Performance-based Guidelines for Energy Efficient Mobile Applications

Luis Cruz
University of Porto
HASLab/INESC TEC
Porto, Portugal
luiscruz@fe.up.pt

Rui Abreu
Instituto Superior Técnico, University of Lisbon
INESC-ID
Lisbon, Portugal
rui@computer.org

***Abstract*—Mobile and wearable devices are nowadays the *de facto* personal computers, while desktop computers are becoming less popular. Therefore, it is important for companies to deliver efficient mobile applications. As an example, Google has published a set of best practices to optimize the performance of Android applications. However, these guidelines fall short to address energy consumption. As mobile software applications operate in resource-constrained environments, guidelines to build energy efficient applications are of utmost importance. In this paper, we studied whether or not eight best performance-based practices have an impact on the energy consumed by Android applications. In an experimental study with six popular mobile applications, we observed that the battery of the mobile device can last up to approximately an extra hour if the applications are developed with energy-aware practices. This work paves the way for a set of guidelines for energy-aware automatic refactoring techniques.**

***Index Terms*—Green Computing; Mobile Computing; Anti patterns;**

I. Introduction

Energy consumption in mobile applications is a concern for both users and developers. Applications that drain battery life of mobile devices can ruin user experience, and therefore tend to be removed, unless they offer a crucial feature to users. Thus, any improvement in the energy efficiency of an application has a great impact on its success. Nevertheless, mobile application development is usually feature oriented. This means that nonfunctional requirements such as energy efficiency might not be so perceivable during the development phase. In that case, energy inefficiencies are spotted only after publishing the application to users [1].

A study on mobile application usage patterns have collected data from 4,125 users between August, 2010 and January, 2011 [2]. In this study, users spent on average one hour per day interacting with their mobile device. Recent studies revealed that in 2016 the usage has increased to 2.4 hours per day and 3.75 hours for heavy users[1]. This trend suggests that users will keep increasing mobile phone usage and consequently having energy efficient software is an important matter.

Energy consumption can be affected by many factors such as temperature, background tasks, and active components. Different executions of the same code may have different consumption. Thus, the impact of energy improvements is hard to measure and time consuming. Previous work shows the lack of knowledge developers have on energy efficiency in mobile applications [3]. Having a notion of practices that benefit energy efficiency would help developers ship more efficient applications without having too much hassle profiling energy consumption.

Energy efficiency can also be seen as a performance problem. It is sort of expected that a faster execution consumes less energy. However, this is not always true [4, 5, 6, 7]. For instance, mobile architectures have different types of Central Processing Units (CPU): fast but power-hungry CPUs are used for heavier tasks, and slow CPUs that consume less energy are used for simpler tasks (e.g., heterogeneous architecture *big.LITTLE*[2] used in mobile devices) [8]. Another example is the tail power states of components in mobile devices. This phenomenon consists in components in a high power state up to several seconds after finishing a task [9].

Heavy graphics processing is a source of unnecessary energy consumption in Android applications, according to a study that analyzed the *StackOverflow* community [10]. User interface (UI) is prone to inefficiencies that developers might not be aware of. A common problem is illustrated in Figure 1. UI views are described using several nested layouts that will be drawn on top of each other. If each of these layouts has a background color, the same pixel has to be drawn several times on each refresh. This code smell is known as *Overdraw*. Since views are designed using XML notation, this anti-pattern can hardly be spotted by analyzing the execution trace or the Java source code. Thus, in this study we have analyzed anti-patterns in both Java and XML sources.

We have measured the impact of eight Android best practices in terms of energy consumption in real, mature Android applications. Aiming at providing a methodology to help developing energy efficient Android applications, this work answers the following research questions:

- **RQ1**: Can programming practices be blindly applied in order to improve energy efficiency in an Android

[1]Dscout's 2016 Mobile Touches Report (visited in March 3, 2017): https://pages.dscout.com/mobile-touches-download-form

[2]http://www.arm.com/products/processors/technologies/biglittleprocessing.php (visited in March 3, 2017).

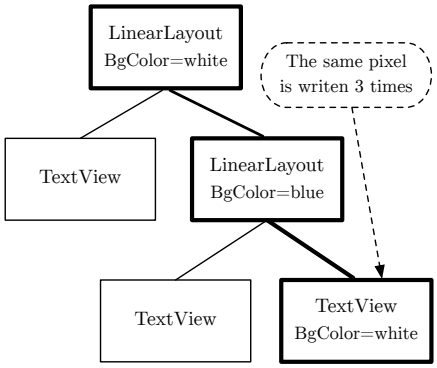

Fig. 1: Example of a tree with the hierarchy of UI components in an Android app. Pixels in the white *TextView* had to be painted with white and blue and finally white.

application?

- **RQ2**: Do best practices for performance improvement also improve energy efficiency?
- **RQ3**: Do these best practices actually have an impact on real, mature Android applications?

The main contributions of this work are: 1) to provide a set of best practices to develop energy efficient mobile applications; 2) to study and discuss the impact of each practice on energy consumption; and 3) pave the way for a toolset to automatically detect and refactor energy inefficiencies in mobile applications. In particular, the takeaway message of this paper is

> Fixing anti-patterns, viz. *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam* and *Recycle*, lead to more energy efficient mobile application, saving up to an hour of battery life.

As a side contribution of this paper, and to foster reproducibility, a benchmark suite containing all subjects and test suites used in our experiments is freely available from http://www.github.com/luiscruz/greenbenchmark.

II. Related Work

Energy profiling in Android software is a challenging task that has been addressed in many different ways. Some works measure energy consumption using software tools such as *PowerTutor* [11], *vLens* [12], *eProf* [13], or *eCalc* [14]. They provide an estimation using a power model based on previously collected data. This approach has been widely used to evaluate energy efficiency in mobile applications [15, 16, 17, 18, 19]. Estimation-based approaches are very handy since they do not require a special hardware setup. However, they are only compatible with specific smartphone models and Android versions, and the measurement validation is very difficult. In fact, the Android framework provides a tool to collect battery data from the device — *Batterystats* tool — but *Google* strongly discourages using this data for experimental studies[3].

In other works [11, 16, 20, 21], energy consumption has been measured by using hardware power measuring tools such as *Monsoon*[4], or even custom made tools that measure power directly in the phone's battery connectors. Since this approach can provide measurements with a higher accuracy, measurements in this work were performed by using a mobile device with power sensors. Previous work showed that there is a large set of code smells that are appearing in Android applications [22, 23]. The analysis was performed by checking the application's source code, bytecode, and metadata but no refactor was performed, and the impact on energy consumption was not studied.

