University of Lille (Lille 1)
Université du Québec à Montréal

Master of computer science, MoCAD

Research internship report

Eco-Design of Android Applications

An Automated Approach to Assess and Improve the Energy Consumption of Android Applications

Student :
Antonin Carette

Supervisors :
Pr. Naouel Moha
Dr. Romain Rouvoy
PhD Geoffrey Hecht

29 août 2016

Contents

Acknowledgements

The internship opportunity I had with the research laboratory Latece in Montréal, Canada, was a great chance to develop my research and professional skills, while discovering a new culture. I am also grateful for having a chance to meet many wonderful people and professionals who led me though this internship period.
Bearing in mind previous, I am using this opportunity to express my deepest gratitude and special thanks to Mrs. Naouel Moha, associate professor at Université du Québec à Montréal and my principal supervisor of Latece who, in spite of being extraordinarily busy with her duties, took time out to hear, guide and keep me on the correct path.
I express my deepest thanks to Mr. Romain Rouvoy, associate professor at University of Lille, and Mr. Geoffrey Hecht, PhD student at Université du Québec À Montréal (UQÀM) and University of Lille, for taking part in useful decision and giving necessary advices.
To finish, I express my best regards and deepest sense of gratitude to Mr. Olivier Pietquin, head of the Master MoCAD at University of Lille and research scientist at Google DeepMind, for his careful and precious guidance before this internship, and Quebeckers for their kindness and the poutine (obviously).

Introduction

From 2007 to 2015, more than 1,5 billion of smartphones have been sold to end-users worldwide [1]. For the 4th quarter 2015, 352 millions of Android smartphones have been sold, compared to 71.53 millions of Apple mobile devices and 4.4 millions of Microsoft mobile devices [2]. In 2015, nearly 2 billions of the Android operating system have been installed on a mobile device, against 463 millions of iOS operating system and 45 millions of Windows operating system [3]. Android is the first mobile phone operating system used in the world. Due to its open source architecture and its free cost, Android is the first mobile platform for mobile developers, with 86% mobile developers programming for Android [4], in November 2015. Unfortunately, the rise of mobile apps development induces ever faster developments and less focus on good development practices, which may cause the appearance of bad practices.

Code smells are symptoms of the possible presence of design smells, included in design smells [1], which are poor solutions to solve a problem. Code smells are present in object-oriented programming languages, and some of those are well-known by developers. For Android, performance code smells have been warned by Google in the Android developer documentation [2] [3], to help developers to make better apps. A recent research had demonstrated that tracking code smells for an Android app is a quality and performance proof for the app [4]. Also, the correction of some Android code smells can improve up to 12.4% performance, on UI metrics [5]. The main focus of Android developing these last years was especially on improving performance of Android apps. Currently, the main problem of smartphone owners is the energy consumption [6]. Today, a smartphone is gone flat in a few hours or

1. Number of smartphones sold to end-users from 2007 to 2015 : `http://goo.gl/WzR8ck`.

2. Global smartphone sales to end users from 2009 to 2015, by operating system : `http://goo.gl/Kufy6x`.

3. Installed base of smartphones by operating system in 2015 : `http://goo.gl/Bw9ldR`.

4. Mobile platforms currently developing for according to global app developers, as of November 2015 : `http://goo.gl/UJW9j4`.

a few days, depending of the full-charge of the battery, its capacity, and how the smartphone's owner uses it all day, through the usage of Android apps for example.
Today, few researches have been done to study the effect of code smells on energy consumption [7] [8]. Those researches have focused on evaluating the energy consumption of code smells performance using custom, or made on proposal, Android apps, instead of popular Android apps available in public apps stores like the Play Store [9] or F-Droid [10].
We propose to evaluate the energy consumption of Android bad practices using popular Android applications. For this, I developed an approach, supported by a framework, to deal with this problem.

During this internship, I enumerated three main contributions :

1. the development of an automated approach to assess and improve the energy consumption of Android applications, called Hot-Pepper,
2. the validation of Hot-Pepper using an empirical study on five popular Android apps, available in popular apps stores,
3. the evaluation of six Android bad practices, in terms of energy consumption.

This research report is organized as follow. the first chapter concerns bad practices in Android development and energy consumption understanding. The second chapter is a state-of-the-art about Android bad practices, and the energy consumption analysis for Android. The third chapter deals with our developed approach to evaluate the impact of Android bad practices, in terms of energy consumption. The fourth chapter describes our empirical study and discusses the results obtained. Finally, the last chapter is dedicated to the conclusion of this internship, and future work.

A research paper is currently under preparation, which deals about the approach and the empirical study [11].

Chapter 1

Background

In this chapter, we briefly introduce the Android operating system, the OS that we target in our experiments, some bad practices in Android development, and interesting energy metrics to get the energy consumption of an Android app.

1.1 The Android Operating System

To explain bad practices in Android development, we have to introduce the operating system that we use in our experiments and in our approach. Android is an open source multitasking OS, for embedded devices like mobile phones, smartwatches, cars or televisions [12]. Based on the Linux kernel [1], Android is developed by Google [2] since 2005. Four principal layers compose the software architecture of the operating system (figure 1.1). These layers, from the lowest to highest, are :

— the *Linux kernel* (*Linux kernel*),
— some C/C++ libraries and the Android virtual machine (*Native libraries*),
— the *Android Framework*,
— some apps and widgets installed by default (*Android Applications*).

Android apps are developed using the Java language, and take advantage of the Google API to compose with the whole smartphone's features.
As Android uses the Java language, each app is compiled in an Android bytecode, called DEX, to run into a virtual machine [3]. The virtual machine can be Dalvik or Android RunTime (ART), according to the version of the

1. `https://en.wikipedia.org/wiki/Linux_kernel`
2. `https://www.google.com/intl/fr/about/`
3. `https://en.wikipedia.org/wiki/Virtual_machine`

FIGURE 1.1 – Android architecture schema.

OS [13]. As Android is multitasking, it is capable to launch and run simultaneously several apps. In order to perform the multitasking feature, each app has to be launched in a virtual machine and, for security reasons, has to contain its own instance of the virtual machine [14]. So, Android uses its own virtual machine essentially to optimize the launch of its multiple instances, each time is launched. In this way, the Android virtual machine is different than the Java Virtual Machine (JVM) [15].
To let the Android developer communicate with his Android device, he can use the ANDROID DEBUG BRIDGE (ADB) [16]. ADB is a client-server program that permits to access to Android features like installing or uninstalling an app, enabling or disabling the log feature, checking the current state of an app or event launching some Android specific programs, like ANDROID MONKEY [17].

1.2 Bad Practices in Android Development

In this research report, we introduce the expression *bad practices* as poor solutions located inside and/or outside of the source code. We talk specifically about a bad practice as a *code smell* if this one is located in the source code.
Code smells are associated with poor coding practices. Those practices are associated to long-term maintainability problems, and can mask bugs. [18] [4] Also, code smells hind the evolution of applications, and degrade the quality of the software [4]. As a result, the end-user experience is impacted [19].
Even if Android applications are developed using Java, an Object Oriented programming language, objects oriented anti-patterns are not fully supported for Android [20] [21] [18] [22]. First of all, mobile applications use event-driven programming, and do not use desktop application's libraries or desktop application's APIs. Secondly, an Android developer can not use libraries or APIs like Swing, JavaFX to develop his app. Also, GUIs are only declared as XML files. [23] Thirdly, changes induces in the usage of a specific virtual machine have important implications both the compilation and the execution of the app [13] [24] [25]
Reimann et *al.* have identified 30 possible code smells for Android. [26] [27] We propose to study briefly 3 common specific code smells in Android applications, which can be found as substantial proportions : HashMap Usage (HMU), Internal Getter/Setter (IGS) and Member Ignoring Method (MIM) [4].

