Investigating the Energy Impact of Android Smells

Antonin Carette[1], Mehdi Adel Ait Younes[1], Geoffrey Hecht[1,2], Naouel Moha[1], Romain Rouvoy[2,3]
[1] Université du Québec à Montréal, Canada
[2] University of Lille / Inria, France
[3] IUF, France
antonin.carette@gmail.com, ait_younes.mehdi_adel@courrier.uqam.ca, geoffrey.hecht@inria.fr,
moha.naouel@uqam.ca, romain.rouvoy@inria.fr

***Abstract*—Android code smells are bad implementation practices within Android applications (or apps) that may lead to poor software quality. These code smells are known to degrade the performance of apps and to have an impact on energy consumption. However, few studies have assessed the positive impact on energy consumption when correcting code smells. In this paper, we therefore propose a tooled and reproducible approach, called HOT-PEPPER, to automatically correct code smells and evaluate their impact on energy consumption. Currently, HOT-PEPPER is able to automatically correct three types of Android-specific code smells: Internal Getter/Setter, Member Ignoring Method, and HashMap Usage. HOT-PEPPER derives four versions of the apps by correcting each detected smell independently, and all of them at once. HOT-PEPPER is able to report on the energy consumption of each app version with a single user scenario test. Our empirical study on five open-source Android apps shows that correcting the three aforementioned Android code smells effectively and significantly reduces the energy consumption of apps. In particular, we observed a global reduction in energy consumption by 4,83% in one app when the three code smells are corrected. We also take advantage of the flexibility of HOT-PEPPER to investigate the impact of three picture smells (bad picture format, compression, and bitmap format) in sample apps. We observed that the usage of optimised JPG pictures with the Android default bitmap format is the most energy efficient combination in Android apps. We believe that developers can benefit from our approach and results to guide their refactoring, and thus improve the energy consumption of their mobile apps.**

***Keywords*—Android, energy consumption, code smells, picture**

I. INTRODUCTION

Mobile devices have known a huge success along the last five years, for example Android's sales increased by more than 500% since 2011 [8]. With more than 50% of devices sold worldwide, Android has become one of the most popular operating system [6]. As the number of devices has increased, the number of applications (or *apps*) also grew rapidly along the last years. Therefore, the number of mobile developers also increases. Apps are mostly written using popular *Object-Oriented* (or OO) programming languages like Java, Objective-C, Swift or C#. Yet, mobile development is not as similar as traditional software development [54] and developers must consider the mobile specificities. Also, the user demand keeps increasing and forces mobile developers to add new features and maintain their apps as quickly as possible. Unfortunately, this pressure leads developers to adopt bad implementation practices, also known as *code smells* [28]. Code smells can lead to cause resources leaks in CPU, memory, battery, etc [25]. Leaks may deteriorate the quality of the app in terms of stability, user experience, maintainability, etc. It is also important to note that more than 18% of Android apps exhibit code smells [43]. Our previous studies have investigated the impact of code smells on performance and concluded that the correction of code smells improves the app performance [34]. In particular, the major code smells that impact the performance of Android apps are *HashMap Usage* (HMU), *Internal Getter/Setter* (IGS), and *Member Ignoring Method* (MIM) [2], [20], [34].

Like performance, the battery lifespan or energy consumption of an app is a critical quality criteria [1]. D. Li and W. Halfond [39] have proven that two of the three performance code smells listed above also have an energy impact on fictive app. However, these experiments were not performed on a real user app.

In this paper, we therefore propose an automated approach, called HOT-PEPPER, supported by a framework for Android developers that allows them to assess and improve the energy consumption of their Android apps. Concretely, HOT-PEPPER enables developers to detect and correct code smells, and evaluate their impact in terms of energy consumption in Android apps. HOT-PEPPER relies on PAPRIKA, a static analysis tool dedicated to the detection and correction of code smells in Android apps.

For the impact evaluation of code smells, HOT-PEPPER relies on the tool NAGA-VIPER, which uses a physical measurement device and monitors energy-related metrics (execution time, intensity, and voltage) on Android apps. For the validation of HOT-PEPPER, we performed an empirical study that allow us to answer to the following two research questions:

RQ_1: *Does the correction of Android code smells improve the energy consumption of the mobile phone?*

Finding: Yes, the correction of Android code smells improves the energy consumption of the mobile phone. We observed that the correction of at least one code smell reduces the energy consumption of the mobile phone. Moreover, the correction of all code smells reduces the energy consumption even more significantly.

RQ_2: *Do picture smells have an impact on the energy consumption of the mobile phone?*

Finding: Yes, studied picture smells have a bad impact on the energy consumption of the mobile phone. We observed that

978-1-5090-5501-2/17/$31.00 © 2017 IEEE

using optimised JPG pictures and the default Android bitmap format reduces significantly the energy consumption of the mobile phone.

The main contributions of this paper are:

1) An automated approach, supported by a framework, to detect and correct Android code smells, and evaluate their impact in terms of energy consumption;
2) An empirical study on five open source Android apps and a specific custom app, which proves that six Android smells can have an impact on the energy consumption of the mobile phone.

This paper is organized as follows: Section II introduces the catalog of Android smells investigated in this paper, and the energy metrics used to evaluate the energy consumption. Section III describes the proposed approach, HOT-PEPPER, along with its supporting tools, PAPRIKA and NAGA VIPER. Section IV details the experimental study performed to analyse the energy consumption of six Android smells in five open-source Android apps and a specific custom app. Section V reports the results of this study. Section VI discusses existing approaches that deal with the correction of code smells and automated approaches to analyse and evaluate the energy consumption, both in mobile apps. Finally, Section VII summarises our work and outlines future work.

II. BACKGROUND

In this section, we introduce a catalog of Android smells that we investigated in this work for the evaluation of the energy consumption. This catalog consists of six Android smells splitted in two categories: three performance code smells and three pictures smells. Then, we introduce the energy metrics that we use to estimate the energy consumption of those Android smells.

A. Studied Smells

We chose the following smells as they are commonly observed in Android apps [35] and are reported to have a negative impact on Android performance. This includes two types of smells: performance code smells and pictures smells. Two of these performance code smells, *Member Ignoring Method* (MIM) and *Internal Getter/Setter* (IGS), are already reported to have a significant impact on energy consumption in a fictive Android app [39]. Our approach aims to measure empirically the energy consumption of the studied smells in common Android apps by using a user-driven scenario.

1) Performance code smells:

HashMap Usage (HMU) : The Android framework promotes `ArrayMap` and `SimpleArrayMap` as replacements of the standard Java `HashMap`. They are supposed to be more memory-efficient and trigger less garbage collections with no significant difference on operations performance for maps containing up to hundreds of values [20]. Thus, unless a complex map for a large set of objects is required, the use of `ArrayMaps` should be preferred over the usage of `HashMap` for Android apps. Therefore, creating small `HashMap` instances can be considered as a code smell [20], [33]. However, a performance degradation of using a `HashMap` can occur when facing an unpredicted growth of the map. An example of HashMap Usage is provided in Listing 1.

