit/9464cc7`
[17] `https://github.com/xwiki/xwiki-commons/commit/5f85426`

```
--- src/m/j/o/x/extension/internal/DefaultExtensionLicenseManager.java
+++ src/m/j/o/x/extension/internal/DefaultExtensionLicenseManager.java
@@ -83,8 +83,8 @@
  Collection<URL> licenseURLs = ClasspathHelper.forPackage(
      LICENSE_PACKAGE);

  Reflections reflections = new Reflections(new ConfigurationBuilder()
-   .setScanners(new ResourcesScanner()).setUrls(licenseURLs)
-   .filterInputsBy(
-     new FilterBuilder.Include(FilterBuilder.prefix(LICENSE_PACKAGE)))
-   );
+   .setScanners(Scanners.Resources).setUrls(licenseURLs)
+   .filterInputsBy(
+     new FilterBuilder().includePackage(LICENSE_PACKAGE))
+   );
    for (String licenseFile : reflections.getResources(Pattern.compile
        (\".*\\\\.license\"))) {
      URL licenseUrl = getClass().getClassLoader().getResource(
          licenseFile);
```

Listing 10 Diff of the selected breaking commit for xwiki.

```
--- src/main/java/org/xwiki/job/internal/DefaultJobStatusStore.java
+++ src/main/java/org/xwiki/job/internal/DefaultJobStatusStore.java
@@ -145,14 +144,9 @@
-   PropertiesBuilderParameters parameters = new Parameters()
-     .properties();
-   if (file.exists()) {
-     new Parameters().properties().setFile(file);
-   }
-
    FileBasedConfigurationBuilder<PropertiesConfiguration> builder =
-   new FileBasedConfigurationBuilder<>(PropertiesConfiguration.class)
-     .configure(parameters);
+   new FileBasedConfigurationBuilder<>(
+     PropertiesConfiguration.class, null, true
+   ).configure(new Parameters().properties().setFile(file));
@@ -160,7 +154,7 @@
-   builder.getFileHandler().save(file);
+   builder.save();
    }
  } catch (Exception e) {
    this.logger.error(\"Failed to load jobs\", e);
```

Listing 11 Diff of the selected passing commit for xwiki.

6 Deviations & Threats to Validity

In this section, we first report on the deviations from the pre-registered protocol [8]. Then, we discuss the main threats to the validity of our study, following the structure recommended by Wohlin et al. [17].

6.1 Deviations from the Pre-registered Protocol

Quartile Coefficient of Dispersion. The first deviation is that we dropped the use of the quartile coefficient of dispersion in RQ_1, mainly because it was

redundant with the coefficient of variation since both of them give the same information, that is to say, the stability of the values.

Fault Localization. In the pre-registered report, we wanted to explore the possibility of applying fault localization techniques for breaking commits to give the developers hints about which portion of the modified code would be responsible for the energy regression. However, we decided to remove this extra step and associated RQ_3 and focus on the measurements, stability, and detection of energy regressions, as it has proven to be very challenging.

Tool for Energy Consumption Measurements. We developed a new C library to measure the energy consumption of any process using an API to start and stop monitoring of code snippets. This library replaced J-Joules and JUnit-JJoules, which we planned to use initially, which are Java-specific and less accurate.

Targeted Dataset. We changed the dataset from the one proposed by Ournani et al. [2] to the one proposed by Danglot et al. [13]. The main reason for this change is that performing test selection is difficult as it requires the projects to be compatible with a specific test framework. Since these projects had already been used with the test selection implementation in [13], it allowed us to focus on the key steps of our study. Moreover, the projects from both datasets are comparable, as they are all popular Java libraries with comprehensive test suites.

6.2 Threats to Validity

Construct Validity. An important threat is that developers' tests might not be a realistic representation of a classical production workload. It means that the energy consumption we compute and the decision to label a commit as breaking or passing might change with a relevant workload. This problem is even more apparent for the formulas where we aggregate the test-wise measurements (see Section 5), where the increase of consumption of some tests could be compensated by the decrease on other tests and go unnoticed (although we did not find any evidence of such cases in our experiment). Unfortunately, the projects of our corpus do not have any baseline workload that we could use as ground truth to assess the importance of this threat. To reduce the importance of this threat, we selected only software libraries in our corpus, where we conjecture that unit tests cover at least some realistic clients' use cases. A second mitigation measure is that we proceeded to a manual qualitative analysis of several commits to put the automated approach's decisions into perspective.

Internal Validity. One major threat relates to the stability of the measurements. It is well-known that energy measurements captured by power meters, like RAPL, are unstable. Therefore, there is a threat that an energy measurement is not representative of the true energy consumption. We followed the guidelines proposed by Ourmani et al. [5] to reduce the noise introduced by external events.

Another threat is that we developed a complex experimentation code, as well as two software libraries, Diff-test-selection and TLPC-sensor, to perform our experiments. We cannot guarantee that all these programs are bug-free and, therefore, some measurements might turn out to be incorrect. We unit-tested these programs to reduce this threat. As a complement, we also made our source code freely available for scrutiny.