Energy efficiency might be improved by offloading heavy tasks to the cloud [17, 24, 25]. Network components often lead to high energy consumption but depending on the complexity of the task, there is a tradeoff between CPU and network operations. Making such optimization would require structural changes in applications. In this paper we do not follow the same principles and network operations were not studied.

Previous work identified common energy-greedy sequences of Android Application Programming Interface (API) calls in 55 mobile applications [20]. Most energy-greedy API calls were found to be related to UI manipulation and database tasks. We take another step further by identifying alternative sequences that might lead to less energy consumption.

Other approaches have used visualization tools to help developers spot high energy-consuming I/O events, guiding developer to fix those events to reduce energy consumption [9]. Debugging energy-related defects has also been done by relating user reported defects with common energy-inefficient API calls patterns using log files [26]. Net, Camera, Database, and UI operations were considered the most energy-intensive components in 405 real-world mobile applications [18]. Our approach lies on finding common inefficient patterns and providing standard optimizations. Ideally this can help developing energy efficient applications before having to profile energy bundles.

Some works suggest changing the application's feature set (e.g., reducing third-party advertisements, different UI colors in Organic Light-Emitting Diode (OLED) displays [27], etc.) as a solution to optimize energy consumption [9, 13, 28, 29]. Such optimizations have to be considered when designing an energy efficient application. Nevertheless, our work intends to preserve original functionality of applications.

Energy efficiency has been improved at UI level by removing unnecessary display updates [30]. This approach is slightly different from ours since it requires modifications at the operative system level.

The idea of having energy code smells, plus refactoring fixes, for mobile applications has been studied before. Directed graphs were used to describe and analyze applications at the

[3]https://developer.android.com/studio/profile/battery-historian-charts.html (visited in March 3, 2017).

[4]https://www.msoon.com/LabEquipment/PowerMonitor/ (visited in March 3, 2017).

source code level, and a graph repository query language was proposed to detect code smells [31]. However, the effect of these code smells on energy consumption was not evaluated. Our work presents results of the optimization of mobile application along with statistical significance tests and effect sizes. Previous work reported that, depending on the application, code smells could have opposite impacts on energy consumption [32]. However, as opposed to the work presented in this paper, the study did not include mobile applications. Other work has used a similar approach to measure the impact of performance tips in Android applications [7]. However, these tips focus on internal aspects of the way Java is assembled in Android, which is expected to have impact in heavy processing tasks rather than in normal android applications. Our work takes it to another level and focuses on the way applications consume the Android framework's APIs.

Previous works have studied the influence of the pattern *Internal Getter/Setter* [7, 33, 34, 35, 36, 37] but it has been reported by *Google* as not having any effect in performance since Android 2.3[5][6]. Thus, we considered this optimization obsolete and left it out of our study.

Making an efficient use of the resources of a mobile device (e.g., GPS, Camera, Wifi) is a good way of saving energy. The use of these resources can be optimized by creating an event-flow graph (EFG) that represents UI states of a given Android app [38]. Defects are detected and refactored by matching expressions with a deterministic finite automaton based on the EFG. Although our work has similar goals, we have focused instead on code smells that directly affect CPU usage, including UI optimizations. In addition, significance tests were performed to consolidate the relevance of our results.

III. Empirical Study

We conducted a study in which energy consumption (measured in Joules (J)) was our dependent variable, while Android optimizations were the independent variable. In this section we describe our methodology and empirical study:

A. Android application selection
B. Static analysis and refactoring
C. Generation of automatic UI tests
D. Energy measurement tools setup
E. Experiments execution
F. Data analysis

A. Android Application Selection

The technique proposed in this paper, hence the empirical evaluation, needs the source code of the applications under analysis. For that matter, we use *F-droid*, a free and open source software catalog of Android applications. Currently, *F-droid* offers over $2,300$ open source applications[7]. The *Google Play Store* was used to obtain further details about the popularity of the application in the Android community. Applications were selected according to the following criteria: 1) open source, 2) active development, 3) not using heavy network operations, since we are not optimizing them. Applications were randomly selected and filtered when no performance issue was found by *lint*. Given the complexity of the experiments for each application we have limited our study to six applications:

Loop - Habit Tracker An application to track habits. Users' rating for version 1.4.1 is 4.7 out of 5.

Writeily Pro A note editor with markdown support. Users' rating for version 1.3.2 is 4.3 out of 5.

Talalarmo Alarm Clock A minimalist alarm clock. Users' rating for version 3.9 is 4.4 out of 5.

GnuCash A finance application to keep track of personal expenses. Rating for version 2.0.7 is 4.3.

Acrylic Paint A basic drawing application. This application is only available through the *F-droid* application store.

Simple Gallery Application to view pictures stored in the mobile device. Rating for version 1.15 is 4.7.

Table I shows information regarding the complexity of the applications as well as statistics from Google Play store. It presents number of installs, rating, and number of users that rated the application at Google Play Store, as well as lines of code (LOC) in Java, LOC in XML, number of classes, and McCabe's cyclomatic complexity (CC). *Loop - Habit Tracker* and *GnuCash* are the most complex applications, with over $25,000$ LOC. Besides having a large number of LOC in Java, *GnuCash* also has a large number of LOC in XML, more than $20,000$, which in Android is used for specification of the UI and other resources. It also has the highest CC value, $2,846$. *Talalarmo Alarm Clock* is the simplest application with approximately $1,000$ lines of Java code, $1,043$ of XML, and a CC of 131. This was expected since it provides a small set of features.

B. Static Analysis and Refactoring

In order to measure the impact of performance-based guidelines in Android applications it is necessary to systematically detect parts of code that did not comply with those guidelines. We have performed static code analysis to automatically detect code smells, such as *Overdraw*, mentioned in Section I. The Android Software Development Kit (SDK) provides a tool for this purpose, *lint*[8], which detects problems related with the structural quality of the code.

Code smells were chosen by considering performance-related suggestions given by *lint* that (1) are common in Android applications, and (2) potentially modify important parts of the application to fix the problem. Eight patterns resulted from this selection. Below we detail the eight patterns, including a rough estimation of priority provided by *lint* documentation. In *lint*, priority is an integer between 1 and 10, with 10 being the most important — this is merely used to sort issues relative to each other.

[5]http://stackoverflow.com/q/4912695/ (visited in March 3, 2017)

[6]http://tools.android.com/tips/lint-checks (visited in March 3, 2017)

[7]https://f-droid.org/wiki/page/Repository_Maintenance visited in March 3, 2017

[8]http://developer.android.com/tools/debugging/improving-w-lint.html visited in March 3, 2017.

TABLE I: Metrics of applications used in experiments.