Listing 1. Example of HashMap Usage in SoundWaves Podcast app

```
if (itemMap == null) {
   itemMap = new HashMap<>();
   for (int i = 0; i <
       ItemColumns.ALL_COLUMNS.length; i++)
     itemMap.put(ItemColumns.ALL_COLUMNS[i],
         ItemColumns.ALL_COLUMNS[i]);
}
```

Internal Getter/Setter (IGS) : IGS is an Android code smell that occurs when a field is accessed, within the declaring class, through a getter (`var = getField()`) and/or a setter (`setField(var)`). This indirect access to the field may decrease the performance of the app. The usage of IGS is a common practice in OO languages like C++, C# or Java because compilers or virtual machines can usually inline the access. However, there is only the simple inlining for Android [19] and, consequently, the usage of a trivial getter or setter is often converted into a virtual method call, which makes the operation at least three times slower than a direct access. This code smell can be corrected by accessing the field directly within a class (`var = this.myField`, `this.myField = var`) or declaring the getter and setter methods in the public interface. Correcting IGS with this refactoring is therefore a way to increase the performance of the method accessing a field [2], [25]. Of course, non-trivial getters/setters, as illustrated in Listing 2, are not concerned by this code smell. Therefore, a possible drawback of fixing an IGS can happen when a trivial getter/setter is modified into a non-trivial getter/setter in a future version of an app.

Listing 2. Example of non-trivial getter in SoundWaves Podcast app

```
public String getURL() {
  String itemURL = "";
  if (this.url != null && this.url.length() >
      1)
    itemURL = this.url;
  else if (this.resource != null &&
      this.resource.length() > 1)
    itemURL = this.resource;
  return itemURL;
}
```

Member Ignoring Method (MIM) : In Android, when a method does not access an object attribute or is not a constructor, it is recommended to use a static method in order to increase performance. The static method invocations are about 15%–20% faster than dynamic invocations [2]. It is also considered as a good practice for readability since it ensures that calling the method will not alter the object state [2], [25]. However, there is one possible side effect in terms of inheritance since all the extending classes have to declare or to refer to the same static methods. Listing 3 is an example of MIM.

Listing 3. Example of Member Ignoring Method in SoundWaves Podcast app

```
private boolean
    animationStartedLessThanOneSecondAgo(long
    lastDisplayed) {
  return System.currentTimeMillis() -
     lastDisplayed < 1000 && lastDisplayed !=
     -1;
}
```

2) Pictures Smells:

Picture Format : The usage of the PNG picture format is known as a bad practice when using large pictures. The JPG or GIF formats should rather be preferred in such cases [5] [13]. The GIF format is a lossless compression picture format limited to 256 colors. It is used on the web to animate plural pictures. JPG is a lossy compression picture format, which makes it a common choice for storing big pictures like photographs or realistic images. This picture format uses 16 bits per pixel. PNG is a lossless compression picture format, which makes it a common choice for pictures on the web. Two different formats of PNG can be found, PNG-8, which is similar to GIF, and PNG-24, which uses 24 bits per pixel. Although it is lossless and uses a greedier pixel compression format, PNG is a good choice for storing pictures with a small file size, and with few colors. Therefore, any developer has to wisely decide on the format of pictures to display within the app depending on the size of the picture, its number of colors and its usage (icon, photograph, etc.) [9], [10].

Picture Size : A common problem when storing pictures is to get the best trade-off between the size of the picture and its quality. The goal here is to reduce the file size without degrading a lot the quality. An existing solution is to downgrade the quality of a high-resolution picture without strongly impacting the user experience. Existing algorithms, such as PSNR [15], SSIM [53] and Butteraugli [3], are solutions to measure with accuracy the difference and similarity of two pictures. This calculation is related to the perceived difference of both same pictures, in different resolution, by the human eye. These solutions provide excellent ways to find the best trade-off between the compression and the quality of a picture. Using these algorithms and other techniques, it is possible to reduce the size of the file without degrading the user experience [16], [17].

Picture Bitmap Usage : Android bitmap objects can consume a lot of memory for large pictures, especially when they are displayed at the same time by Android views like ListView or GridView [14]. However, a common problem with Dalvik, the Android virtual machine used for Android versions until 5.1, is that it cannot defragment in real-time the memory for a single running app. Thus, for an app that displays only a GridView of high-resolution pictures, the virtual machine may not respond to the request of the app when asking to allocate or re-allocate memory to store a new rich picture. Due to this behaviour, the app becomes slower. Android developers have to consider how to load and cache effectively bitmaps, using for example bitmap formats like RGB_565, instead of ARGB_8888. The last bitmap format is the default and greedier one for the latest version of Android. The usage of expensive bitmap formats can be considered as a bad practice in situations where the picture displayed in the app is too small to notice a difference with a picture adopting a lower pixel density [21].

Listing 4. Example of the ARGB_8888 Compression Format Usage

```
Bitmap bitmap_image =
    BitmapFactory.decodeResource(getResources(),
    myPictureResource);
Bitmap mutable_bitmap_image =
    bitmap_image.copy(Bitmap.Config.ARGB_8888,
    true);
imageView.setImageBitmap(mutable_bitmap_image)
```

B. Energy Consumption Metrics

We propose to evaluate the energy consumption by measuring the average intensity, the average voltage of the device and the time between these two measures. The intensity and the voltage are the flow and the force of electricity through an electrical line, respectively. The energy consumption is expressed in Joules considering the intensity, the voltage and the time of the experiment. The intensity of the battery is limited, and the voltage is constant all long of the phone usage. Therefore, we estimate the energy consumption of an Android app as following:

$$E = \sum (V * \Delta t * I_{Cons}) \quad (1)$$

where E is the general energy consumption of the Android app (joules), V is the current voltage of the battery (volts), t is the execution time of the studied app (seconds) and I_{Cons} is the average intensity (mA) of the mobile app and the operating system.

III. Hot-Pepper: Improving the Energy Consumption of Android Apps

In this section, we introduce an automated approach, supported by a framework, called Hot-Pepper. This approach consists in the detection and correction of code smells, in order to evaluate their impact in terms of energy consumption on Android apps.

A. Overview

The Hot-Pepper approach is illustrated in Figure 1. It is supported by two main tools, Paprika and Naga Viper.
Hot-Pepper uses Paprika [36] to detect Android code smells. We also improved Paprika to automatically correct these smells. As a first step, Paprika takes as input the original app's APK to detect Android code smells. Next, this tool uses detected code smells, and the Android app source code, to generate several fixed APK versions of the app. Paprika generates as many fixed versions of the Android app as detected Android code smells, plus a version fixing all these code smells.
Hot-Pepper uses Naga Viper, a tool that we developed to evaluate the impact of each APK version of the Android app, including the original one, iteratively. As a first step,

NAGA VIPER takes as input all existing APK versions of the Android app, a scenario based on user events, and an Android smartphone plugged to a physical measuring device. This step consists in computing energy metrics for each APK. After this, NAGA VIPER compares these energy metrics to return the most energy-efficient APK, the associated source code, and the list of refactorings from the `Paprika`'s correction.
In the following, we describe the steps covered by PAPRIKA and NAGA VIPER.

