Analyzing and Classifying Energy Consumption in Android Applications

Marco Couto[a,b], Jácome Cunha[c,b], João Paulo Fernandes[d], Rui Pereira[a,b], João Saraiva[a,b]

[a] *Universidade do Minho, Portugal*
[b] *HASLab / INESC TEC*
[c] *Universidade Nova de Lisboa*
[d] *RELEASE, Universidade da Beira Interior, Portugal*

Abstract

The use of powerful mobile devices, like smartphones, tablets and laptops, is changing the way programmers develop software. While in the past the primary goal to optimize software was the run time optimization, nowadays there is a growing awareness of the need to reduce energy consumption.

This paper presents techniques and tools to detect anomalous energy consumption in Android applications, and to relate it directly with the source code of the application. We present a methodology to classify program execution and program methods according to the energy consumed. Thus, we monitored and analyzed the results of energy consumed by a corpus of Android applications, and we defined greenaware energy thresholds. Such thresholds are used to classify program methods according to energy consumption. Moreover, we manually inspect the source code of both energy efficient and inefficient methods and we identify program features that influence energy consumption.

Keywords: Green Software Computing, Energy-Aware Software, Source Code Analysis

1. Introduction

The software engineering and programming languages research communities have developed advanced and widely-used techniques to improve both programming productivity and program performance. For example, they developed powerful type and modular systems, model-driven software development approaches, integrated development environments that, indeed, improve programming productivity. These communities are also concerned with providing efficient execution models for such programs, by using compiler-specific optimizations (such as, tail recursion elimination), partial evaluation [1], incremental computation [2], just-in-time compilation [3], deforestation and strictification of functional programs [4, 5, 6], for example. Most of those techniques aim at improving performance by reducing both execution time and memory consumption.

While in the previous century computer users were mainly looking for fast computer software, this is nowadays changing with the advent of powerful mobile devices, like laptops, tablets and smartphones. In our mobile-device age, one of the main computing bottlenecks is energy-consumption. In fact, mobile-device manufacturers and their users are as concerned with the performance of their device as they are with battery consumption/lifetime.

This growing concern on energy efficiency may also be associated with the perspective of software developers [7]. Unfortunately, developing energy-aware software is still a difficult task. While programming languages and their programming environments/compilers provide several compiler optimizations [8], debuggers and fault localization tools [9], memory profiler tools [10, 11], testing tools [12, 13, 14], refactoring

Email addresses: `marcocouto@di.uminho.pt` (Marco Couto), `jacome@fct.unl.pt` (Jácome Cunha), `jpf@di.ubi.pt` (João Paulo Fernandes), `ruipereira@di.uminho.pt` (Rui Pereira), `jas@di.uminho.pt` (João Saraiva)

tools [15], benchmark and runtime monitoring frameworks [16], there are no equivalent tools/frameworks to profile/optimize energy consumption.

In this paper we propose a methodology to monitor and detect anomalous energy consumption for the Android ecosystem: a widely used ecosystem for mobile devices. More precisely, our methodology aims at providing Android application developers techniques and tools to analyze and localize in the software source code abnormal energy consumption. We propose a three layer methodology to monitor energy consumption and to related it to the source code of an Android application:

- Firstly, we reuse and adapted the power tutor framework that provides an energy consumption model for the Android ecosystem [17]. This is an open source model that considers all major hardware components of an Android device, which includes cpu, gps, wifi, etc. Although power tutor was developed as a stand alone software tool, we have updated its source code so that it can be used as an API to monitor the energy consumption of other Android applications.
- Secondly, we develop an Android application that automatically instruments the source code of a given Android application the developer wishes to monitor its energy consumption. The instrumentation is performed by embedding calls to the power consumption model API within the source code.
- Thirdly, we use a testing framework for Android applications in order to execute the (previously compiled) instrumented application. For each execution of a test case, we collect the consumed energy, the runtime execution of each test, and the methods call tree.

This methodology *per se* allows the monitoring of energy consumption only. In order to locate abnormal energy consumption in an application, however, we need to compare different executions of those applications. Different program executions have different energy consumption, and we define an energy-wise classification of program executions which considers four levels: *red*, *orange*, *yellow*, and *green*, where *red* is the most energy inefficient program execution and *green* the most efficient one. To define proper energy thresholds for those four categories, we have instrumented and monitored the energy consumption of a corpus of open source Android applications together with their test cases.