Application	Installs (May 2016)	Rating	# ratings	LOC (Java)	LOC (XML)	Classes	CC
Loop - Habit Tracker	50,000–100,000	4.7	1,252	28,295	7,302	193	2,471
Writeily Pro	1,000–5,000	4.3	84	3,251	2,612	86	498
Talalarmo Alarm Clock	1,000–5,000	4.4	63	1,043	192	26	131
GnuCash	50,000–100,000	4.3	2,460	26,532	20,757	286	2,846
Acrylic Paint	n.a.	n.a.	n.a.	961	384	18	119
Simple Gallery	1,000–5,000	4.7	18	2,227	685	37	434

DrawAllocation: Allocations within drawing code It is a bad practice allocating objects during a drawing or layout operation. Allocating objects can cause garbage collection operations that will slow down the operation and create a nonsmooth UI. The recommended fix is allocating objects upfront and reusing them for each drawing operation. Lint priority: ||||||||| 9/10.

WakeLock: Incorrect wake lock usage Wake locks are mechanisms to control the power state of the mobile device. This can be used to wake up the screen or the CPU when the device is in a sleep state in order to perform tasks. If an application fails to release a wake lock or uses it without being strictly necessary, it can drain the battery. As an example, some applications use a wake lock to keep the screen on. This requires developers to properly release the wake lock when it is no longer necessary. Alternatively, the application can set the flag `FLAG_KEEP_SCREEN_ON` and the system will properly manage the wake lock, being less prone to errors. Lint priority: ||||||||| 9/10.

Recycle: Missing recycle() calls There are collections such as `TypedArray` that are implemented using singleton resources. Thus, they should be released so that calls to different `TypedArray` objects can efficiently use these same resources. Lint priority: ||||||||| 7/10.

ObsoleteLayoutParam: Obsolete layout params During development, UI views might be refactored several times. In this process, some parameters might be left unchanged even when they have no effect on the view. This causes useless attribute processing at runtime. Lint priority: ||||||||| 6/10.

ViewHolder: View Holder Candidates This pattern is used to make a smoother scroll in *List Views*. When in a *List View*, the system has to draw each item. To make this process more efficient, data from the previous drawn item can be reused. The number of calls to the method `findViewById`, which is known for being a very expensive method, decreases with this technique. Lint priority: ||||||||| 5/10.

Overdraw: Painting regions more than once Another common inefficiency in Android applications is when views are being overdrawn. This means that the same pixel has to be written several times, leading to unnecessary processing (see Figure 1). This can be improved by removing the background of views, or by clipping drawing, when possible. The recommended fix is adding a statement in the view creation that removes the background of the parent view: `getWindow().setBackgroundDrawable(null);`. Lint priority: ||||||||| 3/10.

UnusedResources Resources, such as icons or UI elements, may become obsolete due to changes in the software. However, developers may forget to remove them from the project which makes applications larger, consequently slowing down builds. Lint priority: ||||||||| 3/10.

UselessParent: Useless parent layout Since interface layouts suffer several changes throughout the development process, layouts frequently become useless. The latter can eventually be replaced by a descendant layout. Lint priority: ||||||||| 2/10.

Static code analysis is performed to automatically detect these patterns in the applications. For each detected pattern, the application was manually refactored and a new version of the application was produced. In addition, a version complying with all the practices was also created. For the sake of comparison, the original version was also used during the experiments. Table II shows the anti-patterns that were found in each of the analyzed applications. *Writeily Pro* resulted in five new versions, *Simple Gallery* in one version, and the others in three versions.

C. Generation of Automatic UI tests

The energy consumption of a mobile phone while running an application depends on several conditions (e.g., services that are running in the device, background tasks). To obtain meaningful results, the same execution needs to be replicated several times. The applications used in our study, unfortunately, do not provide test suites that mimic user interaction with the app. Thus, automatic UI test scripts were manually created to replicate user interaction in these applications.

The scripts were built using the *Python* library *Android View Client*[9]. This library allows the interaction with UI components by querying a view id, description or content, which makes tests compatible across different devices.

Tests mimic the usual interaction of a user. Algorithms 1 to 6 describe the interaction for the applications *Loop - Habit Tracker*, *Writeily Pro*, *Talalarmo*, *GnuCash*, *Acrylic Paint*, *Simple Gallery*, respectively.

For each execution of the test, the application was uninstalled and installed with the Android Application Package (APK) of the version under analysis. Thus, all user data was erased at the beginning of the experiment, making sure each execution of the test would have a similar initial state. On the other hand, cleaning user data requires the application to setup

[9]https://github.com/dtmilano/AndroidViewClient visited in March 3, 2017.

TABLE II: Anti-patterns found in open source applications.

Anti-Pattern	Loop - Habit Tracker	Writeily Pro	Talalarmo	GnuCash	Acrylic Paint	Simple Gallery
DrawAllocation	—	—	2 8+ 3-	—	3 7+ 2-	—
WakeLock	—	—	1 11+ 4-	—	—	—
Recycle	—	—	—	1 1+	—	—
ObsoleteLayoutParam	—	—	—	2 2-	—	—
ViewHolder	—	1 37+ 21-	—	—	—	—
Overdraw	5 5+ 2-	3 7+ 8-	—	—	3 10+ 7-	4 6+ 3-
UnusedResources	3 231-	67 1+ 318-	—	—	—	—
UselessParent	—	2 3+ 14-	—	—	—	—

Each refactoring is reported with the number of files changed ( ), number of insertions (+), and number of deletions (-). A dash (—) is present when a given anti pattern was not found in the application.

Algorithm 1 *Loop - Habit Tracker* interaction script

```
SkipIntroductoryTips()
for i ← 1 to 10 do
    for i ← 1 to 7 do
        CreateNewHabit(i)
        CheckHabitDetails(i)
        ScrollThroughTheReport()
        GoBack()
    end for
    DeleteAllHabits()
end for
```

Algorithm 2 *Writeily Pro* interaction script

```
GoToSettings()
GoBack()
for i ← 1 to 20 do
    folderOne ← CreateFolderWithFoldersInside()
    folderTwo ← CreateFolderWithNotesInside()
    MoveAllNotesToFirstFolder()
    CreateNote()
    folderThree ← CreateFolder()
    MoveItemToFolder(folderOne, folderThree)
    DeleteFolder(folderThree)          ▷ Removes all files
end for
```

in every experiment. This initial setup is not a real use case scenario, since it would happen only once after installing the app. To ensure that such scenario does not have a significant impact on results, we repeat subsequent scenarios a reasonable number of times — between 10 and 200 — depending on the complexity of the interaction.