B. PAPRIKA

In order to produce the most energy-efficient app, HOT-PEPPER uses PAPRIKA, a static tool analysis for Android apps. This analysis consists in detecting and correcting Android code smells.

Step 1: Detecting Code Smells

Input: The original Android app APK to analyse.
Output: All detected code smells in the app.
Description: First, HOT-PEPPER needs to analyse the APK, built from the developer, in order to retrieve Android code smells. For this purpose, we use the PAPRIKA analysis. In this step, PAPRIKA takes as input the Android APK and runs the analysis phase, which consists in detecting code smells in the app. As a result, PAPRIKA returns a list of code smells that contains specifically, for each code smell type, their proportion and their location in the app.
Implementation: To perform the analysis step, PAPRIKA needs to build a model of the Android app in order to retrieve 8 entities: `App`, `Class`, `Method`, `Attribute`, `Variable`, `ExternalClass`, `ExternalMethod`, and `ExternalArgument`. PAPRIKA is based on the SOOT framework and its DEXPLER module [24] to analyse the APK, in order to build the app's model. SOOT converts the Dalvik bytecode of mobile apps into an internal representation, composed by previous entities, that we translate as a NEO4J graph. Finally, PAPRIKA detects code smells by querying the NEO4J graph model using CYPHER (cf. Listing 5).

Listing 5. Cypher query to detect Internal Getter/Setter

```
MATCH (m1:Method)-[:CALLS]->(m2:Method),
(cl:Class)
WHERE (m2.is_setter OR m2.is_getter)
AND cl-[:CLASS_OWNS_METHOD]->m1
AND cl-[:CLASS_OWNS_METHOD]->m2
RETURN m1
```

Step 2: Correcting Code Smells

Input: Detected code smells in the Android app and the app source code.
Output: The corrected APK versions of the Android app, and the original one.
Description: For this step, PAPRIKA requires the Android source code of the app and the list of detected code smells. For each type of code smells, PAPRIKA generates a version of the app without those. Moreover, at the end, the tool generates a version without any detected code smells. At the end, PAPRIKA returns each refactored version, and the original APK. During our empirical study, we prove that the automatic correction of detected code smells does not alter the behaviour of the app.
Implementation: The correction phase is performed using SPOON [49], a library for static analysis and transformation of Java source code. For the transformation, SPOON works on a model of the input source code. SPOON generates the model parsing the input source code, to produce a raw *Abstract Syntax Tree* (AST). Each node of the AST corresponds to a source program element like `Class`, `Method`, `Field`, `Statement`, `Expression` and so on. SPOON allows the navigation through a type of nodes (*e.g.*, classes) via SPOON processors. SPOON processors allow developers to perform a specific action, like refactoring. For each type of code smell detected in the first step, we write a dedicated SPOON processor to perform the correction of this type. PAPRIKA starts each SPOON processor independently on the app source code to generate as many APK versions as code smell types. Finally, the tool applies the whole set of SPOON processors on the app source code to correct all of those, and generates a fully corrected version.
Processor Example (IGS): The IGS's processor correction works on the *invocation* (*e.g.*, *methods call's*) elements of each class of the app in the analysis output. For the IGS type, the analysis output corresponds to the class that the IGS is detected. This processor checks if the IGS called is a getter or a setter. After this, it retrieves the accessed field from the getter or the setter body. For example, for the getter case, it gets the field from the return statement, catching the returned expression. To correct IGS code smells, we have to replace all getter and setter calls by a direct access to the given class field. For the getter, we replace the *expression* of the getter with a direct call to the field. For the setter, we replace the *invocation* of the setter with an assignment *statement*.

Listing 6. Replace the invocation of the setter with a direct assignment

```
public class IGSProcessor extends
    AbstractProcessor<CtInvocation> {
  ...
@Override
public void process(CtInvocation invok) {
  ...
CtStatement igsSetter = getFactory().Code().
  createCodeSnippetStatement(getField + " = "
      + invok.getArguments().get(0));

invok.replace(igsSetter);
   ...
```

C. NAGA VIPER

To deliver the most energy-efficient Android app, our approach evaluates different versions of an app, by collecting energy metrics for each versions. To this end, HOT-PEPPER uses a tool called NAGA VIPER, to compute energy metrics and evaluate the impact of corrected APKs.

Step 3: Computing Energy Metrics

Input: The original Android APK, each fixed Android APKs

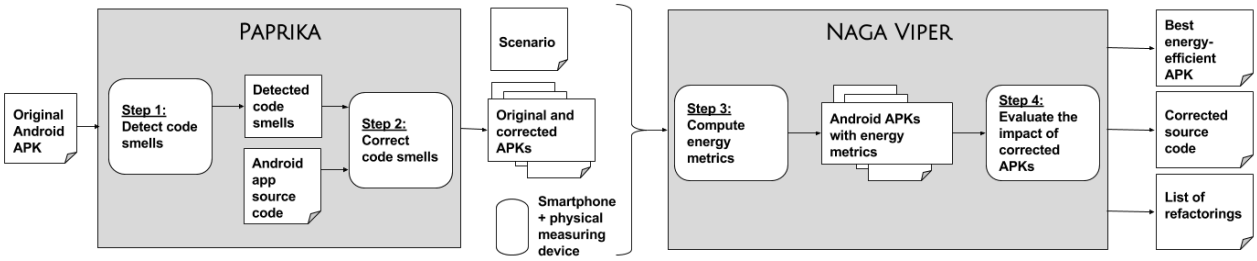

Fig. 1. Overview of the Hot-Pepper approach.

from PAPRIKA, a user scenario of the Android app, and an Android smartphone plugged to a physical measuring device.
Output: Computed energy metrics for the original Android APK and all fixed ones
Description: NAGA VIPER computes at runtime energy metrics on an Android smartphone by executing a user scenario on the app under test. The tool computes the average energy consumption of the smartphone during the execution of the app, the average execution time of the app and the voltage of the battery powering the smartphone.
Implementation: NAGA VIPER starts collecting a parameterised number of metric values per second, as soon as the scenario starts and until it stops. For the computation of the average smartphone's energy consumption (I_{Cons}), the tool uses a physical measuring device (ammeter, voltmeter...) plugged on the phone. This device provides an API to retrieve easily accurate measures from the smartphone. This method allows NAGA VIPER to compute the metrics based on accurate values in runtime, which correspond to the intensity of the smartphone along the scenario executions, that the smartphone consumes (ampere, volts, etc.). The second energy metric to compute is the average execution time of the scenario (t), for each Android version. The third metric value to take account is the voltage of the smartphone's battery (V). This last metric is a constant, so NAGA VIPER have to compute only one time the voltage of the battery. All metrics values are sent to a remote server, which collect them in order to compute average energy metrics. NAGA VIPER runs a parameterised number of times the Android app and the collection of energy metrics, in order to reduce the impact of interferences that can be caused by external factors. Once those running are performed, the remote server computes all average energy metrics, and iterates on the next Android app version to evaluate. Finally, when all Android apps have been evaluated, the server send computed energy metrics and associated Android APKs to the next step.