Having defined such a classification, we validate our techniques by analyzing in detail two Android applications. According to the automatic (energy-wise) classification of their test case executions, we manually inspect the methods associated with purely *green* and *red* executions. The aim is to identify source code features that do influence energy consumption. Our first results show that green methods usually consists of few lines of loop-free statements and few method calls. On the contrary, red methods are longer methods that include the use/manipulation of recursive data structures (collections or arrays) and the access to databases/HTTP connections.

Finally, we have implemented our methodology in our GreenDroid framework: one tool was developed to automatically instrument the source code of an application whose developer wishes to monitor in terms of energy consumption. A second tool performs the monitoring of the previously instrumented application: it executes it with a given set of test cases and updates our method classification thresholds.

This paper is organized as follows: Section 2 presents the Android power consumption model. We also detail the changes made to the power tutor consumption model so it can be used to monitor power consumption at the source code level, as well as the changes which the framework applies to the application source code. In Section 3 we describe our techniques to automatically classify program executions and method in terms of energy consumption. Section 4 introduces the GreenDroid framework, which automatically instruments Android applications, given their source code, and monitors its energy consumption by executing it with test cases. In Section 5 we analyze in great detail the results produced by our techniques giving two android applications. Moreover, we discuss the source code features of methods associated to green and red executions. Finally sections 6 and 7 present the related work and the conclusions, respectively.

2. Energy Consumption in Android Source Code

Modern programming languages offer their users powerful compilers, which include advanced optimizations, to develop efficient and fast programs. Such languages also offer advanced supporting tools, such as debuggers or execution and memory profilers, so that programmers can easily detect and correct anomalies in the source code of their applications.

We begin this section by briefly discussing the Android power consumption model [17]: This is a statically calibrated model that considers the energy consumption of the main hardware components of a mobile device. Afterwards, we present our approach which uses/adapts the previously defined power consumption model, to be the building block of an energy profiling tool for Android applications. The idea is to offer Android application developers an energy profiling mechanism, very much like the one offered by traditional program profilers [10, 11]. This is to say that we wish to provide a methodology, and respective tool support, that automatically locates, within the source code of the application being developed, the responsible code fragments for abnormal energy consumption. Our methodology consists of the following steps: First, the source code of the monitored application is instrumented with calls to the calibrated power model as displayed in Figure 1.

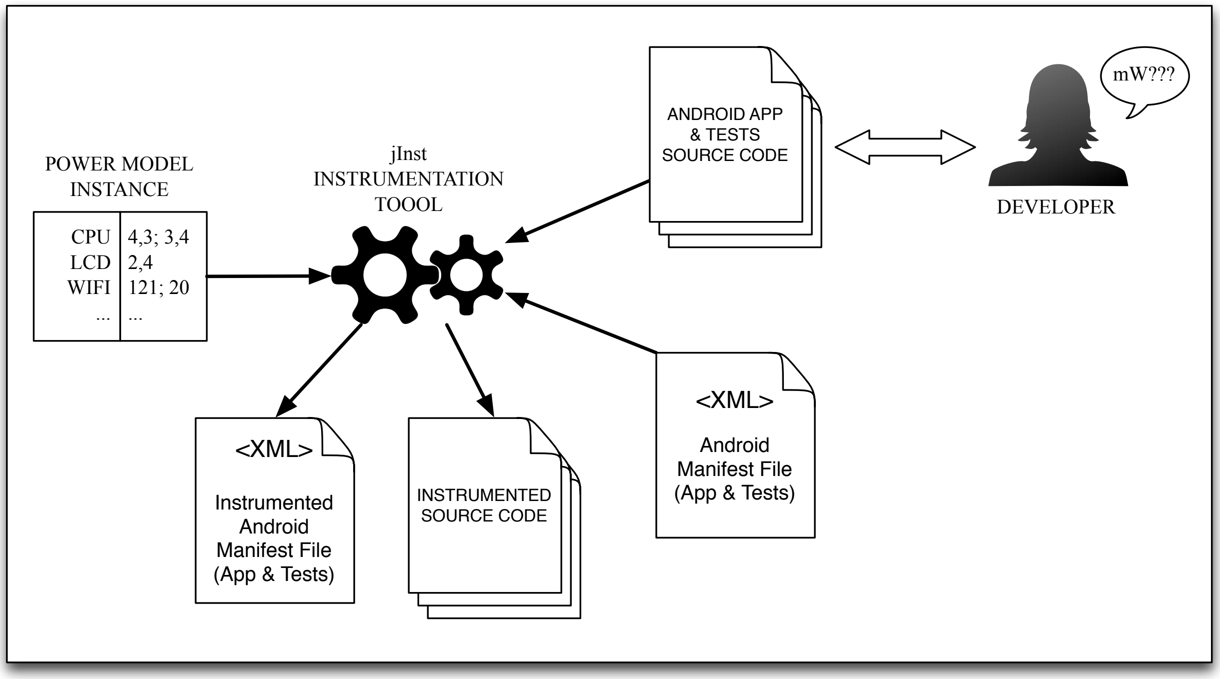
POWER MODEL INSTANCE
CPU 4,3; 3,4
LCD 2,4
WIFI 121; 20
... ...
jInst INSTRUMENTATION TOOOL
ANDROID APP & TESTS SOURCE CODE
mW???
DEVELOPER
<XML> Instrumented Android Manifest File (App & Tests)
INSTRUMENTED SOURCE CODE
<XML> Android Manifest File (App & Tests)