Algorithm 3 *Talalarmo* interaction script

```
SetAlarmOn()                           ▷ Starts next minute tick
Sleep(5.minutes)
StopAlarm()
for i ← 1 to 200 do
    SwitchAMAndPM()
end for
for i ← 1 to 12 do
    SetAlarmOn()
    SetAlarmOff()
    SwitchAMAndPM()
    GoToSettings()
    SwitchBetweenDarkAndLightTheme()
    GoBack()
end for
```

Algorithm 4 *GnuCash* interaction script

```
SkipIntroductionSteps()
for i ← 1 to 10 do
    for all account ∈ {"Assets","Equity"} do
        for i ← 1 to 20 do
            SelectAccount(account)
            EditAccount()
            GoBack()
        end for
    end for
end for
```

Algorithm 5 *Acrylic Paint* interaction script

```
SkipIntroduction()
for i ← 1 to 20 do
    for i ← 1 to 10 do
        DrawLine()
    end for
    GoToColorMenu()
    for i ← 1 to 10 do
        SetColor()
    end for
    GoBack()
end for
```

D. Energy measurement tools setup

To measure the energy consumed in each execution, we use the bare-board computer *ODROID-XU* running Android version 4.2.2 - API level 17.

This device is known for having an architecture similar to a smartphone. Components such as cellular, location, accelerometer, and screen can be separately integrated. Nevertheless, these components are not being evaluated since the provided power sensors only report data for the main CPU, the secondary CPU, memory, and Graphics Processing Unit (GPU).

Power sensors provide data with a sample period of 263808 microseconds. Since the clock provided by *ODROID* in this setup has a precision of one second, data was down sampled. I.e., different samples with the same timestamp were aggre-

Algorithm 6 *Simple Gallery* interaction script

```
for i ← 1 to 100 do
    SelectAlbum()
    SelectPicture()
    GoBack()
    GoBack()
end for
```

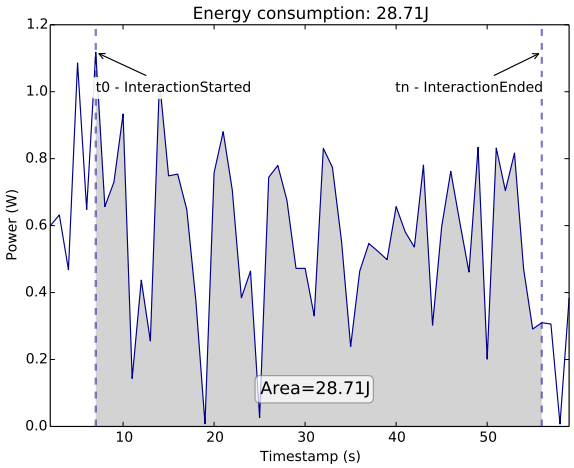

Fig. 2: Energy consumption calculation.

gated using the average to a period of one second.

The total energy consumption (i.e., work performed) between timestamps t_0 and t_n is calculated by integrating power over time:

$$W = \int_{t_0}^{t_n} P(t)dt \quad (1)$$

where P is power and W is the energy consumption (i.e., work). In this case, numerical integration was calculated based on the general Trapezoid Rule:

$$\int_{t_0}^{t_n} P(t)dt \approx \frac{\Delta t}{2}[P(t_0) + 2P(t_1) + 2P(t_2) + ... + 2P(t_{n-1}) + P(t_n)] \quad (2)$$

This calculation is illustrated in Figure 2, generated with data extracted from experiments with *Writeily Pro*. The energy consumed during an experiment is given by the area of the function of Power between the timestamp when the interaction started and the timestamp when interaction ended. In this case, energy consumption was 28.71J.

E. Experiments Execution

Each experiment was designed to be independent of previous experiments. The execution of a single experiment is illustrated by Figure 3. In every experiment, before running the UI interaction script, the energy logger is uploaded to the *ODROID* and set ready to start. The application, if existing, is uninstalled, the given APK is installed and finally the application is automatically opened.

After the execution of the UI interaction script, the energy logger is stopped and the data is collected from *ODROID* storage. This process is repeated 30 times for each different version of the application.

In addition, we measured interaction scripts with a blank application that we developed. The application does nothing and aims to give an idea of the energy consumed with the same UI interaction when the application is in an idle state. This gives an approximate measure of the overhead of energy consumed by the experiment setup and by the Android framework.

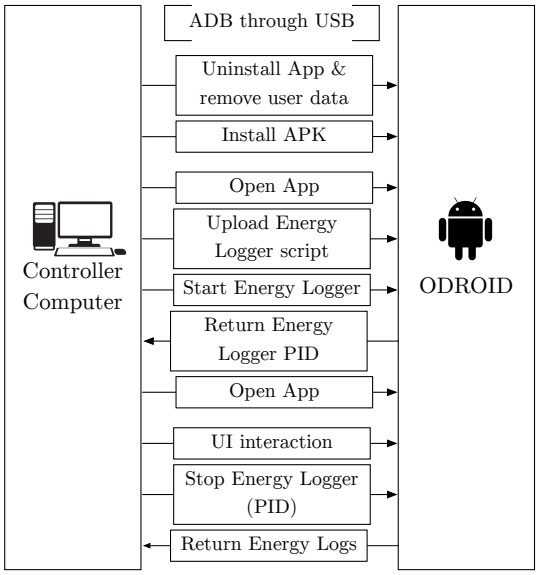

Fig. 3: Experiments' workflow.

F. Data Analysis

a) Downsampling: As described in Section III-D, data was downsampled to one sample per second in order to synchronize energy logs with *ODROID* timestamps.

b) Outlier Removal: Execution of experiments is prone to failures. This can happen due to a system dialog that popped up during the experiment, or due to a nontrivial bug that stopped the application, or to a slower response of the application that was not expected by the test script. Thus, there are executions that consumed considerably more or less energy. In order to reduce the effect of outliers, experiments with energy consumption outside the range $[\bar{x} - 2s, \bar{x} + 2s]$, where $\bar{x}$ is the sample mean and s is the sample standard deviation, were discarded.

IV. Results

In the end, 18 different APKs were tested with a total of 900 executions. It took 94 hours (roughly 4 days) and 75MB of raw data was collected.

For each app, Table III presents the sample size (n), i.e., the number of executions of the interaction script after outlier removal, mean ($\bar{x}$) and standard deviation (s) of energy consumption, and the p-value for the Shapiro-Wilk test. Shapiro-Wilk test for normality is a statistical test for detecting if the experiments follow a normal distribution. Column *Pattern* expresses the code smell that is fixed in that particular fixed version. *Original* is a version of the application that was not modified, serving as baseline. *All* stands for a version of the application in which all code smells were fixed. Results for the executions using the blank application, described in Section III-E, are also shown.

Figures 4 to 9 plot the results for each of the tested applications. As the violin plots show a bell-shaped curve and Shapiro-Wilk test's p-value is greater than 0.05, we conclude that data follows a normal distribution.