Step 4: Evaluating the Energy Impact of Corrected APKs
Input: Android APKs and associated energy metrics.
Output: A bundle that contains the most energy-efficient APK to deploy, the fixed Android source code associated to this APK, and a file that describes source code changes.
Description: HOT-PEPPER identifies the best energy-efficient Android app by comparing energy saving percentages between each Android APK version, including the original one. Once this evaluation is done, NAGA VIPER delivers the most energy-efficient APK, the associated source code, and the file that contains source code transformations.
Implementation: First, based on energy metrics computed in *Step 3*, NAGA VIPER computes the global energy consumption (cf. Equation 1) for each Android app version. Second, NAGA VIPER computes the percentage of energy saving between each corrected version and the original one. This percentage value is used to evaluate the difference of energy consumption between those studied versions. Finally, NAGA VIPER gets the Android APK associated to the highest percentage of energy saving, and informs the developer which APK is the best energy-efficient version, for the given scenario. The tool outputs a bundle that contains the best energy-efficient APK to deploy, the associated Android source code, and the file that describes all source code refactorings.

IV. STUDY DESIGN

In this section, we explain the design of our study, which aims to evaluate the energy impact of Android bad practices described in Section II.

For this purpose, we address the following research questions:
RQ_1: *Does the correction of Android code smells improve the energy consumption of the mobile phone?*
RQ_2: *Does picture smells have an impact on the energy consumption of the mobile phone?*

A. Objects

We propose to evaluate the energy consumption of three different Android code smells using HOT-PEPPER. To this end, we had to find several open-source Android apps, which are available in public and popular apps store, like *F-Droid*. The apps were selected according to the following criteria:

- the category and usability of the Android app;
- the popularity of the app, based on the rating and the number of downloads;
- and the presence of two code smell types (minimum), previously studied in Section II: HMU, IGS, and MIM.

Therefore, we ran the PAPRIKA analysis on over 1,900 apps, through *F-droid*. Following the previous criteria, we found 34 apps in six categories : Music, Reader, Productivity, Utility, Sport, and Education. We selected, for each category, the apps that contain the highest number of code smells, for at

least one type. Also, we ensured that these apps could be built and that the license of each app is fully open-source. At the end of this process, among the 34 apps, we selected five Android apps from five different categories, as reported in Table I. We did not select any Sport apps, because the only app found in this category required to use a smartwatch. The selected apps are *Aizoban*, *Calculator*, *SoundWaves Podcast*, *Todo*, and *Web Opac*. *Aizoban* is an online/offline catalogue of mangas that allows the user to read, save, and download their favorites mangas. The version of the app is the version 1.2.5. *Calculator* (version 5.1.1) is the default calculator of the CyanogenMod ROM 11[7]. *SoundWaves Podcast* is a podcast app that allows the user to search, download, and listen to podcasts. The study was performed on the version 0.130 of the app. *Todo* (version 1.0) is an app that allows users to create and customise their 'to do' lists. *Web Opac* (version 4.5.9) allows the access to a catalog of more than 500 public libraries, over 30 countries. The functionalities of this app include: search a library, find a book in the catalog of a library, and make a book reservation. The source code of each app is available on GitHub or SourceForge, via their own F-Droid homepage.

In addition to this analysis, we performed experiments on the three picture smells. However, most of the open-source apps available on F-Droid do not store their pictures locally. As a consequence, we cannot edit enough pictures of popular apps to achieve our study. For this purpose, we rather used seven custom apps to evaluate the energy impact of picture smells. Each custom app loads several random pictures, with a specific picture format, size, and bitmap configuration.

B. Subjects

The subjects of the study were the three Android code smells and the three picture smells previously described in Section II.

C. Experimental Design

For the three Android code smells studied, we corrected the five apps reported in Table I using PAPRIKA, and we obtained several corrected versions of the app. Those versions are described in Table II. V_0 is the original version of the app. V_{HMU}, V_{IGS}, and V_{MIM} are derived from V_0 by correcting HMU, IGS and MIM respectively. Finally, V_{ALL} is the version where all Android code smells are corrected.

For the study of the three picture smells, we created seven custom apps. For the picture format smell, we use two apps that contain PNG or JPEG pictures: V_P and V_J . For the picture size smell, we propose to evaluate the V_J version and a new one: V_{JO}. This version contains a compressed version of each JPEG picture from V_J. The compression has been performed using a proprietary software named JPEGMini[12], which reduces the size of the picture without degrading its visual quality. Finally, we propose to evaluate four Android apps to study Android bitmap code smells: V_{A_P} and V_{A_JO} correspond respectively to the use of the ARGB_8888 bitmap configuration on PNG and JPEGO format , and V_{R_P} and V_{R_JO} correspond to RGB_565 bitmap configuration associated to PNG and JPEGO formats.

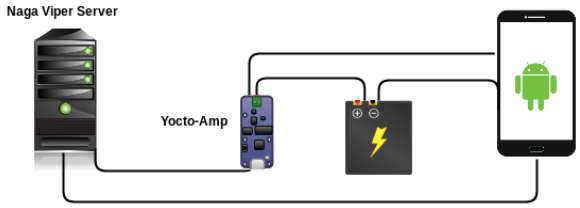

Fig. 2. Yocto-Amp embranchment with the NAGA VIPER server and an Android phone

D. Procedure

Technical Environment: The mobile device used is a Google Nexus 4 and the OS is CyanogenMod 11, which uses Android 4.4.4 (KitKat) version. CyanogenMod 11 is very similar to the 4.4.4 version of Android. The major differences between these versions are that CyanogenMod includes additional developers options, *e.g.* it offers a simple way to turn on/off the root option (that is essential for our experiments), and does not include some apps and services provided by Google. As depicted in Figure 2, we used also an ammeter as a physical measuring device and plugged it in a specific battery harness to the smartphone's model. The harness has been plugged between the battery and the mobile phone, and the ammeter is connected to the harness and the server to collect metrics.
Minimization of External Factors: To make our experiments reproducible, we set a specific environment with specific rules. First, to communicate with the mobile phone during our experiments, we use the USB port of the phone to reach the ANDROID DEBUG BRIDGE (ADB). We disable the USB charging to use NAGA VIPER, and we re-enable it to reload the battery. After reloading it, we shut it down during five minutes to cool down the battery heat. Also, we noticed some side effects on the energy consumption of Android apps under 50% of the battery level. Therefore, it is important to maintain the battery level above this value. We disable mobile data, Bluetooth, GPS, and Wi-Fi, when the application does not need an Internet access (*e.g.*, Todo app). The screen brightness and the sound are set to the minimum. Before we start each experiment for a given app, we make sure that all the other apps of the phone are stopped. For each app, we use a deterministic scenario, based on user events. We run 20 times each scenario on a specific app and collect the energy metric values using NAGA VIPER. Therefore, we make sure to reduce the influence of any external factors. Furthermore, the phone must be fully charged between each set of 20 runs, to start the scenario of each app at the same battery level. Finally, we restart the phone, and we repeat the procedure for an other app.