Three Android Code Smells

HMU is a code smell related to memory management [28]. An `HashMap` is a data structure that can map keys to values, where each key is associated to a unique slot of values. Even if Oracle confirms that the implementation provides constant-time performance for the basic operations, a `HashMap` is a complex data structure and has a heavy memory and energy cost to store more that hundred elements [29] [30]. For this reason, Android provides `ArrayMap` and `SimpleArrayMap` as replacements from traditional Java `HashMap`, which are more memory-efficient [29].

IGS and **MIM** are related to micro-optimisation code smells [28]. An Internal Getter/Setter is an Android code smell that occurs when an attribute is accessed in the app code. This attribute is accessed through a getter or a setter. Considering that the Android virtual machine can not inline the access, the usage of getters and setters are indirect access to attributes, and may decrease the performance of the app [31]. A Member Ignoring Method is an Android code smell that occurs when the app does not use a static method

for a method that does not access an object attribute. The correction of this code smell improve the performance reaction between 15 to 20% [3].

Three Picture Bad Practices

The usage of pictures can deal with several performance problems in Android apps, if bad practices are introduced. We propose to deal with three recognized bad practices for pictures : picture formats, the picture compression, and the picture bitmap usage.

Picture Formats

Three picture formats are frequently used in Android apps : GIF, JPG and PNG.
The GIF format is a lossless compression picture format limited to 256 colors. GIF files are used on the web to animate plural pictures. Its usage is highly discouraged in Android apps [32] [33]. JPG [34] is a lossy compression picture format, which makes it a common choice too for storing big pictures like photographs or realistic images. This picture format uses 16 bits per pixel. PNG [35] 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. As it lossless and uses a greedier pixel compression format, PNG is a good choice for storing pictures at a small file size and images with few colors. Moreover, this format is known as a bad practice for big pictures. In such case, the JPG format should be preferred [32] [34] [35].
Here, Android developers have to take care about the format they are using for each picture they want to display, according to the size of the picture, the number of colors encoding the file and the usage (icon, photograph, etc...) [33].

Picture Compression

A common problem for storing pictures is to have the best trade-off between the size of the picture and it quality. The more the size of the file augments and the more it required resources to load and display it in a screen. Also, reducing a lot a picture can degrade a lot the picture and have a bad impact on the visual quality. A solution is to reduce the picture size without degrading the visual quality of this one.
Existing algorithms like PSNR [36], SSIM [37] and Butteraugli [38] are solutions to measure accurately the difference and similarity of two pictures.

The reference of this evaluation is related to the perceived difference of both same pictures, in different resolution, using the human eye. These solutions provide excellent ways to get the best tradeoff between image compression and visual quality of a picture.
Using these algorithms and other techniques, it is possible to reduce the size of the file without degrading the users experience [39] [40].

Picture Bitmap Usage

Android bitmap objects can consume a lot of memory for large pictures, especially when several pictures are displayed in the same time in Android views [41]. A common problem with Dalvik is that it can not defragment in real time the memory, for a single running app. Due to this behavior, the app become slower. Android developers have to take care about how to load and cache effectively bitmaps. Android provides three different bitmap formats : `ARGB_8888`, `ARGB_4444` and `RGB_565` [42].
`ARGB_8888` allows to use 4 bytes to encode each pixel of the file. `ARGB_4444` allows to use 2 bytes for each pixel. The usage of this bitmap format is discouraged due to poor quality of its configuration. For `RGB_565`, each pixel is stored on 2 bytes, and only RGB channels are encoded.
Even if `ARGB_8888` uses more bytes per pixel, this format is the default one for 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 observe a big visual difference with the same picture but with lower pixel density [43].

1.3 Energy Consumption Understanding

Our study is based on the energy model proposed by Chiyoung Seo, Sam Malek and Nenad Medvidovic from the University of California [44] [45]. Their model, referred in Equation 1.1, is interesting because it defines the global energy consumption of a Java application as the energy to access and compute data, the energy to send data on the network, and the infrastructure cost (like the JVM).

$$E_{\mathsf{javaApp}} = E_{\mathsf{comp-access}} + E_{\mathsf{com}} + E_{\mathsf{infra}} \tag{1.1}$$

We propose to adapt this model in measuring a mobile application adding the energy spent by the operating system, during the execution time of the application. Our model is modelled as the Equation 1.2.

$$E_{\mathsf{mobileApp}} = E_{\mathsf{OS}} + E_{\mathsf{javaApp}} \tag{1.2}$$

To measure the global power consumption of an Android application, we accept to model this one (E_{global}) as the sum of the voltage of the battery (V, in Volts), multiplied by an intensity measure (I_{average} in Amperes) and the delta time corresponding to the interval between the associated timestamp and the timestamp of the next intensity measure (Δt in seconds) - see Equation 1.3.

$$E_{\mathsf{global}} = \sum (V * \Delta t * I_{\mathsf{average}}) \tag{1.3}$$

Chapter 2

Related Work

In this chapter, we discuss the literature focusing on Android bad smells and energy analysis on Android devices.

Detection and Correction of Android Bad Practices

Many tools have been created to detect OO anti-patterns in Java applications, like DECOR [1], JDEODORANT [46], KLOCWORK [47] or FINDBUGS [48]. Those tools are inspired by the work of William Brown et al. [49] and Martin Fowler [20].
As explained in Chapter 1, OO anti-patterns are not fully supported for Android, and those tools are not specific to Android.
G. Hecht and *al.* have developed a Java application called PAPRIKA [24] that detect automatically 17 OO code smells, including 14 specific Android code smells. This tool has been improved these three last months to correct automatically detected code smells. Today, PAPRIKA can correct three Android code smells.
Some experiments uses API invocation trees [50] [51] or modified function call graph [52] to detect statically Android bad smells.
Dynamic detection is the most used technique to experiment on network [53], using test scripts [54] [55] [56] or scenarios [57].
Jabbarvand and *al.* uses both static and dynamic energy leaks detection to rank applications of the same category [58]. The static of this tool, called ECODROID, annotates the call graph of the android app for each bad smell detected. The dynamic part to run test cases on the annotated code to create a profiler for this app. The combination of these models produces a score, which is used to rank the application.

Evaluation of Android Bad Practices

Most of the literature, specific to the energy impact of Android bad practices, is based on high-level energy bugs like wakelocks [59] [60], ads [61] and network usage [62] [63] [50]. In contrast, we are interested in reducing the energy consumption of Android mobile phones in correcting specific Android code smells and picture bad practices.
Two research papers focus on Android code smells. Ricardo Pérez-Castillo and Mario Piattini investigated the energy effect of God Class refactoring for Android apps [64]. They found that this correction increases the energy consumption of the mobile, by implementing new methods and classes that increase the memory used by the app. Ding Li and William G. J. Halfond [55] measures the energy saving for IGS and MIM code smells, for a custom Android application. Those evaluations consist in looping 5,000,000 times on a single line of code, which contains a code smell. For this approach, they found that the correction of IGS and MIM consumes respectively 35% and 15% less energy than a code that contains those code smells.
Quite the contrary, we create an approach that evaluate useful Android apps that can be found on the Google Play [9] and the F-Droid [10] stores.