Figure 1: The behavior of the instrumentation tool

After compiling the instrumented version of the source code, the resulting application is executed with a set of test cases. The result of such executions are statistically analyzed in order to determine which packages/methods are responsible for abnormal energy consumption. To instrument the source code with calls to the power model, we need to model it as an API. The final sections describe this automatic source code instrumentation and execution of test cases.

2.1. The Android Power Tutor Consumption Model

We know that different hardware components have different impacts on a mobile device's power consumption. As a consequence, an energy consumption model needs not only to consider the main hardware components of the device, but also its characteristics. Mobile devices are no different from other computer devices: they use different hardware components and computer architectures that have completely different impacts on energy consumption. If we consider the CPU, different mobile devices can use very different CPU architectures (not only varying in computing power, but also, for example, in the number of CPU cores), which can also run at different frequencies. The Android ecosystem was designed to support all different mobile (and non-mobile) devices (ranging from smart-watches to TVs). As a result, a power consumption model for Android needs to consider all the main hardware components and their different states (for example, CPU frequency, percentage of use, etc).

There are several power consumption models for the Android ecosystem [18, 19, 20, 21, 17], that use the hardware characteristics of the device and possible states to provide a power model. Next, we briefly present the power tutor model [17]: a state-of-the-art power model for smartphones [18]. The Power Tutor [17] model currently considers six different hardware components: CPU, Display, GPS, Wi-Fi, 3G and Audio, and different states of such components, as described next.

CPU. : CPU power consumption is strongly influenced by its use and frequency. The processor may run at different frequencies when it is needed, and the percentage of usage can vary between 1 and 100, depending on what is being done; There is a different coefficient of consumption for each available frequency used by the processor. The consumption of this component at a specific time is calculated by multiplying the coefficient associated with the frequency in use with the percentage of utilization.

Display. : The LCD display power model considers only one state: the brightness. There is only one coefficient to be multiplied by the actual brightness level (that has 10 different levels).

GPS. : This component of the power model depends on its mode (active, sleep or off). The number of available satellites or signal strength end up having little influence on the power consumption, so the model has two power coefficients: one to use if the mode is on and another to use if the mode is sleep.

Wi-Fi. : The Wi-Fi interface has four states: *low-power*, *high-power*, *low-transmit* and *high-transmit* (the last two are states that the network briefly enters when transmitting data). If the state of the Wi-Fi interface is *low-power*, the power consumption is constant (coefficient for *low-power* state), but if the state is *high-power* the power consumption depends on the number of packets transmitted/received, the uplink data rate and the uplink channel rate. The coefficient for this state is calculated taking this into account.