1) Evaluation of Android Code Smells using HOT-PEPPER*:* **Detection and Correction of Code Smells:** First, we get the original APK of each selected app (V_0) and the associated

TABLE I
LIST OF THE FIVE ANDROID APPS (category, main package, number of classes, methods, and code smells)

Apps	Category	Main package	#Class	#Method	#HMU	#IGS	#MIM	All
Aizoban	Reader	com.jparkie.aizoban	524	2773	39	190	110	339
Calculator	Utility	com.android2.calculator3	147	830	0	10	8	18
Web Opac	Education	de.geeksfactory.opacclient	367	2176	48	77	43	168
SoundWaves	Music-Streaming	org.bottiger.podcast	520	2,672	5	47	14	66
Todo	Productivity	com.xmission.trevin.android.todo	161	610	9	3	0	12

TABLE II
EXPERIMENTAL VERSIONS FOR THE ANDROID APPLICATION

Version	Corrected Code Smells
V_0	None
V_{HMU}	HashMap Usage (HMU)
V_{IGS}	Internal Getter/Setter (IGS)
V_{MIM}	Member Ignoring Method (MIM)
V_{ALL}	All (IGS + MIM + HMU)

source code to run PAPRIKA's analysis and get the list of detected code smells. For each code smell type, we have a list of classes and methods where code smells are located.

After the detection of code smells, PAPRIKA applies the correction step on the source code of the app. PAPRIKA starts the correction processors independently on the V_0 version of the app to generate each of its corrected versions: V_{HMU}, V_{IGS}, and V_{MIM}. Then, the tool runs the correction processors simultaneously on the V_0 version of the app to generate V_{ALL}. We validated the automated correction done by PAPRIKA by comparing manually, for a given app namely `Soundwave`, a correction done manually with an automated one. This comparison has been performed in terms of energy consumption, scenario behaviours, and source code, and was perfectly identical.

Scenario Test: After the correction of code smells, NAGA VIPER runs a specific scenario on the app. This scenario uses `Calabash` [4], an automated testing technology for mobile apps written in Ruby. For each app, we create a `Calabash` scenario that browses a maximum of app's features. Each `Calabash` scenario is based on *x* steps and *y* waits. Steps correspond to user events (*e.g.*, click, scroll, input text, clear text, etc.) and waits correspond to delays, in seconds. Synchronisation barriers are used to ensure that all views are fully loaded before using any feature.

As reported in Tables I and III, scenarios cover a representative number of code smell types and occurrences.

Computation and Evaluation of Energy Metrics: Each scenario has been run 20 times for an app version. The tool collects 75 intensity values per second directly on the smartphone, during the execution time of the scenario, via the ammeter's API. Those values are associated to timestamps that allow NAGA VIPER to compute the execution time of the app. Once intensity values and the execution time of the app have been collected, NAGA VIPER computes energy metrics as explained in Section II, for each run. These energy metrics, for the 20 runs, are used to compute the global energy consumption of the app. In addition to these energy metrics, we compute Cliff's δ stats to evaluate if the difference of energy consumption between each app is significant or not.

2) Evaluation of Pictures Smells using NAGA VIPER*:*

Pictures apps: To run our experiments on picture smells, we created seven versions of a custom app. This custom app is composed of 12 pictures found on the internet. These pictures are initially PNGs and have a transparent background, and a size between 0.12 and 1.0 Mbyte each. The app uses a `GridView` (3x4) to display them. This app contains two user events: scrolling to browse pictures and taping on a picture to display it for three seconds. Each version of this app has been explained in the experimental design.

Computation and Evaluation of Energy Metrics: We developed a specific scenario that lasts around 2 minutes. The scenario scrolls and taps several pictures during its execution. We use this scenario for each version of the app. Each scenario has been run 30 times for each version. NAGA VIPER collects the same energy values as Android code smells.

E. Variables

Independent Variables: In our experiments, the independent variables correspond to the number of each of the three code smells IGS, HMU and MIM corrected in each app. In case of our experiments on images, the independent variables are the size and the format of the image as well as the bitmap format used.

Dependent Variables: Previously, in Section II, we have listed the metrics used to evaluate the energy consumption of the app: the execution time of the app, the voltage and the average intensity. These metrics are our dependent variables.

F. Analysis Method

For our analysis method, we perform Cliff's δ effect size [51] to support the results difference's between each version. The tests are performed using a 99% confidence level. A small effect size ($\delta > 0.147$) represents a slight difference hardly visible, a medium effect size ($\delta > 0.330$) represents a visible difference that can be observed, while a large effect ($\delta > 0.474$) is significantly larger than medium. We use the average values of all experiments to compare versions.

V. CASE STUDY RESULTS

This section reports and discusses the results we obtained to answer our research questions.

RQ_1: Does the correction of Android code smells improve the energy consumption of the mobile phone?: For answering $\mathbf{RQ_1}$, we use the 5 popular Android open-source apps listed before. In Table III, we report the global energy consumption for the different versions and the one that consumes the less, in bold. Globally, all corrected versions consume less energy than the original one, except for two corrected versions—*i.e.*, V_{MIM} in `Aizoban` and in `SoundWaves`. However, these two versions have a negligible increase in terms of energy—*i.e.*, these versions consumes 0.08% and 0.29% more than the original app, which may be caused by a measurement error with the physical device.

The best version for `Calculator` and `Todo` is V_{ALL} with an energy saving of 1.69% and 4.83%. The one for `Aizoban` is V_{HMU} with an energy saving of 2%. The one for `SoundWaves` is V_{IGS} with an energy saving of 1.43%. Finally, the one for `Web Opac` is V_{MIM} with an energy saving of 3.86%. Based on Tables IV and V, we can notice that there is a large significant difference between the original app and each corrected version, for `Aizoban`, `Web Opac` and `Todo`. For `Calculator` and `SoundWaves`, we notice that there is a low and a medium difference between the original version and each corrected one. Finally, we can notice that even if V_{ALL} is not the best version for `Aizoban`, `SoundWaves` and `Web Opac`, this version is statistically equivalent to the best version of those apps with -0.01, 0 and -0,11 as Cliff's δ results. So, V_{ALL} can be also considered as the best version for the five selected Android apps.