TABLE III: Descriptive statistics of experiments

Application	Pattern	n	$\bar{x}$ (J)	s	p-value
Loop - Habit Tracker	Original	28	335.6	35.3	0.76
	Overdraw	29	340.8	34.4	0.34
	UnusedResources	30	343.1	32.9	0.17
	All	29	336.4	34.7	0.58
	Blank	29	86.7	1.5	0.31
Writeily Pro	Original	30	119.7	7.2	0.42
	Overdraw	30	119.8	6.4	0.63
	UnusedResources	30	119.7	6.4	0.43
	ViewHolder	30	114.3	6.7	0.16
	UselessParent	30	119.3	7.1	0.10
	All	30	114.2	7.2	0.06
	Blank	30	87.2	9.9	0.20
Talalarmo	Original	29	58.2	0.76	0.28
	DrawAllocation	28	57.3	0.79	0.29
	WakeLock	28	57.35	0.69	0.59
	All	29	57.7	0.93	0.53
	Blank	29	41.8	0.46	0.78
GnuCash	Original	30	195.6	2.31	0.40
	ObsoleteLayoutParam	29	194.1	2.48	0.89
	Recycle	28	194.3	1.44	0.32
	All	30	194.0	2.48	0.89
	Blank	29	71.4	0.82	0.56
Acrylic Paint	Original	30	62.7	1.01	0.68
	DrawAllocation	29	62.5	1.06	0.12
	Overdraw	28	64.1	0.68	0.36
	All	29	64.0	0.77	0.85
	Blank	28	52.9	0.56	0.71
Simple Gallery	Original	30	145.9	3.67	0.28
	Overdraw	29	149.0	1.92	0.88
	Blank	29	45.1	0.61	0.71

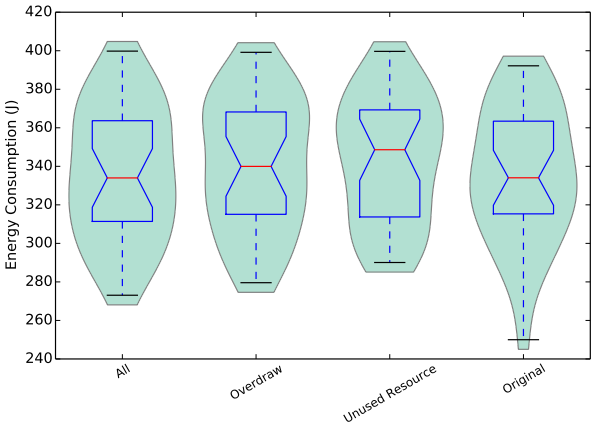

Fig. 4: Energy consumption for *Loop - Habit Tracker*.

To validate changes in energy consumption, we tested the following hypotheses:

$$H_0 : \mu_{fixed} - \mu_{original} = 0$$

$$H_1 : \mu_{fixed} - \mu_{original} \neq 0$$

where $\mu_{original}$ stands for the mean of the energy consumption for the original version, and μ_{fixed} of the fixed versions. Considering the samples as independent and since we have nonpaired data in which the standard deviation of populations is not known, we used Welch's two-sample t-test as the most appropriate test for our analysis. A two-tail p-value was used with a p-critical value (α) of 0.05. Results are shown in Table IV.

Table V presents the effect size results for the patterns with significant impact on energy. It presents the mean difference (MD) ($\bar{x}_{fixed} - \bar{x}_{original}$), Cohen's d ($\frac{\bar{x}_{fixed} - \bar{x}_{original}}{s}$), a method to indicate a standardized difference between means,

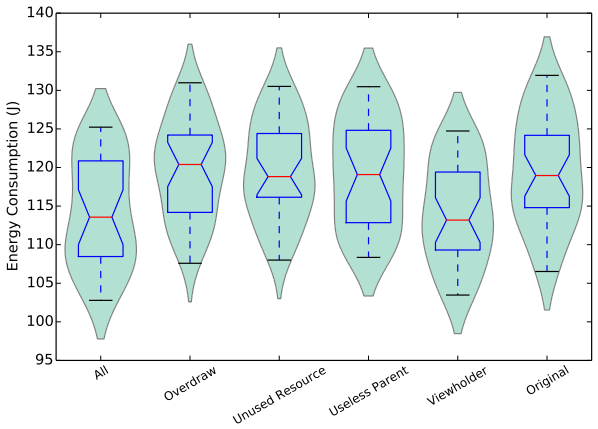

Fig. 5: Energy consumption for *Writeily Pro*.

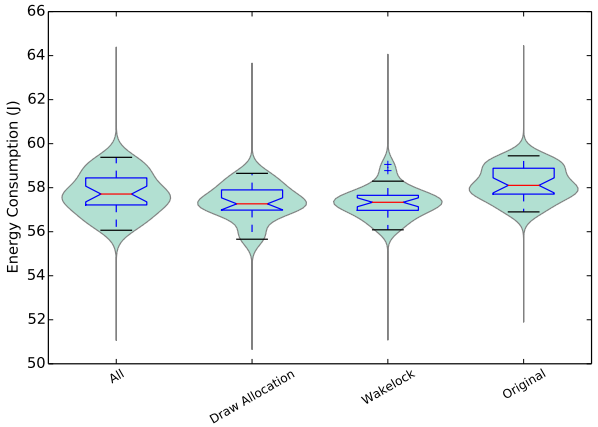

Fig. 6: Energy consumption for *Talalarmo*.

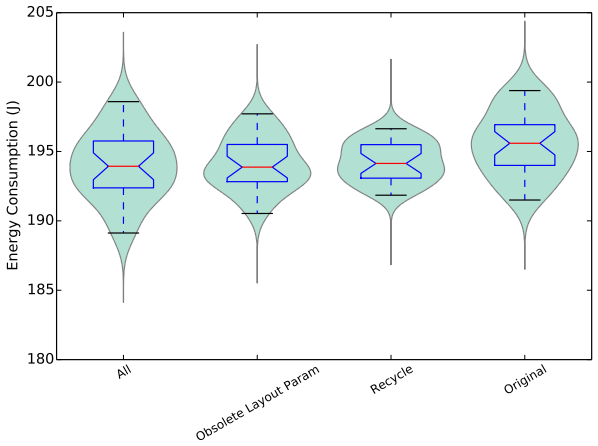

Fig. 7: Energy consumption for *GnuCash*.