Automated Approaches to Evaluate the Energy Consumption of Bad Practices

Two research papers propose to estimate the energy consumption on a device's emulator, in order to help the developer to build the most energy-efficient app as possible, in real time [65] [66]. Those papers simulated a processing speed and network characteristics, to match the app behavior. In contrast, we based our experiments on a popular Android device, the Nexus 4. The approach to evaluate the energy consumption on a real device involve to focus on stability and accurate results.
Liu Y. and *al.* studied device's sensors usage, using a dynamic tool called GREENDROID [67]. They evaluated GREENDROID on 13 popular android applications.
Linares-Vasquez and *al.* uses scenarios based on user events in order to compute the energy consumption of 55 mobile apps, based on API calls [57]. Results of this research paper is that some good implementation practices like the use of Model-View-Controller, can have a non-negligible bad impact on the energy-consumption of the mobile phone. Also, they suggest to carefully balance the tradeoff between an high maintainability solution and a better energy-efficiency solution.

Pathak, Hu and Ming measured the energy consumption in Android's software components, using the EPROF grained energy tool [68]. EPROF exposes that 65% to 75% of the energy-consumption in free apps is spent in ads, and can reveals several energy bugs like wakelocks, in Android apps.
Lide Zhang and *al.* focused on a automatic power estimator for Android, based on user-reported failures [51]. POWERBOOTER is accurate to within 4.1% on average, instead of our approach that is accurate to less than 1%, using a physical measuring device plugged on the smartphone.
J. Duribreux and *al.* [63] and M. Lineares-Vasquez [57] used an hardware power monitor to measure efficiently the energy consumption for a single Android app, instead of mesuring bias using an Android application. This physical device can measure precisely the energy consumption spent in the smartphone, plugged between the battery and the smartphone.
Gottschalk and *al.* proposed a method to improve energy-efficiency on application level, by applying re-engineering techniques like code analysis and code restructuring [59]. This method is applying on mobile apps and uses loop bugs, dead code, inline methods and cache usage as energy code smells.

Chapter 3

HOT-PEPPER : An Automated Approach to Assess and Improve the Energy Consumption of Android Applications

In this chapter, we introduce Hot-Pepper, an automated approach that evaluate the energy consumption of an Android application. This approach is supported by a framework, using three different tools. Using Hot-Pepper, we are focusing only on the application running on the smartphone, and we actually reducing the harmful effects of noise as much as possible.

3.1 Overview

Hot-Pepper is an approach which allows the Android developer to deploy a most energy-efficient app than the original one. For this to happen, the approach takes as input the source code of the app to detect and correct code smells. Once code smells have been corrected, the approach build an Android version of the app for each code smell corrected. After that, it evaluates each Android version by computing a complex energy metric for those, each metric associated to the corresponding Android version, in order to find the version that consumes the less energy. The approach is summarized in Figure 3.2.
The Hot-Pepper approach is supported by a framework, which uses three main tools :

— Paprika[4], a Java application to statically detect and correct Android code smells,

— Ghost Pepper, a Python software to create a scenario based on user events, for a specific Android application,
— and Naga Viper, a Python software to collect, compute and analyze energy metrics.

First of all, Hot-Pepper uses Paprika to detect and correct Android code smells in the Android app. Resulting to the correction, Paprika returns Android APKs that corresponds to corrected APKs and the original one. Considering that the approach performs the collection and computation of energy metrics in runtime, we have to run the app using a deterministic scenario. Ghost Pepper has been developed to help the developer to implement easily and faster a scenario based on user events. This step is optional, because the developer can upload his own scenario to run the app. Finally, when the scenario is ready to run the app, Naga Viper uses given APKs from Paprika, the scenario based on user events and a smartphone plugged to a physical measuring device to perform the collection and computation of energy metrics. The smartphone has to be plugged to the physical measuring device because Hot-Pepper is intrusive and performs computation on accurate collecting values. Naga Viper collects and computes energy metrics for each APK available to evaluate them, in order to return to the developer a bundle, which contains the best energy-efficient APK, the associated Android source code and a log file concerning the correction.
The whole approach is fully detailed in figures 3.3, 3.4, 3.5 and 3.6.
We extracted 13 specifications to use this framework, listed in figure 3.1. Without these specifications, we don't guarantee accurate and repeatable experiments.
In the next section, we explain how works Hot-Pepper through Paprika, Ghost Pepper and Naga Viper.

3.2 PAPRIKA

To evaluate energy metrics of Android code smells, Hot-Pepper has to produce as much versions of the app as corrected code smells. To analyze an Android app, Hot-Pepper uses Paprika, a static tool analysis for Android apps. This tool offers to detect Android code smells, and to correct them, using the Android APK and the app source code to analyze.

Step 1 : Detect and Correct Android Code Smells

Input : The APK and the source code of the Android app.
Output : Original and corrected versions of the Android app.

1. The smartphone's battery must be removable,
2. battery's pins must be connected, directly or indirectly, to the smartphone,
3. the developer may use an external measuring device, like an ammeter or a voltmeter, to measure the energy consumption of the smartphone,
4. no SIM data usage during the experiments,
5. the luminosity may not to be automatic,
6. no dynamic wallpaper,
7. experiments must be repeated x times,
8. the battery must be charged at 100% before each set of x runs,
9. the smartphone must be shutdown approximately 5 minutes between each set of x runs,
10. if the developer wants to use an automated acceptance testing tool like Calabash, it is advisable to use the USB connection instead of the Wifi connection,
11. if the USB connection to launch the scenario is preferred, the developer has to shutdown the USB charging before each set of x runs,
12. the Wifi has to be shutdown when the application doesn't require it usage,
13. if the Wifi has to be used, the developer must assure that his internet network is stable.

Figure 3.1 – Thirteen specifications to obtain accurate and repeatable experiments using Hot-Pepper

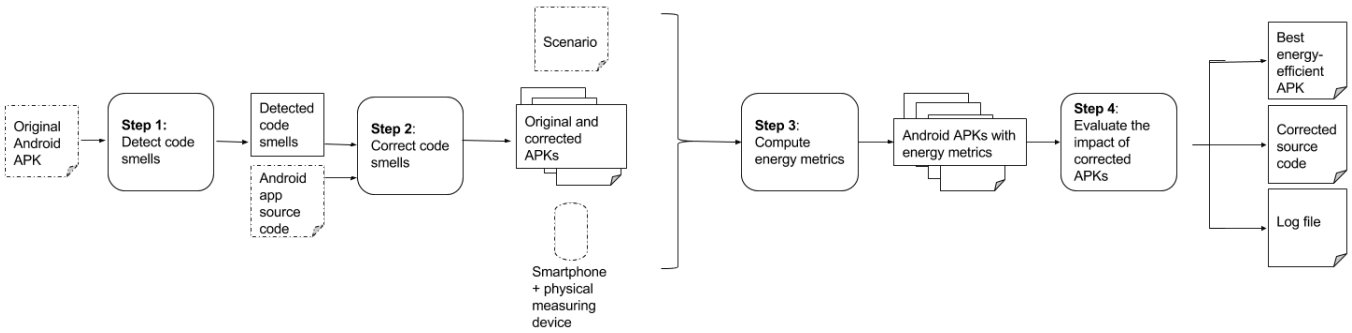

Figure 3.2 – Overview of the Hot-Pepper approach.