In summary, the results of our research show that the correction of at least one Android code smell (HMU, IGS or MIM) reduces the energy consumption of the mobile phone. Also, based on Cliff's δ results, we can assert that the correction of all code smells on an app has a better impact that the correction of only one of them.

RQ_2: For answering $\mathbf{RQ_2}$, we studied three different variables: the picture's format (JPEG or PNG), the size of the file (only for JPEG, in MBytes) and the bitmap format of the picture (ARGB_8888 or RGB_565). Overall, we performed our experiments on seven versions of the picture app to measure the impact of each variable, as explained in the study design.

To perform our experiments on JPEG pictures, we converted the previous pictures to JPEG, and reduced the size of each file until the minimum size has been reached, using JPEGMini.

The original directory size that contains PNG directory is 700 KBytes, 1 MBytes for the JPG directory and 367 KBytes for the optimised JPG directory. Results are given in Table VI.

Each experiment, for each app version, has been run 30 times. Using 30 tests, the average power consumption of the PNG app is 0.0525 Joules, 0.0536 Joules for the JPG app and 0.0535 Joules for optimized JPG app. We notice that only the power consumption of the PNG app has changed. However, we notice also that there is no statistical difference between the JPEG app and the optimised JPG app. Those observations rely on Cliff's δ computations in Table VII, and comfort us to think that the size has no influence in the global energy consumption of an Android app. For Android bitmaps the global energy consumption of V_{R_JO} is up to 0.0490 Joules, that is greater than V_{A_JO} that consumes 0.0487 Joules, which is the best version of the app for optimised JPEG files. Also, we found that the version V_{R_P} is up to 0.0524 Joules, in contrary to V_{A_P} that consumes 0.0490 Joules, which is the best version of the app for PNG files. Based on Cliff's δ results in Table VII, we found that the difference between V_{A_P}/V_{R_JO}, V_P/V_{R_P} and V_J /V_{R_P} is not significant. For each other version, the difference of energy consumption is significant, and contributes to conclude that V_{R_P}/V_{A_JO} is the most energy-efficient app. Those results indicate also that V_{A_P} and V_{R_JO} are statically similar and that V_{R_JO} can be considered as the second best version app, in terms of energy consumption.

In summary, we can conclude that the bitmap format has an impact on the energy consumption of the mobile phone. Contrary to what was stated in the Android performance tips, we found that the larger bitmap format (ARGB_8888) has a better energy consumption than the lower (RGB_565).

A. Threats to Validity

In this section, we discuss the threats to validity of our study based on the guidelines provided by Wohlin et al. [55].

Construct validity threat is the relation between the theory and the observations made. For our experimentation, it could be due to measurements errors. For this purpose, we ran many experiments - 20 run for apps with code smells and 30 run for pictures apps -, and used averages values instead of instant values. In addition, in Section IV-D, we described how we tried to reduce the impact of external factors to make our measurements as accurate as possible. Also, measures of our ammeter could be wrong, because they are physical measurements. However, we tried two different ammeters when we ran `Hot-Pepper`, which provided similar results in both cases. Moreover, our reasoning are based on the difference between measures, therefore the impact of such errors is diminished.

Internal validity threat is the causal relationship between the treatment and the outcome. As our results depend on the correction of code smells, we make sure that `Paprika` correct all code smells in the app by investigating manually the source code after the correction process. For pictures smells, the correction is already done manually.

External validity threat is the possibility to generalise our finding. Our results depend on various parameters such as the app (the category, the developer, the permissions, etc.), scenarios used, the number of code smells detected - and their occurrences during a scenarios -, the Android version, and the smartphone. For performance code smells, our results are dedicated to our `Workbench`, explained in Section IV-D, they cannot be generalised at this time. However, we tried to use various apps with different categories, permissions, and code smells, which have been ran with different user scenarios on

TABLE III
GLOBAL ENERGY CONSUMPTION FOR THE DIFFERENT VERSIONS OF THE FIVE ANDROID APPS (# STEPS, WAITS IN SECONDS, # CODE SMELLS CALLS, GLOBAL ENERGY CONSUMPTION IN JOULES)

Apps	#Step	Waits	$\#HMU_c$	$\#IGS_c$	$\#MIM_c$	V_0	V_{HMU}	V_{IGS}	V_{MIM}	V_{ALL}
Aizoban	169	17s	10	1300	0	0.0424	**0.0415**	0.0419	0.0425	0.0418
Calculator	325	0s	0	6122	1350	0.0300	-	0.0300	0.0299	**0.0295**
SoundWaves	172	53s	420	8053	6560	0.0859	0.0856	**0.0847**	0.0867	0.0848
Todo	248	5s	40	20	0	0.0369	0.0361	0.0362	-	**0.0352**
Web Opac	136	79s	6	133	40	0.0392	0.0384	0.0384	**0.0377**	0.0378

TABLE IV
GLOBAL ENERGY CONSUMPTION BENEFIT FOR THOSE THREE FIRST CODE SMELLS STUDIED, IN PERCENT.

Apps	HMU	IGS	MIM	ALL
Aizoban	**-2.00%**	-1.09%	+0.08%	-1.38%
Calculator	-	-0.18%	-0.45%	**-1.69%**
SoundWaves	-0.38%	**-1.43%**	+0.29%	-1.29%
Todo	-2,40%	-2,04%	-	**-4,83%**
Web Opac	-2.06%	-2.08%	**-3.86%**	-3.50%

TABLE V
CLIFF'S δ RESULT FOR EACH VERSION APP

Apps	V_{HMU}	V_{IGS}	V_{MIM}	V_{ALL}	Best vs V_{ALL}
Aizoban	0.58	0.46	-0.06	0.58	-0.01
Calculator	Null	0.11	0.18	0.42	0
SoundWaves	0.08	0.26	-0.24	0.26	0
Todo	0.66	0.62	Null	0.92	0
Web Opac	0.43	0.46	0.69	0.60	-0.11

the same devices, in order to support our results. These results show that in some contexts, there is an impact on energy consumption when correcting the aforementioned smells. So, there is potentially a similar overall effect for most apps. Unfortunately, we can not yet generalise results for pictures smells, as they are based on a sample app. More studies are still needed to confirm our results.

Reliability validity threat is the possibility of replicating this study. To reproduce our experiments, we provide some rules in Subsection IV-D that must be respected, in order to have comparable conditions. In addition to the informations provided in this paper, the apps used and their scenarios are available in a public git repository [1].

Conclusion validity threat supports that the conclusions reached in a study are correct. We paid attention to not violate the assumptions of our statistical tests. For this purpose, Hot Pepper uses only non-parametric tests which are not making any assumptions about the distribution of the metric. Finally, we were careful to not generalise our findings.

TABLE VI
EVALUATION OF EACH VERSION OF THE STUDIED APP, USING NAGA VIPER, IN TERMS OF ENERGY CONSUMPTION. THE DIRECTORY SIZE IS MBYTES. THE UNIT IS JOULES.