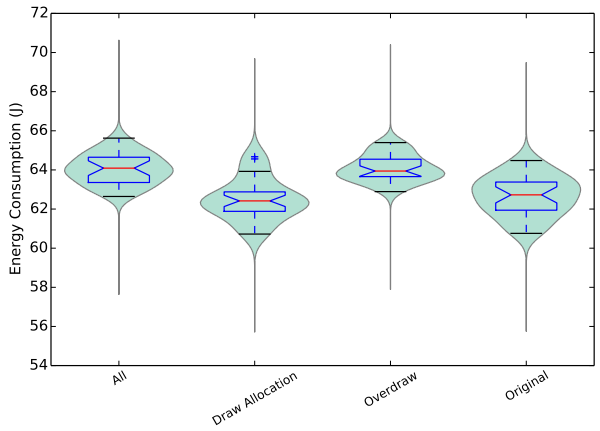

Fig. 8: Energy consumption for *Acrylic Paint*.

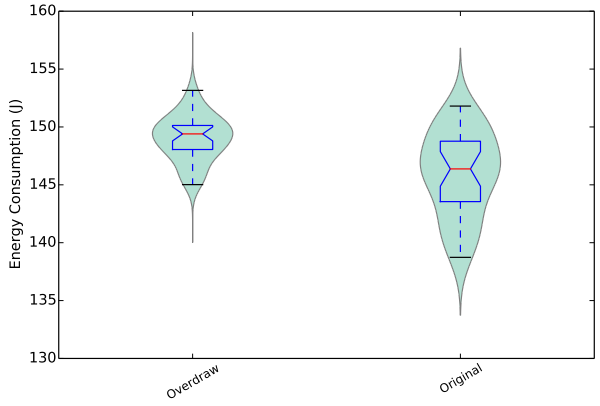

Fig. 9: Energy consumption for *Simple Gallery*.

TABLE IV: Significance Welch's t-test results

Application	Pattern	Test	p-value
Loop - Habit Tracker	Overdraw	-0.56	.5784
	UnusedResources	-0.83	.4121
	All	-0.08	.9362
Writeily Pro	Overdraw	-0.10	.9180
	UnusedResources	-0.03	.9790
	ViewHolder	3.02	.0038
	UselessParent	0.20	.8434
	All	2.93	.0049
Talalarmo	DrawAllocation	4.18	.0001
	WakeLock	4.43	$<$ .0001
	All	2.16	.0353
GnuCash	ObsoleteLayoutParam	2.57	.0127
	Recycle	2.55	.0140
	All	2.47	.0164
AcrylicPaint	DrawAllocation	0.64	.5221
	Overdraw	45.88	$<$ .0001
	All	-5.84	$<$ .0001
Simple Gallery	Overdraw	-4.04	.0010

improvement (IMP) compared to the original consumption ($\frac{\bar{x}_{fixed}-\bar{x}_{original}}{\bar{x}_{original}}$), and the column *Savings*, which provides the number of minutes of battery life saved after repeating the same usage of the application during 24 hours.

For example, the application *GnuCash* in Table III presents 5 rows, each for a different tested version of the application and for the blank application. The fixed version for the *Recycle* pattern has 28 experiments (n) which on average ($\bar{x}$) consumed 194.3J with a standard deviation (s) of 1.44 and a p-value for the normality test of 0.32. Significance tests presented in Table IV, show that this version of *GnuCash* can significantly reduce energy consumption, since the p-value obtained with the Welch's t-test is 0.0140 which is lower than our significance level $\alpha = 0.05$. All fixed versions that passed the significance level were reported in Table V. The table shows that this version of the application provided a MD of -1.28J, which means that it saved 1.28J, providing an improvement of 0.65% over the original version. This means that after 24 hours of using the app, the battery could last approximately 9 more minutes. The same analysis can be made with the other applications.

V. Discussion

Results show that *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam*, and *Recycle* are patterns that need to be taken into account to develop an energy efficient mobile application. *ViewHolder* is the pattern with the greatest impact, with an improvement of approximately 5% in *Writeily Pro*. The original version consumed 119.7J while it consumed 114.3J after being modified. This translates into 65 minutes of savings after 1 day of usage (see Table V). When considering a usage of 3.75 hours, this would translate in extra 10 minutes, without affecting user experience.

DrawAllocation also provides an interesting improvement. Although it occurred in a tiny part of the user interaction in *Talalarmo*, we observed an improvement of 1%. *DrawAllocation* was also tested with *Acrylic Paint* but it did not have a statistically significant improvement. The fix affected the color picker redraw routine. Using the Android developer options to debug view updates, we can see that redraw is only happening a single time when a new color is chosen. The impact of this fix in the overall execution was minimal, which might have been the reason for not having significant changes in energy consumption.

Fixing incorrect *WakeLock* usage also provided an improvement of 1%. In the original version of *Talalarmo*, the wake lock was not being properly released which could have lead to energy drain in particular cases. For instance, when the application is no longer in the alarm mode and the wake lock was not properly released, the device cannot activate a lower power state. This would consume energy unnecessarily but, given the nature of our tests, such a scenario is not being effectively tested. Thus, the effect size is expected to be higher in a real case scenario.

ObsoleteLayoutParam and *Recycle* anti-patterns were found in the application *GnuCash*. Although results showed a small effect size, with improvements of less than 1% (see Table V), they were statistically significant, as shown in Table IV. After analyzing the changes made to fix *ObsoleteLayoutParam*, we saw that it only required removing two obsolete view attributes. One in a list view and another in a list item. Thus, a big effect was not expected from this optimization. Still, for a more energy efficient practice, this issue should be considered.

TABLE V: Effect size of significant patterns

Application	Pattern		MD	Cohen's d	IMP (%)	Savings (min)
Writeily Pro	ViewHolder	↓	-5.39	-0.78	4.50	65
	All	↓	-5.42	-0.76	4.53	65
Talalarmo	DrawAllocation	↓	-0.86	-1.11	1.47	21
	WakeLock	↓	-0.85	-1.17	1.46	21
	All	↓	-0.48	-0.57	0.82	12
GnuCash	ObsoleteLayoutParam	↓	-1.41	-0.67	0.72	10
	Recycle	↓	-1.28	-0.66	0.65	9
	All	↓	-1.53	-0.64	0.78	11
Acrylic Paint	Overdraw	↑	1.42	1.64	-2.26	-33
	All	↑	1.37	1.51	-2.18	-31
Simple Gallery	Overdraw	↑	3.08	1.04	-2.11	-30

UI changes during the application development and obsolete attributes can be easily forgotten. This is a common issue, since it does not affect the UI appearance. Regarding *Recycle*, the issue occurred when accessing the color of an account. *GnuCash* allows the user to create several accounts. Each account can be customized with a different color. To get the account's color options a `TypedArray` needs to be accessed. The issue lay in the fact that `TypedArray` was not being closed after it had been accessed. This only happens when a user opens the settings of an account. Regardless, it was able to have significant impact on energy consumption, according to the results of the Welch's t-test in Table IV.