Description : PAPRIKA operates the analysis of an Android app in two steps : the detection of Android code smells, and the correction of those detected code smells. The detection of code smells uses the SOOT framework, a Java application that analyze an Android app in order to build a comprehensive representation of the app. Using this representation, PAPRIKA extracts some informations for each code smells type, in order to correct them. Those informations are used by the corrective part. PAPRIKA uses SPOON [69], an open source library to analyze and transform Java source code [1], to correct Android code smells. For each code smell detected, PAPRIKA generates a version of the app without this code smell. Also, the tool generates a version without any detected code smells. At the end of the process, PAPRIKA returns generated Android APKs and the original one. To finish with the corrective part, it has been proved experimentally that the correction of code smells don't change the Android app's behaviour.
This first step of HOT-PEPPER is sketched as figure 3.3.
Implementation : In order to retrieve Android code smells, PAPRIKA uses SOOT to build a comprehensive representation of the Android app. This representation, or model, is directly used by PAPRIKA to transform it into a graph. So, the Android app is represented as a NEO4J [2]'s graph. This graph is modeled as follow : nodes are Android entities, like `App`, `Class`, `Method`, `Attribute`, ..., and edges are relations between those entities. Informations of this graph, like code smells, are retrieved using specified queries with CYPHER [3], a declarative and SQL-inspired language for describing patterns in graphs. Finally, for the detection part, PAPRIKA returns a comprehensive list of informations about Android code smells. These informations can be for example the proportion and the location of each Android code smells type.
For the correction of code smells, this phase takes as input the list of detected code smells and the original source code of the app. Using SPOON, PAPRIKA build a comprehensive abstract syntax tree (AST) of the app. PAPRIKA browses this AST with SPOON processors. Each SPOON processor can visit and transform a source program element, like `Class`, `Method`, `Field`, `Statement`, etc... and is allowed to transform it. Using the list of detected code smells, PAPRIKA starts each processor independently in order to visit each element corresponding to the location of a code smell and correct this one. Finally, PAPRIKA starts in the same time each processors to create an app that doesn't contain detected Android code smells.

1. `http://spoon.gforge.inria.fr`
2. `https://neo4j.com`
3. `https://neo4j.com/developer/cypher-query-language/`

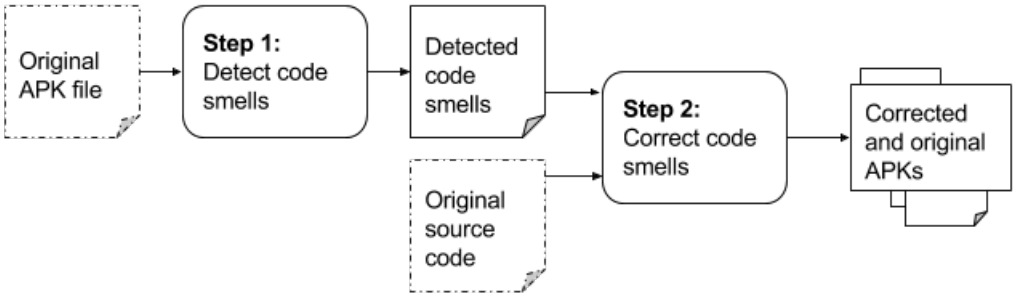

FIGURE 3.3 – Overview of PAPRIKA, the Java application which powered analysis of Android apps for HOT-PEPPER.

3.3 GHOST PEPPER

To measure accurately and automatically the energy consumption in runtime, we propose to create a scenario, based on user events. User events are events that a smartphone user can do, like scrolling, touching, texting, change view, etc...
The developer can propose his own scenario using an automated acceptance testing tool, like CALABASH[70] or ROBOTIUM[71]. Running the application on a scenario created by the developer is the best option, because the developer has the knowledge to explore the energy consumption of his entire application or just a single feature of this one. However, this method can be time consuming, and create a specific scenario for a given Android app can takes many hours or days for refinement and approval. To overcome this problem, we developed a tool, called HOT-PEPPER, which uses ANDROID MONKEY, an Android feature which stress the application by sending a given number of user events.

Step 2 : Elaborate a Scenario based on User Events

Input : None.
Output : A scenario, based on user events.
Description : GHOST PEPPER is an extension to HOT-PEPPER that permits to generate scenarios based on user events, using ANDROID MONKEY. A specification is to generate scenarios that called a lot of code smells, in order to evaluate the energy consumption on interesting scenarios. For GHOST PEPPER, we propose to generate at least two scenarios, and select the best one for the experimentation part. The best scenario of the generated set is the

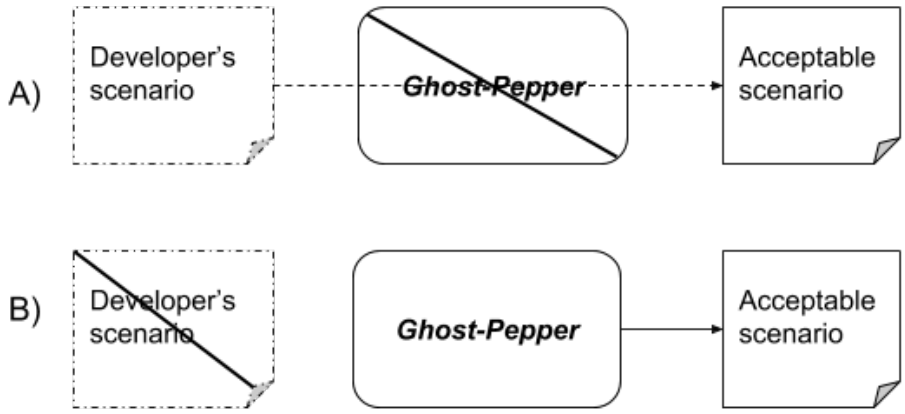

FIGURE 3.4 – Sketch of elaborating a scenario, based on user events. The *option A)* takes as input the developer's scenario, and doesn't use GHOST PEPPER. The *option B)* consists in elaborating a scenario using GHOST-PEPPER, without help from the developer.

scenario that call much code smells than other one, during their executions. This step is sketched as figure 3.4.

Implementation : To implement GHOST PEPPER, we developed in Python a wrapper to communicate easily with ADB[16]. The tool uses this wrapper to send commands to the Android device. Those commands can be to clear logs, to save logs or to launch a new ANDROID MONKEY instance. For each ANDROID MONKEY instance launched, GHOST PEPPER activates the log output. This log output is useful to the process because the tool will obtain app's informations during it execution parsing it, like the activity launched or if a code smell has been called and what type is it. At the end of an instance, we retrieve those informations and count the occurrences of each code smells called and the total number of code smells called. Once all instances have been launched, GHOST PEPPER selects the deterministic seed from ANDROID MONKEY, corresponding the scenario that called the highest number of code smells during the execution.

3.4 NAGA VIPER

One of the contribution using HOT-PEPPER is to evaluate the energy consumption of a given Android app in runtime. For this, we use NAGA VIPER, a Python tool that collect, compute and analyze energy metrics in order to evaluate the impact of corrected APKs.