Application	Pic. dir. size	DRAW	ARGB_8888	RGB_565
PNG	0.70	0.0525	0.0487	0.0524
JPEG	1.00	0.0536	0	0
JPEGO	0.37	0.0535	0.0482	0.0490

TABLE VII
CLIFF'S δ ESTIMATION BETWEEN EACH VERSION OF THE ORIGINAL APP.

	V_J	V_{JO}	V_{A_P}	V_{A_JO}	V_{R_P}	V_{R_JO}
V_P	-0.74	-0.72	1	1	0	1
V_J	-	0.10	1	1	0.69	1
V_{JO}	-	-	1	1	0.72	1
V_{A_P}	-	-	-	0.2	-0.96	0
V_{A_JO}	-	-	-	-	-0.98	-0.23
V_{R_P}	-	-	-	-	-	0.43

VI. RELATED WORK

In this section, we discuss the relevant literature focusing on energy code smells, energy analysis on Android devices and automated energy optimisation techniques.

Energy Leaks. Most of the literature specific to Android energy leaks takes care of high-level energy bugs like *wake-locks* [23], [29], ads [30] and network utilisation[22], [26], [41]. These approaches focus on energy bugs detection and correction. In contrast, we are interested in reducing energy leaks induced by energy code smells.

Two research papers focus on code smells. Pérez-Castillo and Piattini investigated the effect of God Class refactoring on energy consumption [50] for Android apps. They found that the God Class refactoring increases the energy consumption of an app, this is due to the implementation of new methods, classes and by the increase in the number of lines of code as a consequence of the refactoring. Halfond et al. [39] measures the energy saving for IGS and MIM code smells.This approach is based on a custom app. They found that the IGS refactoring consumes 35% less energy, and that the MIM refactoring consumes 15% less energy, when in our approach IGS refactoring consume at best 2.08% less energy and MIM refactoring consume at best 3.86% less energy. However, the important difference with our results can explained by the context of the experimentation. First, we run our experiments on a different set of real and available apps, when they only used one sample app made for their experiments. In addition, we run our tests on a user-based scenarios when they loop $50,000,000$ times over the code smells invocation. Gottschalk et al. [29] propose a method to improve energy-efficiency on app level, by applying re-engineering techniques, like code analysis and code restructuring. This method is applied on

[1] https://github.com/SOMCA/hot-pepper-data/tree/master/scenarios

mobile apps and considers loop bugs, dead code, in-line methods and cache utilisation as energy code smells.

Energy Consumption Analysis for Android. The overall related work can be split into two categories: static and dynamic energy analysis. Mittal et al. [47] present an energy emulation tool that allows developers to estimate the energy consumption of a mobile app during its development. Marcu and Tudor [45] also explores this aspect to measure the energy consumption with a special device emulator. On the contrary, we based our experiments on a popular android device, the Galaxy Nexus. Experiments using static energy leaks detection uses API invocation trees [41], [56] or modified function call graph [32]. Dynamic detection is the most used technique to experiment on network [31], using test scripts [37], [39], [40] or scenarios [42]. Jabbarvand et al.[38] use both static and dynamic energy leaks detection to rank apps of the same category. The static part is used as a static model extractor to annotate the call graph of the android app, and the dynamic part to run some test-case and have a profiler about this app. Zhang et al. [56] worked on a automatic power estimator for Android, based on user-reported failures. POWERBOOTER is accurate within 4.1% on average, whereas our approach is accurate to within 2mA (in average less than 1% for an android app run), using a physical device plugged directly on the smartphone's battery. Duribreux et al. [26] and Lineares-Vasquez [42] used an hardware power monitor to measure efficiently the energy consumption for a single Android app, instead of measuring bias using an Android app. This physical device can precisely measure the energy consumption spent in the smartphone since it is plugged between the battery and the smartphone. Our work is capable to measure energy consumption per application, using a user-based scenario, with an accurate physic ammeter: the YOCTO-AMP [18].

Correction of Android Code Smells. Actually, except LINT[11], the tools proposed with the *Android Developer Tools* (ADT) and our tool PAPRIKA, there is no specific tools dedicated to the Android platform for the analysis and correction of these code smells. By checking the LINT documentation, we discovered that it can only detect one Android-specific code smells of this study (IGS). There are also specific environments and platforms dedicated to software analysis like IPlasma [46], which is a specific environment for quality software analysis and DECOR [48] for the detection and correction of antipatterns. For the code transformation, there is some tools and studies on Java code refactoring related to code smells. Like JDEODORANT [27], which is an ECLIPSE plug-in that analyse and refactor the code smell detected. However these tools are not specific to Android, and thus consider only Object oriented code smells of the Java language.

VII. CONCLUSION AND FUTURE WORK

Energy consumption has become an important topic in software development and it is even more critical for mobile software engineering, as mobile devices run on limited resources and a limited battery. Various studies prove that some Android code smells decrease Android app's performance, and for this purpose many tools were developed to help developers during their development to detect these code smells [34], [44], [52]. However, there are few works on the energy impact of code smells. Moreover, there are no tools or frameworks available to evaluate Android apps and help developers to detect and correct code smells.

In this paper, we presented an empirical study on six Android apps (five Android open source apps and one custom app) to assess the energy impact of Android performance code smells (*HashMap Usage* (HMU), *Internal Getter/Setter* (IGS), and *Member Ignoring Method* (MIM)) and Android picture smells (*Picture Format, Picture Size*, and *Picture Bitmap Usage*). For this purpose, we developed an automated approach called `Hot-Pepper` that detects and corrects these code smells and assess their energy impact to retrieve the best version of the app.

Our results show that these code smells have an impact on the energy consumption, and that by correcting them we can improve the energy efficiency of an Android app. The correction of only one performance code smell (MIM) can reduce up to 3.86% the energy consumption of an app, while the correction of all code smells can reduce up to 4,83% of the energy consumption. For Android picture smells, we found that using the best compression, associated to the default Android bitmap format has a positive impact on the energy consumption of the mobile phone. Globally, we observed that the correction of all Android code smells is the best practice, even if it can be outperformed in some cases by the correction of a single code smell.

We believe that this study will benefit the developers by helping them to avoid the usage of code and picture smells. Therefore, we hope that our approach will be useful to evaluate apps and help to reduce their energy consumption. For future works, we will extend our studies by investigating more code smells and apps. In addition, we plan to work on a non-intrusive method to collect our energy metrics. Finally, we also plan to deploy our approach and protocol on a cloud platform to make it more accessible for developers.

REFERENCES

[1] Android developer, core app quality. https://developer.android.com/distribute/essentials/quality/core.html. [Online; accessed October 2016].

[2] Android performance tips. http://developer.android.com/training/articles/perf-tips.html. [Online; accessed January-2016].

[3] Butteraugli, a tool for measuring differences between images. https://github.com/google/butteraugli. [Online; accessed May-2016].