Surprisingly, after fixing *Overdraw*, applications ended up consuming more energy. Although *Overdraw* can create a laggy UI, fixing it can lead to more energy consumption. In the applications *Acrylic Paint* and *Simple Gallery*, it decreased battery life approximately 30 minutes after one day of usage. Having a simple UI layout hierarchy is always a good practice, but adding extra code to avoid *Overdraw* requires processing, which might not be worth it, depending on the scenario.

When the application has a view that remains active for a considerable amount of time, this view will have to redraw itself several times. In this case, having an efficient redraw is important, and fixing *Overdraw* is expected to create interesting results. On the other hand, if a view is being created several times but does not remain active for a reasonable amount of time, fixing *Overdraw* might be creating an unnecessary overhead during the creation of the view. Since modeling user behavior is not a trivial task, in our experiments the time a user spends in a view is not being considered. Thus, views with long lifetime were not explored.

UnusedResources and *UselessParent* did not have any significant effect, as showed by the Welch's t-test in Table IV. *UnusedResources* was tested in the applications *Loop - Habit Tracker* and *Writeily Pro*. Having unused resources in the application can increase build time, APK size, and complexity of project maintenance. Thus, it is still an anti-pattern to be considered, although it does not affect energy consumption. *UselessParent* was tested in the application *Writeily Pro*. Since the test is focusing in common use case scenarios rather than a particular anti-pattern, it is possible that *UselessParent* was not a relevant issue in this scenario. This means that optimization was not necessary in this particular case, but it can still be useful in other applications and scenarios.

It is interesting to note that in a few cases, improvements were higher after fixing a single anti-pattern than after fixing all of them (e.g., *Talalarmo*). The main reason for this lies in the fact that experiences are prone to random variations related with the power meter and the mobile device. Thus, results may change from experiment to experiment, and effect size measures are not very precise. Nevertheless, this does not affect statistical significance, which shows that energy consumption reduces after using these patterns.

The results for the blank application show that in some experiments a great part of energy is consumed in the experimental setup. Analyzing Table III, regarding the application *Acrylic Paint*, the interaction script running with the original version consumed on average 62.68J, whereas on the blank application consumed 52.93J on average. This means that the interaction script consumed 84% of the total energy consumption, leaving only 16% for optimization. The least affected application by the interaction script was *Loop - Habit Tracker*, consuming only 26% of the total energy consumption.

> **RQ1:** Can programming practices be blindly applied in order to improve energy efficiency in an Android application?

Optimizations analyzed in this work do not affect the feature set of the application. This means that they can be applied without having to deal with tradeoffs between user experience and energy consumption. Furthermore, the instrumentation made for the applications in this study did not require previous knowledge about the project. Therefore, energy consumption was reduced after applying patterns *ViewHolder* (4.5%), *DrawAllocation* (1.5%), *WakeLock* (1.5%), *ObsoleteLayoutParam* (0.7%), and *Recycle* (0.7%).

> **RQ2:** Do best practices for performance improvement also improve energy efficiency?

Our results show that there are performance optimizations that also have an impact on energy consumption. Fixing *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam*, and *Recycle* improved energy efficiency. Developers should consider performance anti-patterns when developing energy efficient applications. This does not conform with previous work [7] mainly because we have studied Android specific optimizations. However, *UnusedResources* and *Use-*

lessParent did not provide any significant change in energy consumption, while *Overdraw* was found to consume more energy (2.2%). Nevertheless, the impact of these patterns depends on the case in which it is being applied, as it was shown with the *Overdraw* pattern. Different applications and use cases might have better or worse results. Further research need to be made to identify optimal or worst-case scenarios for these patterns.

RQ3: Do these best practices actually have an impact on real, mature Android applications?

Six open source Android applications were included in this study. They are available on F-Droid and, with the exception of *Acrylic Paint*, they can also be downloaded from *Google Play Store*. From these six, we were able to improve energy efficiency in three applications. We observed improvements in *Writeily Pro* (4.5%), *Talalarmo* (1.4%), and *GnuCash* (0.8%). This evidence suggests that any application with patterns *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam*, and *Recycle* can have improvements in energy efficiency. More applications should however be evaluated to corroborate with this intuition. As said in the Introduction, the test cases used in the study are available online to foster reproducibility.

A. Threats to the Validity

The refactor performed and the validation of its changes is limited to our perception of the application's features and to the impact we expect from our code changes.

Results obtained from energy measurements can be affected by several factors that are not easily controlled. Different devices and different versions of Android may respond in a different way to these optimizations. Background tasks performed by other applications affect energy consumption and are hard to control. Although the performed outlier removal intends to discard these cases, some of them may still have impacted some experiments.

In addition, we did not measure energy consumption of the whole device. Other components such as GPS and accelerometer are known to have a significant impact on energy consumption. The same happens with network operations and screen usage. However, the later is expected to have the same energy consumption in all cases. The automatic interaction with the UI, used in our experiments, may also create overhead on the energy spent. This might affect results in terms of effect size but we do not expect it to affect statistic significance.

External factors, such as temperature can also affect experiment results. If a CPU has a higher temperature, it will consume more energy. However, if a CPU has to do more processing, it will increase its temperature, which means that temperature can also be an effect and not the cause. We have repeated 30 times the experiment for each fixed version in order to have a fairer comparison, although this can still be a threat. In addition, the experiments were alternatively executed with other versions of a given app to assure similar external factors for all versions.

Patterns were studied in a small set of applications. Since the impact of these patterns heavily depends upon the case in which it is being applied, further experiments should be conducted to understand overall effects. Nevertheless, this study is an initial exploration of the impact of anti-patterns in energy consumption.

VI. Conclusions and Future Work

As mobile applications operate in a very resource constrained environment, energy consumption is a concern that needs to be addressed while developing mobile applications. In this paper, we have studied whether or not eight Android, performance-related code smells also lead to considerable overhead on energy consumption.

The empirical evaluation reported in this paper, using six Android applications, shows that performance-based patterns/smells lead to more energy efficient mobile application, saving up to an hour of battery life. In particular, from the eight analyzed code smells, five were found to reduce energy consumption.

For energy efficiency, developers should take into account anti-patterns *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParam*, and *Recycle*. This paper suggests that a toolset to automatically refactor mobile applications to avoid these code smells would greatly improve energy efficiency of already deployed mobile applications. It would help developers since they would not need a deep understanding of these practices. Furthermore, it also paves the way to Integrated Development Environments (IDEs) to consider plugins to review code, warn and educate developers when energy inefficient constructs are being used.

Future work includes building an IDE-based toolset with two main features: one to help developers building *greener* applications, and another to label mobile applications with respect to energy efficiency (e.g., as in [39] for code quality, using stars: 1 star — very bad — 5 stars — very good). We also plan to study the impact of other anti-patterns.