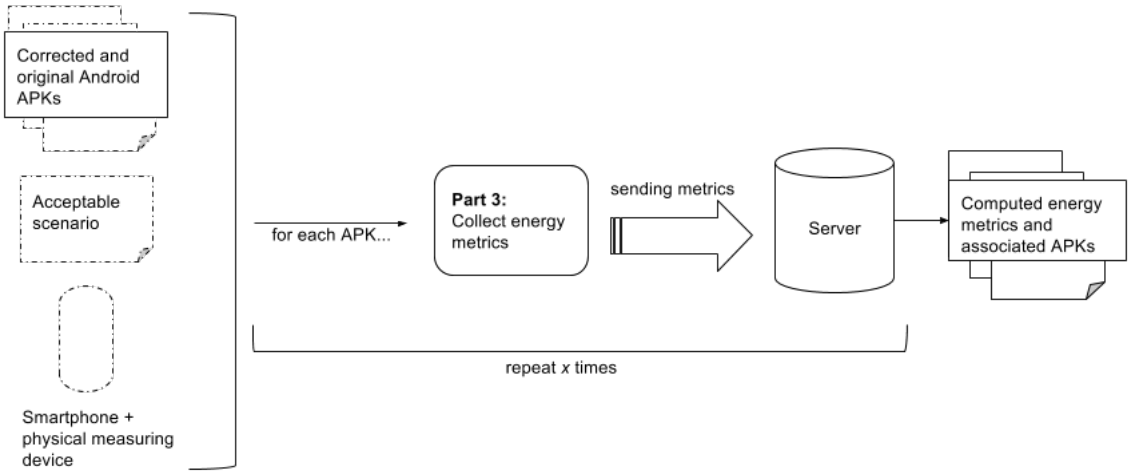

FIGURE 3.5 – Sketch of collecting metrics, using NAGA VIPER.

Step 3 : Compute Energy Metrics

Input : Corrected and original Android APKs, a scenario based on user events, and a smarphone plugged to a physical measuring device.
Output : A file that contains energy metrics for each Android version, associated to corresponding Android APK.
Description : NAGA VIPER is a Python tool that collect, compute and analyze energy metrics. NAGA VIPER is intrusive, so we propose to compute these energy metrics in runtime, on an Android smartphone, via a scenario entirely dedicated to the Android app. 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.
This step is sketched as figure 3.5.
Implementation : The principle energy metric collected is the energy consumption of the battery during tests. This energy metric is collected via the API of an external device, like an ammeter or a voltmeter.
To avoid external interferences and restore the accuracy to our data, we run x times the application, and get as final informations the average ampere value, the average execution time and the average execution time of those x runs.

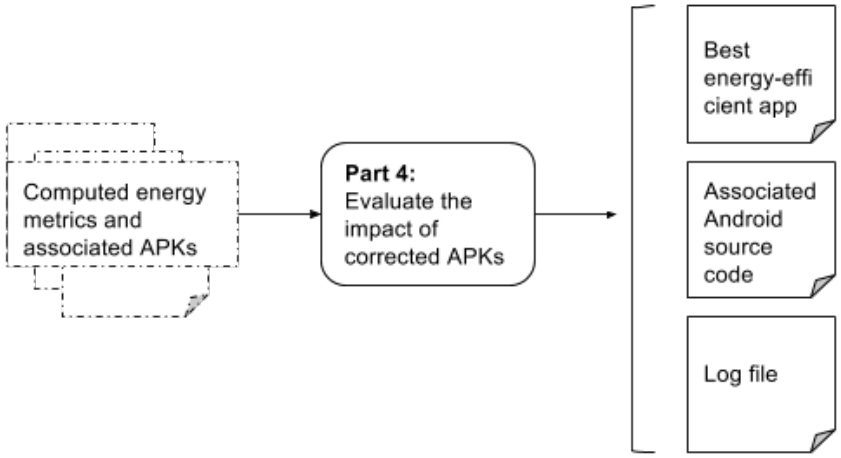

FIGURE 3.6 – Sketch of analyzing energy metrics, using NAGA VIPER.

Step 4 : Evaluate the Impact of Corrected APKs

Input : n informations file, for n version of a given Android application.
Output : The best energy-efficient Android application.
Description : After running n times the step 3, for each version built in step 1, each output from step 3 has to be analyzed.
This part compare the average power consumption, using the average energy consumption and the average execution time, to get the application's version associated to the minimal power consumption, as explained in background.
After compute these power consumption for each version, we confront each of those versions and get the best energy-efficient version. Finally, this version is send to the developer.
This step is sketched as figure 3.6.

Chapter 4

Empirical Study

We propose to measure energy effects of 3 Android performance code smells on 5 popular Android apps, and 3 picture bad practices on 7 versions of a custom Android app. The choice has focused on a custom app to evaluate picture bad practices because we noticed that existing open source Android apps do not have much saved resources. Indeed, most of those apps download from remote servers resources, to display them on screen once downloaded. Considering that we don't have any access to those resources, we can not modify them in order to compare some versions of a unique resource. As the experiment does not require Paprika, we evaluate directly those bad practices using Naga Viper.
We computed the Cliff's Delta effect size to quantify the importance of the difference between the energy consumption of each Android version, for both experiments. Cliff's Delta is a non-parametric effect size measure, which represents the degree of overlap between two sample distributions. It ranges from -1 (if all selected values in the first group are larger than the second group) to $+1$ (if all selected values in the first group are smaller than the second group), and equals 0 when two sample distributions are identical. Each population is an Android version, composed by the n computed average global energy consumption. This is what the Cliff's Delta estimation means, between two populations [5] :

— lower than 0.147, there is no significant difference between these,
— between 0.147 and 0.330, there is a low significant difference between these,
— between 0.330 and 0.474, there is a medium significant difference between these,
— larger than 0.474, there is a large significant difference between these.