[4] Calabash, an automated acceptance testing for mobile apps. http://calaba.sh. [Online; accessed May 2016].

[5] Canvas and drawables. https://developer.android.com/guide/topics/graphics/2d-graphics.html. [Online; accessed June 2016].

[6] comscore. https://goo.gl/ai2yUA. [Online; accessed October-2016].

[7] Cyanogenmod, a customized, aftermarket firmware distribution for several android devices. http://www.cyanogenmod.org. [Online; accessed August 2016].

[8] Global smartphone sales by operating system from 2009 to 2015. https://www.statista.com/statistics/263445/global-smartphone-sales-by-operating-system-since-2009/. [Online; accessed October-2016].

[9] How jpeg works? https://medium.freecodecamp.com/how-jpg-works-a4dbd2316f35#.eo0zkkw0g. [Online; accessed May-2016].

[10] How png works? https://medium.com/@duhroach/how-png-works-f1174e3cc7b7#.22z9sbyyp. [Online; accessed May-2016].

[11] Improve your code with lint. https://developer.android.com/studio/write/lint.html.

[12] Jpegmini - your photos on a diet. http://www.jpegmini.com. [Online; accessed June 2016].

[13] Learn when to use jpeg, gif, or png with this graphic. http://lifehacker.com/learn-when-to-use-jpeg-gif-or-png-with-this-graphic-1669336151. [Online; accessed August 2016].

[14] Managing bitmap memory. https://developer.android.com/training/displaying-bitmaps/manage-memory.html. [Online; accessed June-2016].

[15] Peak signal-to-noise ratio. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio. [Online; accessed May-2016].

[16] Reducing jpeg file size. https://medium.com/@duhroach/reducing-jpg-file-size-e5b27df3257c#.r7db4f69r. [Online; accessed May-2016].

[17] Reducing png file size. https://medium.com/@duhroach/reducing-png-file-size-8473480d0476#.oh8qcg84b. [Online; accessed May-2016].

[18] Yocto-amp, a physical ammeter. http://www.yoctopuce.com/EN/products/usb-electrical-sensors/yocto-amp. [Online; accessed March-2016].

[19] What optimizations can i expect from dalvik and the android toolchain? http://stackoverflow.com/a/4930538, 2011. [Online; accessed January-2016].

[20] Arraymap. http://developer.android.com/reference/android/support/v4/util/ArrayMap.html, 2015. [Online; accessed January-2016].

[21] Displaying bitmaps efficiently. http://developer.android.com/training/displaying-bitmaps/index.html, 2016. [Online; accessed May 2016].

[22] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 280–293, 2009.

[23] A. Banerjee, H.-F. Guo, and A. Roychoudhury. Debugging energy-efficiency related field failures in mobile apps. MobileSoft, 2016. To appear.

[24] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM.

[25] M. Brylski. Android smells catalogue. http://www.modelrefactoring.org/smell_catalog, 2013. [Online; accessed January-2016].

[26] J. Duribreux, R. Rouvoy, and M. Monperrus. An energy-efficient location provider for daily trips. Technical report, 2014.

[27] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: Identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1037–1039, New York, NY, USA, 2011. ACM.

[28] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: improving the design of existing code. 1999. *ISBN: 0-201-48567-2*.

[29] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter. Removing energy code smells with reengineering services. In U. Goltz, M. A. Magnor, H.-J. Appelrath, H. K. Matthies, W.-T. Balke, and L. C. Wolf, editors, *GI-Jahrestagung*, volume 208, pages 441–455, 2012.

[30] J. Gui, D. Li, and W. G. Wan, Mianand Halfond. Lightweight measurement and estimation of mobile ad energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software – GREENS*, May 2016. To appear.

[31] J. Gui, S. Mcilroy, M. Nagappan, and W. G. J. Halfond. Truth in advertising: the hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering – ICSE*, pages 100–110. IEEE, 2015.

[32] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang. Characterizing and detecting resource leaks in android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 389 – 398. IEEE, 2013.

[33] C. Haase. Developing for android, ii the rules: Memory. https://medium.com/google-developers/developing-for-android-ii-bb9a51f8c8b9, 2015. [Online; accessed January-2016].

[34] G. Hecht, N. Moha, and R. Rouvoy. An empirical study of the performance impacts of android code smells. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*. IEEE, 2016. To appear.

[35] G. Hecht, B. Omar, R. Rouvoy, N. Moha, and L. Duchien. Tracking the software quality of android applications along their evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering*, page 12. IEEE, 2015.

[36] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. Detecting antipatterns in android apps. In *Research Report - INRIA, Lille*, 2015.

[37] A. Hindle, W. A., R. K., B. E. J., C. J. C., and R. S. Greenminer:a hardware based mining software repositories software energy consumption framework. In *11th Working Conference on Mining Software Repositories – MSR*, page 12–21, May 2014.

[38] 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*, pages 8–14. IEEE, 2015.

[39] 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*, pages 46–53. ACM, 2014.

[40] D. Li, S. Hao, H. W. G. J., and R. Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis – ISSTA*, pages 78–89, 2013.

[41] D. Li, Y. Lyu, J. Gui, and W. G. Halfond. Automated energy optimization of http requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering – ICSE*, May 2016. To appear.

[42] M. Linares-Vasquez, G. Bavota, C. Bernal-Cardenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk. Mining energy-greedy api usage patterns in android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories – MSR*, pages 2–11, 2014.

[43] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 1013–1024, 2014.

[44] Y. Liu, C. Xu, and S.-C. Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1013–1024. ACM, 2014.

[45] M. Marcu and D. Tudor. Energy consumption model for mobile wireless communication. In *Proceedings of the 9th ACM international symposium on Mobility management and wireless access*, pages 191–194. ACM, November 2011.

[46] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. pages 77–80, 2005.

[47] R. Mittal, A. Kansal, and R. Chandra. Empowering developers to estimate app energy consumption. In *ACM Mobicom*. ACM, August 2012.

[48] N. Moha. Detection and correction of design defects in object-oriented designs. pages 949–950, 2007.

[49] 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*, page na, 2015.

[50] R. Pérez-Castillo and M. Piattini. Analysing the harmfull effect of god class refactoring on power consumption. In *IEEE Software*, volume 31, pages 48–54. IEEE, January 2014.

[51] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[52] G. A. Sağlam. *Measuring And Assesment Of Well Known Bad Pratices In Android Application Developments*. PhD thesis, Middle East Technical University, 2014.

[53] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: From error visibility to structural similarity. In *IEEE TRANSACTIONS ON IMAGE PROCESSING*, volume 13, April 2004.

[54] A. I. Wasserman. Software Engineering Issues for Mobile Application Development. *ACM Transactions on Information Systems*, pages 1–4, 2010.

[55] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[56] L. Zhang, B. Tiwana, R. P. Dick, and Z. Qian. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Hardware/Software Codesign and System Synthesis – CODES+ISSS*, pages 105–114. IEEE, 2010.