External Validity. Our study uses a small corpus of Java libraries with a high-coverage test suite (at least 80% code coverage). Therefore, we cannot guarantee that the results would be the same on different kinds of projects using a different programming stack and with test suites with lower coverage.

7 Related Work

Pereira et al. [1] defined a technique called *SPectrum-based Energy Leak Localization* (SPELL), which highlights energy hotspots in a program. SPELL relies on fault localization techniques that collect software entities (methods, classes, lines, etc.) executed by use cases. They implemented SPELL in Java and experimented with it on 5 projects to demonstrate that SPELL can identify energy hotspots. The evaluation showed that SPELL helped the developers to improve energy consumption.

Mancebo et al. [3] analyzed the relation between maintainability and energy consumption of different versions of Redmine. They computed the maintainability using SonarQube and measured the energy consumption using an *Energy Efficiency Tester* (EET) appliance. They concluded that the number of code lines and the energy consumed is correlated.

Maia et al. [6] proposed the concept of energy debt, which is the amount of energy a system consumes over time due to energy code smells. In this work, the authors defined a set of Android energy code smells and developed E-Debitum, a SonarQube plugin that computes the energy debt between versions of Android applications. They evaluated their approach on 3 Android applications and proved that it could be applied to real-world applications by demonstrating the evolution of their energy debt over time.

Luo et al. [18] proposed a technique called PerfImpact, which recommends inputs and code changes related to performance regressions. To do this, they combine search-based and change impact analysis. They evaluated PerfImpact on 2 OSS projects. They showed that PerfImpact can identify performance issues between 2 versions of a program. PerfImpact is focused

on determining the specific inputs and the code changes that trigger a performance regression in terms of time execution, while, in our study, we leverage the input provided by developers' tests to observe if there are energy regressions. Both of our approaches take place in the context of CI and performance regression, even if our approach is narrowed to energy consumption regression, which can be seen as a special case of performance regression.

Chen et al. [19] performs an exploratory study on the code changes that introduce performance regressions. They perform an evaluation on 1,126 commits from Hadoop, and 135 commits from RxJava. The evaluation reported 243 and 91 commits introducing performance regression in Hadoop and RxJava, respectively. They also identified 6 root causes of performance regressions introduced by code changes.

Ding et al. [10] study the use of tests in the release pipeline as performance tests of Hadoop and Cassandra, 2 OSS projects. They show that, for 102 out of 127 of the performance issues, at least one test can be used to spot a performance improvement. This study reinforces our hypothesis that developers' tests can be used to detect energy regression, as energy consumption is correlated to the program's performance.

Hindle et al. [20] devise GreenMiner. GreenMiner is a dedicated hardware/software mining software repositories test harness. GreenMiner has been created to study relationships between code changes and power consumption. The main difference is that GreenMiner uses physical measurements and works only for Android applications, while we use RAPL to measure energy consumption, an approach that could be applied to any software.

Hindle [21] presents *Green mining*. *Green mining* attempts to estimate empirically the impact of software change on energy consumption to guide developers in reducing SEC. *Green mining* also comes with an abstract methodology that gives key steps to set up the approach on an application. The author performs an evaluation on multiple branches of Firefox, on the release history of Azureus/Vuze, and a study of the effect of multiple versions of the library libTorrent on the energy consumption of its client application rTorrent. The key difference between *Green mining* and our approach is that we try to identify the impact on energy consumption of code changes, while *Green mining* operates at a more coarse-grained level.

Chowdhury et al. [22] propose GreenOracle, a model to predict energy consumption. This model is trained using a large corpus of Android applications, taking as input their system calls, dynamic traces, CPU utilization, and GreenMiner data. They collected 984 versions of 24 different Android applications and estimated energy consumption with less than 10% error.

Romansky et al. [23] propose an approach called *Deep Green*, to construct models that use software performance measurements to predict instantaneous energy consumption based on time series. The authors use GreenMiner to train models to estimate software energy consumption.

The main difference between our approach and the two last works, GreenOracle and *Deep Green*, is that we detect the energy regression from developer

tests, while GreenOracle is based on predictions of the energy consumption of program versions.

8 Conclusion & Future works

In this article, we introduced an approach that leverages developers' tests to automatically detect energy regression in CI. We implemented and evaluated our approach on a dataset of OSS projects. We showed that our approach could measure energy consumption stably on a ratio of commits ranging from 4% to 40%. We also showed that this approach could be tuned by changing the test filters or decision oracles, making it possible to adapt its sensitivity to the context. In addition to this, our approach could effectively detect artificial energy regressions, and we analyzed manually some of the energy regressions detected among the real-world commits. The manual analysis could, in most cases, corroborate the decision made by our approach.

The main limitation of our study, as explained in Section 6.2, is that we miss a ground-truth workload to assess if the regressions detected by our approach are relevant or not for an application. However, we hypothesize that it should be the case on projects where the test suite covers code that is close to the intended usage.

An important caveat of our approach is that measuring representative energy consumption is expensive. Therefore, it might turn out that the energy consumption savings obtained when running the program do not compensate for the energy consumed by our approach, depending on the context. Another caveat is that while our approach can detect a major regression between two successive versions, it might not be robust to several minor successive regressions.