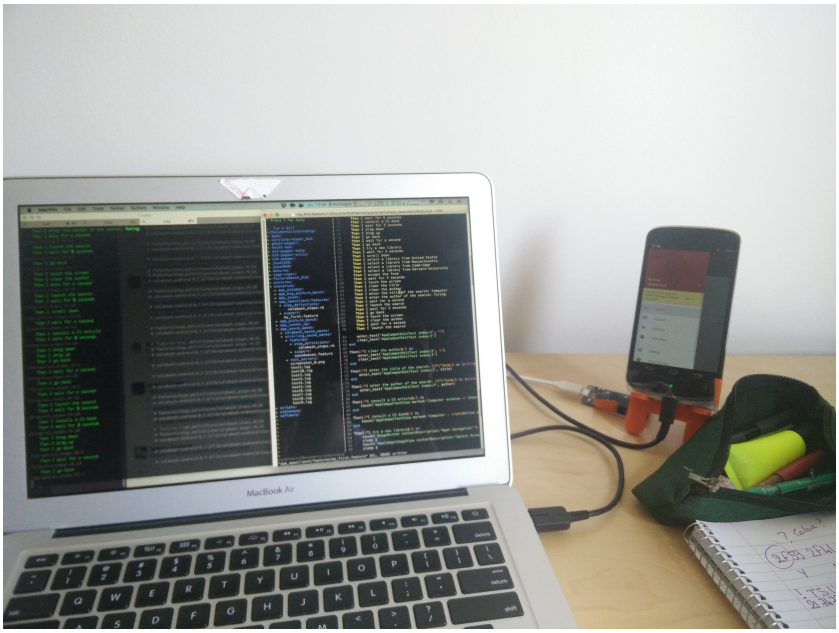

FIGURE 4.1 – A picture of an experiment using the framework Hot-Pepper

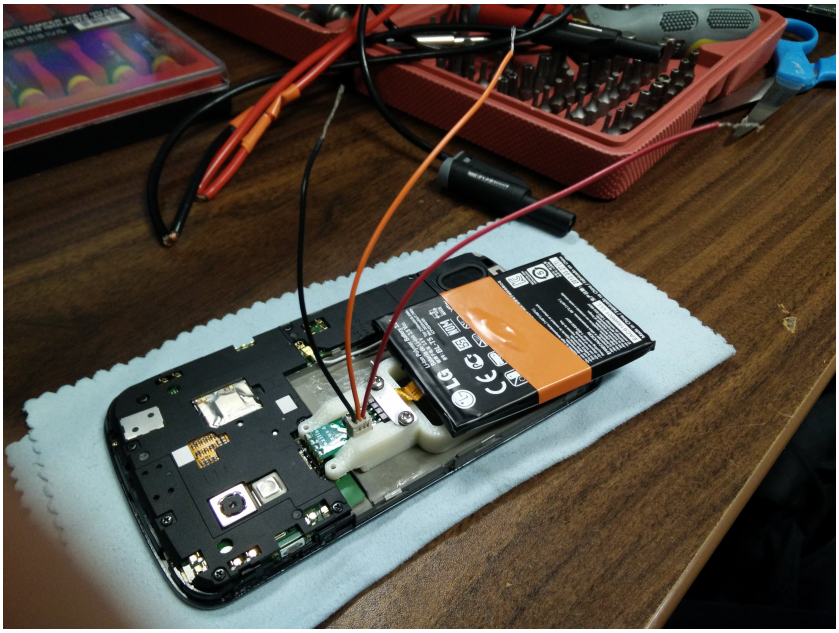

FIGURE 4.2 – A picture of the ammeter, plugged on the battery

Evaluation of Three Android Code Smells

We performed our experiments on Android code smells using 3 of those : HashMap Usage (HMU), Internal Getter/Setter (IGS) and Member Ignoring Methods (MIM). The code smells chosen are respectively listed and explained in Chapter 2. For this experiment, we use `Hot-Pepper`.
To reproduce our results on Android code smells using HOT-PEPPER, Figure 4.3 lists ten specifications to respect.
To experiment our framework on popular Android apps, we had to find open source Android apps that contain those code smells, and respect following criteria :

— two given apps are not listed in the same category,
— the usability of the Android app must be easy to run it using a simple scenario based on user events,
— the app must contain at least two of code smells we currently study.

To find those apps, we used F-Droid, an open source app store that exceeds 2,000 Android applications since June 2016 [72] and the following method. First of all, we developed a Ruby web crawler that extracts data from an XML file that describes the list of Android code smells available through the store. Once those data extracted, we downloaded the last version and associated meta-data of each open source app. After that, we used `Paprika` to detect and list detected code smells in those apps, and put aside apps that contains at least 2 of the three studied code smells. At the state of the method, we have listed 37 apps for 6 different categories : Music, Reader, Productivity, Utility, Sport, and Education. For each category, we get the app that contained the most of code smells and that can be build using the code source associated to the app. To finish, we get 5 apps for 5 different categories :

— `Aizoban` (Reader - version 1.2.5), an online and offline Manga reading application [1],
— `Calculator` (Utility - version 5.1.1), a fork of the official Cyanogen-Mod calculator [2],
— `SoundWaves` (Music - version 0.136.10), a client to manage, fetch and listen to podcasts [3],
— `Todo` (Productivity - version 1.0), an app to create and manage tasks [4],
— and `Web Opac` (Education - version 4.5.9), a client that offers online

1. `http://goo.gl/6mZblY`
2. `http://goo.gl/2vhmCW`
3. `http://goo.gl/nQk7qQ`
4. `http://goo.gl/NXdNMj`

books catalogues of universities around the world [5].
Here, it is important to note that `Calculator` and `Todo` do not required Wifi, unlike the three other Android apps. Once apps have been selected, we developed a scenario based on user events for each of those, using Calabash. Scenarios for these apps are available in the Github space of Hot-Pepper [6]. The major issue for us, in this part, is make sure that the scenario explores the app in order to call each view and each feature at least one time. However, those scenarios does not take care about log in for remote accounts like Google, Facebook, Twitter, etc... After elaborating a scenario, we detected and corrected Android code smells in the app to obtain 5 different versions of each app : the original version of the app, the version without any HMU (V_{HMU}), the version without any IGS (V_{IGS}), the version without any MIM (V_{MIM}) and, finally, the version without the 3 code smells (V_{ALL}). Finally, we launched Naga Viper on each app 20 times, using a Nexus 4 mobile phone combined with an ammeter, the YoctoAmp, to collect via the measuring device's API energy values. Once energy values collected for 20 runs, we computed energy metrics and analyzed them. Also, to be sure that the energy saving for each corrected app is valuable, we use the Cliff's Delta statistical method to deliberate if this one is significant or not.
Static data for each selected app are available in Figure 4.4.
Figure 4.5 represents the energy saving percentage between each corrected app and the original one. Figure 4.6 presents cliff's delta estimations for each evaluated Android version. Based on those figures, we can observe a significant difference of the energy consumption. For `Calculator` and `Todo` apps, V_{ALL} is the best version of those with an energy saving of 1.69% and 4.83% respectively. Unlike those apps, V_{ALL} is visually the second best energy-efficient version for `Aizoban`, `SoundWaves` and `Web Opac`. The best version for `Aizoban` is V_{HMU}, which consumes 2% less than the original version. The best version for `SoundWaves` is V_{IGS}, which consumes 1.43% less than the original version. The best version for `Web Opac` is V_{MIM}, which consumes 3.86% less than the original version.
According to Figures 4.4 and 4.5, we notice that even if the total number of Android code smells detected by Paprika is very high, we can elaborate scenarios that does not execute at least once of type. For example, the number of MIM detected by Paprika for `Aizoban` is 110, but the established scenario for this app does not execute once MIM during the scenario. Using the corrected log file for MIM in Aizoban, we have noted that all MIM are located in libraries and code to log in remote accounts, which we did not care

5. `http://goo.gl/PWrRRy`
6. `https://github.com/SOMCA/hot-pepper-data`

1. Use the Nexus 4 smartphone,
2. use the 4.4.1 version of Android,
3. use the Dalvik virtual machine,
4. use the original battery of the Nexus 4 (limited to 1A),
5. put down the sound as lowest as possible - but not cut off,
6. disable the automatic luminosity,
7. put down the luminosity to the minimum,
8. dismiss the wallpaper to leave the screen totally black,
9. perform 75 measures per second using the YoctoAmp ammeter,
10. perform each set of tests 20 times for each version, to avoid bad effects from the usage of Wifi, bluetooth, etc...