VII. Acknowledgments

This work is financed by ERDF – European Regional Development Fund through the Operational Programme for Competitiveness and Internationalization - COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, and by National Funds through FCT – Foundation for Science and Technology as part of project UID/EEA/50014/2013. Luis Cruz is sponsored by an FCT scholarship grant number PD/BD/52237/2013. Furthermore, we would like to thank João Cardoso and the SPeCS research group for lending their ODROID device and Ana Lúcia Andrade, Rui Pereira, and José Campos for their valuable feedback on earlier versions of this paper.

References

[1] A. J. Oliner, A. P. Iyer, I. Stoica, E. Lagerspetz, and S. Tarkoma, "Carat: Collaborative energy diagnosis for mobile devices," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*. ACM, 2013, p. 10.

[2] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling asleep with angry birds, facebook and kindle: a large scale study on mobile application usage," in *Proceedings of the 13th international conference on Human computer interaction with mobile devices and services*. ACM, 2011, pp. 47–56.

[3] C. Pang, A. Hindle, B. Adams, and A. E. Hassan, "What do programmers know about software energy consumption?" *IEEE Software*, vol. 33, no. 3, pp. 83–89, May 2016. [Online]. Available: http://dx.doi.org/10.1109/MS.2015.83

[4] R. Pereira, M. Couto, J. Saraiva, J. Cunha, and J. P. Fernandes, "The influence of the java collection framework on overall energy consumption," in *Proceedings of the 5th International Workshop on Green and Sustainable Software*. ACM, 2016, pp. 15–21.

[5] G. Pinto, F. Castor, and Y. D. Liu, "Understanding energy behaviors of thread management constructs," in *ACM SIGPLAN Notices*, vol. 49, no. 10. ACM, 2014, pp. 345–360.

[6] A. E. Trefethen and J. Thiyagalingam, "Energy-aware software: Challenges, opportunities and strategies," *Journal of Computational Science*, vol. 4, no. 6, pp. 444–449, 2013.

[7] C. Sahin, L. Pollock, and J. Clause, "From benchmarks to real apps: Exploring the energy impacts of performance-directed changes," *Journal of Systems and Software*, vol. 117, pp. 307–316, 2016.

[8] K. Yu, D. Han, C. Youn, S. Hwang, and J. Lee, "Power-aware task scheduling for big. little mobile processor," in *SoC Design Conference (ISOCC), 2013 International*. IEEE, 2013, pp. 208–212.

[9] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 153–168.

[10] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 22–31.

[11] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2010, pp. 105–114.

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

[13] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012, pp. 29–42.

[14] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating android applications' cpu energy usage via bytecode profiling," in *Proceedings of the First International Workshop on Green and Sustainable Software*. IEEE Press, 2012, pp. 1–7.

[15] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang, "Adel: An automatic detector of energy leaks for smartphone applications," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2012, pp. 363–372.

[16] K. M. Saipullah, A. Anuar, N. A. Ismail, and Y. Soo, "Measuring power consumption for image processing on android smartphone," *American Journal of Applied Sciences*, vol. 9, no. 12, p. 2052, 2012.

[17] D. Li and W. G. Halfond, "Optimizing energy of http requests in android applications," in *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*. ACM, 2015, pp. 25–28.

[18] D. Li, S. Hao, J. Gui, and W. G. Halfond, "An empirical study of the energy consumption of android applications," in *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 121–130.

[19] M. Gottschalk, J. Jelschen, and A. Winter, "Saving energy on mobile devices by refactoring." in *EnviroInfo*, 2014, pp. 437–444.

[20] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Mining energy-greedy api usage patterns in android apps: an empirical study," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 2–11.

[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] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen, "Understanding code smells in android applications," in *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, 2016, pp. 225–234.

[23] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien, "Detecting antipatterns in android apps," in *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press, 2015, pp. 148–149.

[24] Y.-W. Kwon and E. Tilevich, "Facilitating the implementation of adaptive cloud offloading to improve the energy efficiency of mobile applications," in *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*. IEEE, 2015, pp. 94–104.

[25] H. Qian and D. Andresen, "Reducing mobile device energy consumption with computation offloading," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*. IEEE, 2015, pp. 1–8.

[26] A. Banerjee, H.-F. Guo, and A. Roychoudhury, "Debugging energy-efficiency related field failures in mobile apps," in *IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft*, vol. 16, 2016.

[27] X. Chen, Y. Chen, Z. Ma, and F. C. Fernandes, "How is energy consumed in smartphone display applications?" in *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*. ACM, 2013, p. 3.

[28] D. Li, A. H. Tran, and W. G. Halfond, "Nyx: A display energy optimizer for mobile web apps," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 958–961.

[29] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing energy consumption of guis in android apps: a multi-objective approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 143–154.

[30] D. Kim, N. Jung, Y. Chon, and H. Cha, "Content-centric energy management of mobile displays," *IEEE Transactions on Mobile Computing*, vol. 15, pp. 1925 – 1938, 2015.

[31] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, "Removing energy code smells with reengineering services." in *GI-Jahrestagung*, 2012, pp. 441–455.

[32] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?" in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2014, p. 36.

[33] D. Li and W. G. Halfond, "An investigation into energy-saving programming practices for android smartphone app development," in *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM, 2014, pp. 46–53.

[34] S. Mundody and K. Sudarshan, "Evaluating the impact of android best practices on energy consumption," in *IJCA Proceedings on International Conference on Information and Communication Technologies*, vol. 8, 2014, pp. 1–4.

[35] A. R. Tonini, L. M. Fischer, J. C. B. de Mattos, and L. B. de Brisolara, "Analysis and evaluation of the android best practices impact on the efficiency of mobile applications." in *SBESC*, 2013, pp. 157–158.

[36] G. Hecht, N. Moha, and R. Rouvoy, "An empirical study of the performance impacts of android code smells," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 59–69. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897100

[37] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, "Investigating the energy impact of android smells," in *24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, p. 10.

[38] A. Banerjee and A. Roychoudhury, "Automated refactoring of android apps to enhance energy-efficiency," in *Proceedings of the International Conference on Mobile Software Engineering and Systems*, ser. MOBILESoft '16. New York, NY, USA: ACM, 2016, pp. 139–150. [Online]. Available: http://doi.acm.org/10.1145/2897073.2897086

[39] R. Jabbarvand, A. Sadeghi, J. Garcia, S. Malek, and P. Ammann, "Ecodroid: An approach for energy-based ranking of android apps," in *Proceedings of the Fourth International Workshop on Green and Sustainable Software*. IEEE Press, 2015, pp. 8–14.