Regarding future works, we first plan to extend our approach with the use of fault localization techniques [24], as it was initially planned [8]. It would assist developers in pinpointing the most suspicious lines of code causing an energy regression, hence better guiding the analysis of breaking commits.

A notable pain point of our approach is that it requires executing the whole test suite on both versions: before and after the commit and computing the code coverage for each test in both versions. This introduces a significant overhead. We plan to leverage regression test selection techniques [25] to avoid launching all tests on the second version.

Conflict of interest

The authors declare that they have no conflict of interest.

Data Availability Statements

Our implementation is available as an Open-source Software repository on GitHub: `https://github.com/davidson-consulting/diff-jjoules`. The data produced by our experiments are available on Zenodo `https://doi.org/10.5281/zenodo.6528916`. Scripts to perform the experiments and generate graph are available on GitHub: `https://github.com/davidson-consulting/diff-jjoules-experiment`.

References

1. R. Pereira, T. Carção, M. Couto, J. Cunha, J. P. Fernandes, and J. Saraiva, "Spelling out energy leaks: Aiding developers locate energy inefficient code," *Journal of Systems and Software*, vol. 161, p. 110463, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219302377
2. Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "Tales from the Code #1: The Effective Impact of Code Refactorings on Software Energy Consumption," in *ICSOFT 2021 - 16th International Conference on Software Technologies*, Virtual, France, Jul. 2021. [Online]. Available: https://hal.archives-ouvertes.fr/hal-03202437
3. J. Mancebo, C. Calero, and F. García, "Does maintainability relate to the energy consumption of software? A case study," *Softw. Qual. J.*, vol. 29, no. 1, pp. 101–127, 2021. [Online]. Available: https://doi.org/10.1007/s11219-020-09536-9
4. J. Mancebo, F. García, and C. Calero, "A process for analysing the energy efficiency of software," *Inf. Softw. Technol.*, vol. 134, p. 106560, 2021. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106560
5. Z. Ournani, R. Rouvoy, P. Rust, and J. Penhoat, "On Reducing the Energy Consumption of Software: From Hurdles to Requirements," in *ESEM 2020 - ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, Bari, Italy, Oct. 2020. [Online]. Available: https://hal.inria.fr/hal-02892900
6. D. Maia, M. Couto, J. Saraiva, and R. Pereira, "E-debitum: managing software energy debt," in *35th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2020, Melbourne, Australia, September 21-25, 2020*, J. Grundy, C. L. Goues, and D. Lo, Eds. ACM, 2020, pp. 170–177. [Online]. Available: https://doi.org/10.1145/3417113.3422999
7. M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An explorative analysis of travis ci with github," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 356–367.
8. B. Danglot, J. Falleri, and R. Rouvoy, "Can we spot energy regressions using developers tests?" *CoRR*, vol. abs/2108.05691, 2021. [Online]. Available: https://arxiv.org/abs/2108.05691
9. G. Meszaros, *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.
10. Z. Ding, J. Chen, and W. Shang, "Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet?" in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 1435–1446.
11. Z. Ournani, M. C. Belgaid, R. Rouvoy, P. Rust, J. Penhoat, and L. Seinturier, "Taming energy consumption variations in systems benchmarking," in *ICPE*. ACM, 2020, pp. 36–47.
12. W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 1977.
13. B. Danglot, M. Monperrus, W. Rudametkin, and B. Baudry, "An approach and benchmark to detect behavioral changes of commits in continuous integration," *Empirical Software Engineering*, vol. 25, no. 4, pp. 2379–2415, Jul. 2020. [Online]. Available: https://hal.inria.fr/hal-03121735

14. V. M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore, "Measuring energy and power with papi," in *2012 41st International Conference on Parallel Processing Workshops*, 2012, pp. 262–268.
15. R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code," *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01078532/document
16. J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*. Routledge, May 2013. [Online]. Available: https://doi.org/10.4324/9780203771587
17. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, 2012th ed. Berlin, Germany: Springer, Jun. 2012.
18. Q. Luo, D. Poshyvanyk, and M. Grechanik, "Mining performance regression inducing code changes in evolving software," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 25–36. [Online]. Available: https://doi.org/10.1145/2901739.2901765
19. J. Chen and W. Shang, "An exploratory study of performance regression introducing code changes," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 341–352.
20. 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 *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 12–21. [Online]. Available: https://doi.org/10.1145/2597073.2597097
21. A. Hindle, "Green mining: a methodology of relating software change and configuration to power consumption," *Empirical Software Engineering*, vol. 20, no. 2, pp. 374–409, 2015.
22. 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*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 49–60. [Online]. Available: https://doi.org/10.1145/2901739.2901763
23. S. Romansky, N. C. Borle, S. Chowdhury, A. Hindle, and R. Greiner, "Deep green: Modelling time-series of software energy consumption," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 273–283.
24. W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.
25. E. Engström, P. Runeson, and M. Skoglund, "A systematic review on regression test selection techniques," *Information and Software Technology*, vol. 52, no. 1, pp. 14–30, 2010.