FIGURE 4.3 – Ten specifications to reproduce our experiments using HOTPEPPER

App	APK Size	#sHMU	#sIGS	#sMIM
Aizoban	5.1	39	190	110
Calculator	2.8	0	10	8
SoundWaves	5.5	5	47	14
Todo	0.26	9	3	0
Web Opac	4	48	77	43

FIGURE 4.4 – APK size, in MBytes, and number of code smells detected by PAPRIKA, for each studied Android app.

in the scenario.
Based on Figure 4.6, 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. So, V_{ALL} can be also considered as the best version for the 5 chosen Android apps.
Finally, based on Figure 4.5, we can noted visually that V_{ALL} is the best app for all apps that does not required the Wifi. As V_{ALL} can be considered as the best version for each Android app statically, we can ask ourselves if energy metrics for `Aizoban`, `SoundWaves`, and `Web Opac`, are less accurate than `Calculator` and `Todo`.

App	#Steps	#dHMU	#dIGS	#dMIM	V_{HMU}	V_{IGS}	V_{MIM}	V_{ALL}
Aizoban	169	10	1300	0	**-2.00%**	-1.09%	+0.08%	-1.38%
Calculator	325	0	6122	1350	Null	-0.18%	-0.45%	**-1.69%**
SoundWaves	172	420	8053	6560	-0.38%	**-1.43%**	+0.29%	-1.29%
Todo	248	40	20	0	-2.40%	-2.04%	Null	**-4.83%**
Web Opac	136	6	133	40	-2.06%	-2.08%	**-3.86%**	-3.50%

FIGURE 4.5 – Number of each code smell called and number of steps during the execution of the scenario, and percentage of energy consumption saving for each version of studied Android apps.

App	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	Null
SoundWaves	0.08	0.26	-0.24	0.26	0
Todo	0.66	0.62	Null	0.92	Null
Web Opac	0.43	0.46	0.69	0.60	-0.11

FIGURE 4.6 – Cliff's Delta estimation between each corrected version and the original one of studied Android apps. If the best energy-efficient APK is not V_{ALL}, we want to estimate the Cliff's Delta value between the best one and the V_{ALL} version.

Evaluation of Three Picture Bad Practices

In this part, we propose to evaluate three picture bad practices using `Naga Viper` and 7 versions of a custom Android app. Those versions shared the same body, pictures, events, and scenario. Only pictures properties have been changed between those 7 versions.
The skeleton of the original app is composed of 12 PNG pictures, displayed using a `GridView` view of 3 x 4 pictures, found on the Internet [7]. The original app contains only 2 features : to scroll vertically and to display a picture 3 seconds tapping on it. The scenario to use this app is deterministic and lasts around 2m30s. It just consists in scrolling and clicking on several pictures during this time. This app is called V_{PNG}, because it emulate an app that display PNG pictures. We created a version that contains JPG pictures, instead PNG, using a software that transforms pictures. The quality of each JPG picture is optimal. This version is called V_{JPG}. To evaluate pictures compression, we reduced the size of each JPG file using `JPEGMini` [8], one of the best software on the market to reduce the size of a JPG file without degrading the quality of the picture. The directory that contains PNG pictures is 2.7 Mb, the JPG directory is 1 Mb and the directory of compressed JPG is 367 Kb. This version is called V_{JPGO}. Finally, we evaluated the following bitmap formats : `ARGB_8888` and `RGB_565`. The format `ARGB_4444` has not been retained because this format is now deprecated in Android. Studied formats have been associated to pictures formats PNG and JPGO, to create 4 new apps : $V_{ARGB-PNG}$, $V_{ARGB-JPGO}$, $V_{RGB-PNG}$, and $V_{RGB-JPGO}$.
Based on results in Figure 4.7, we notice visually that the most energy-efficient version is the version that uses `ARGB_8888` and optimized JPG. Also, we notice that using JPG pictures instead of PNG pictures can save a few quantity of energy. For pictures compression, the version using optimized JPG pictures consumes the same value of energy than the original ones. To finish, contrary to the Android performance tips focusing on pictures [43], we found that using `ARGB_8888` as a bitmap format saves more energy than using `RGB_565`. Also, using optimized JPG associated with `RGB_565` is widening the gap with PNG pictures associated with the same bitmap format.
Based on Figure 4.8, we found that the difference between V_{JPG}/V_{JPGO}, $V_{ARGB-PNG}/V_{RGB-JPGO}$ and between $V_{PNG}/V_{RGB-PNG}$ are not significant. For each other version, the difference of energy consumption is significant, and contributes to conclude that $V_{ARGB-JPGO}$ is the most energy-efficient app. Those results indicate also that $V_{ARGB-PNG}$ and $V_{RGB-JPGO}$ are stati-

7. `https://github.com/SOMCA/hot-pepper-data/tree/master/pictures_xps`
8. `http://www.jpegmini.com`

	V_{PNG}	V_{JPG}	V_{JPGO}	$V_{ARGB-PNG}$	$V_{ARGB-JPGO}$	$V_{RGB-PNG}$	$V_{RGB-JPGO}$
E. cons.	0.0525	0.0536	0.0535	0.0487	0.0482	0.0524	0.0490

FIGURE 4.7 – Evaluation of each version of the studied app, using NAGA VIPER, in terms of energy consumption. The unit is Joules.

	V_{JPG}	V_{JPGO}	$V_{ARGB-PNG}$	$V_{ARGB-JPGO}$	$V_{RGB-PNG}$	$V_{RGB-JPGO}$
V_{PNG}	-0.74	-0.72	1	1	0	1
V_{JPG}	-	0.10	1	1	0.69	1
V_{JPGO}	-	-	1	1	0.72	1
$V_{ARGB-PNG}$	-	-	-	0.2	-0.96	0
$V_{ARGB-JPGO}$	-	-	-	-	-0.98	-0.23
$V_{RGB-PNG}$	-	-	-	-	-	0.43

FIGURE 4.8 – Cliff's Delta estimation between each version of the original app.

cally same versions and that $V_{RGB-JPGO}$ can be considers as the second best version app, in terms of energy consumption.

Conclusion & Future Work

To conclude, our experimental study has proved that Android bad practices have a negative impact on the energy consumption of the smartphone. Our contributions have been to develop an approach that evaluate Android bad practices, in order to deploy the best energy-efficient one. Also, we evaluate the energy consumption of 6 Android bad practices whose 3 in 5 open source Android apps.
More personally, this internship has allowed me to improve myself in searching the best and simple solution for an experimental problem and in practicing my English. Also, I had to manage and overcome some teamwork's problems and difficulties to resolve hardware problems, which discouraged me more than once.
I finished my internship in contributing to the publication of a research paper, describing the HOT-PEPPER approach and the empirical study [11].

We can clearly identified four improvements of HOT-PEPPER for future work. First, we could improve the framework by adding execution traces in Android apps, in order to focus especially on collecting energy values that are interesting in our experiments. Also, given that our approach is intrusive and can be rejected by developers who does not want to plug a physical device on their mobile phone, it could be interesting to study non-intrusive methods to compute energy metrics on a smartphone, or to deploy the framework in the cloud. Finally, generating automatic and accurate Android scenarios using GHOST PEPPER could be beneficial to the approach to be less dependent on the developer.

References

[1] Naouel Moha, Yann-Gaël Ghéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur. Decor : A method for the specification and detection of code and design smells. In *IEEE Transactions on Software Engineering*, volume 36, pages 20 – 36, 2010.

[2] Martin Brylski. Android smells catalogue. `http://www.modelrefactoring.org/smell_catalog`, 2013. [Online ; accessed April-2016].

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

[4] Geoffrey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha, and Laurence 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.

[5] Geoffrey Hecht, Naouel Moha, and Romain 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.

[6] Kolin Paul and Tapas Kumar Kundu. Android on mobile devices : An energy perspective. In *Computer and Information Technology*, August 2010.

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

[8] Ana Rodriguez, Mathias Longo, and Alejandro Zunino. Using bad smell-driven code refactorings in mobile applications to reduce battery usage. In *16º Simposio Argentino de Ingeniería de Software*. ASSE, 2015.

[9] Google play store. `https://play.google.com/store/apps?hl=fr`. [Online ; accessed April-2016].

[10] F-droid play store. `https://f-droid.org/`. [Online ; accessed April-2016].

[11] Geoffrey Hecht, Antonin Carette, Mehdi Ait-Younes, Naouel Moha, and Romain Rouvoy. Hot-pepper : an automated approach to assess and improve the energy consumption of android applications. 2016. In progress.

[12] The android os. `https://source.android.com/source/downloading.html`. [Online ; accessed August-2016].

[13] Art and dalvik. `https://source.android.com/devices/tech/dalvik/index.html`. [Online ; accessed August-2016].

[14] Android packages. `http://developer.android.com/reference/packages.html`. [Online ; accessed August-2016].

[15] What is the difference between dvm and jvm ? `http://stackoverflow.com/questions/3446540/what-is-the-difference-between-dvm-and-jvm`. [Online ; accessed August-2016].

[16] Android debug bridge. `https://developer.android.com/studio/command-line/adb.html`. [Online ; accessed August-2016].

[17] Ui/application exerciser monkey. https ://developer.android.com/studio/test/monkey.html. [Online ; accessed June-2016].

[18] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and Jensen C. Understanding code smells in android applications. In *IEEE/ACM International Conference on Mobile Software Engineering and Systems*. IEEE, 2016.

[19] Suryanarayana G., Samarthyam G., and Sharma T. *Refactoring for Software Design Smells : Managing Technical Debt*, volume 11. Elsevier Science, 2014.

[20] M. Fowler. *Refactoring : improving the design of existing code*. Pearson Education India, 1999.

[21] Lanza M. and Marinescu R. *Object-Oriented Metrics in Practice*. Springer-Verlag Berlin Heidelberg, 2006.

[22] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. The evolution and impact of code smells : A case study of two open source systems. In *3rd international symposium on empirical software engineering and measurement*, pages 390–400, October 2009.

[23] Activities - android developers. `https://developer.android.com/guide/components/activities.html`. [Online ; accessed August-2016].

[24] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting antipatterns in android apps. In *Proceedings of the Second*

ACM International Conference on Mobile Software Engineering and Systems, pages 148–149. IEEE Press, 2015.

[25] A. Carette. Réduction de la consommation énergétique des applications mobiles sur Android. Technical report, University of Lille, Department of Informatics, 03 2016.

[26] Reimann J., Brylski M., and Amann. U. A tool-supported quality smell catalogue for android developers. 2014.

[27] Android smells catalogue. `http://www.modelrefactoring.org/smell_catalog/`. [Online ; accessed August-2016].

[28] Geoffrey Hecht, Naouel Moha, Romain Rouvoy, and Javier Gonzalez-Huerta. An empirical analysis of android code smells. 2016. To be published.

[29] Android arraymap. `http://developer.android.com/reference/android/support/v4/util/ArrayMap.html`. [Online ; accessed August-2016].

[30] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 225–236, Austin, TX, US, May 2016.

[31] What optimizations can i expect from dalvik and the android toolchain ? `http://stackoverflow.com/a/4930538`. [Online ; accessed August-2016].

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

[33] Learn when to use jpeg, gif, or png with this graphic. `http://developer.android.com/training/displaying-bitmaps/index.html`. [Online ; accessed August-2016].

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

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

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

[37] 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.

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

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

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

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

[42] Bitmap configuration. https://developer.android.com/reference/android/graphics/Bitmap.Config.html. [Online; accessed August 2016].

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

[44] C. Seo, S. Malek, and N. Medvidovic. An energy consumption framework for distributed java-based systems. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated software engineering (ASE '07)*, pages 421–424. ACM, 2007.

[45] C. Seo, S. Malek, and N. Medvidovic. Estimating the energy consumption in pervasive java-based systems. In *Proceedings of the 2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications*, pages 243–247. IEEE, 2008.

[46] T. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Jdeodorant : Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008*, page 329–331. IEEE, 2008.

[47] Klocwork tool. http://klocwork.com. [Online; accessed April-2016].

[48] Findbugs tool. http://findbugs.sourceforge.net. [Online; accessed April-2016].

[49] Brown W. J., McCormick H. W., Mowbray T. J., and Malveau R. C. *AntiPatterns : refactoring software, architectures, and projects in crisis*. Wiley New York, 1. auflage edition, 1998.

[50] Ding Li, Yingjun Lyu, Jiaping Gui, and William G.J. 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.

[51] 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.

[52] 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.

[53] Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William 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.

[54] A. Hindle, Wilson A., Rasmussen K., Barlow E. J., Campbell J. C., and Romansky 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.

[55] Ding Li and William GJ 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.

[56] Ding Li, Shuai Hao, Halfond Willian G. J., and Ramesh 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.

[57] Mario Linares-Vasquez, Gabriele Bavota, Carlos Bernal-Cardenas, Rocco Oliveto, Massimiliano Di Penta, and Denys 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.

[58] Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul 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.

[59] Marion Gottschalk, Mirco Josefiok, Jan Jelschen, and Andreas Winter. Removing energy code smells with reengineering services. In Ursula Goltz, Marcus A. Magnor, Hans-Jürgen Appelrath, Herbert K. Matthies, Wolf-Tilo Balke, and Lars C. Wolf, editors, *GI-Jahrestagung*, volume 208, pages 441–455, 2012.

[60] Abhijeet Banerjee, Hai-Feng Guo, and Abhik Roychoudhury. Debugging energy-efficiency related field failures in mobile apps. MobileSoft, 2016.

[61] Jiaping Gui, Ding Li, and William G.J. 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.

[62] Niranjan Balasubramanian, Aruna Balasubramanian, and Arun 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.

[63] Julien Duribreux, Romain Rouvoy, and Martin Monperrus. An energy-efficient location provider for daily trips. Technical report, 2014.

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

[65] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering developers to estimate app energy consumption. In *ACM Mobicom*. ACM, August 2012.

[66] Marius Marcu and Dacian 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.

[67] Y. Liu, C. Xu, S. C. Cheung, and J. Lü. Greendroid : Automated diagnosis of energy inefficiency for smartphone applications. In *IEEE Transactions on Software Engineering*, pages 911–940. IEEE, 2014.

[68] Abhinva Pathak, Y. Charlie Hu, and Zhang Ming. 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 – EuroSys'12*, pages 29–42, 2012.

[69] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon : A library for implementing analyses and transformations of java source code. *Software : Practice and Experience*, page na, 2015.

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

[71] Robotium website. https ://github.com/RobotiumTech/robotium. [Online ; accessed June-2016].

[72] Client 0.100 released. https://f-droid.org/posts/client-0-100-released/. [Online ; accessed August-2016].