3G. : This component of the model depends on the state it is operating on, similar to the Wi-Fi component. The states are *CELL_DCH*, *CELL_FACH* and *IDLE*. The transition between states depends on data to transmit/receive and the inactivity time when in one state. There is a power coefficient for each of the states.

Audio. : The audio consumption is modeled my measuring the power consumption when not in use and when an audio file is playing at different volumes, but the measures indicate that the volume does not interfere with the consumption, so it was neglected. There is only one coefficient to take into account if the audio interface is being used.

2.1.1. Static Model Calibration

In order to determine the power consumption of each Android device's component, the power model needs to be "exercised". That is to say, we need to execute programs and tests that vary the variables of each components state (for example, by setting CPU utilization to highest and lowest values, or by configuring GPS state to extreme values by controlling activity and the visibility of GPS satellites), while measuring the energy consumption of the device. By measuring the power consumption while varying the state of a component, it is possible to determine the values (coefficients) to include in a device's specific instantiation of the model.

A set of *training applications*, that exercise the six hardware components considered on the model, are use to calibrate a specific Android device. Power Tutor uses a static model calibration approach: the training applications are executed in a specific device (which is instrumented in terms of hardware) so that an external energy monitoring device[1] is used to measure the energy consumption. As described in [17] this approach produces a precise model for the device being calibrated. Figure 2 shows the architecture of the calibration of the power model.

[1] A widely used devise is available at `http://www.msoon.com/LabEquipment/PowerMonitor`.

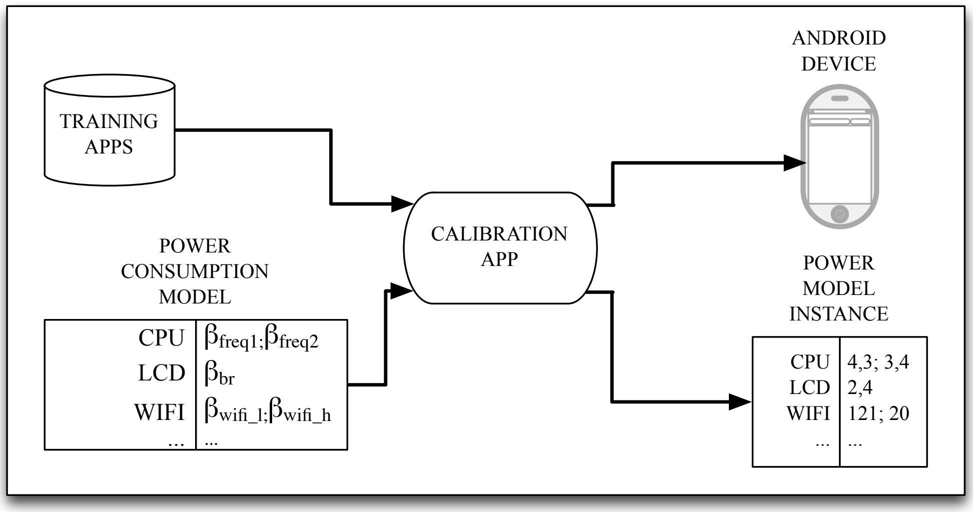

Figure 2: The architecture to calibrate the power model for different devices

2.2. The Model as an API

In order to be able to instrument the source code of an application, with energy profiling mechanisms, we need to adapt the current implementation of power model described in Section 2.1. That power model is implemented as a stand alone tool able to monitor executing applications. Thus, we needed to transform that implementation into an API-based software, so that its methods can be reused/called in the instrumented source code.

To adapt the power tutor implementation, we introduced a new Java class which implements the methods to be used/called by other applications and respective test cases. Those methods work as a link interface between the power consumption model and the applications' source code which is to be monitored.

The methods implemented in the new Java class, called *Estimator*, and which are accessible to other applications are:

- traceMethod(): The implementation of the program trace.
- config(): Performs the initialization of auxiliary variables.
- start(): Starts the energy monitoring thread.
- stop(): Stops the energy monitoring thread and saves the results.

2.3. Source Code Instrumentation

Having updated the implementation of the power model so that its energy profiling methods can be called from other applications, we can now instrument an application source code to call them.

In order to automatically instrument the source code, we need to define the code fragments to monitor. Because we wish to do it automatically, that is by a software tool, we need to precisely define which fragments will be considered. If we consider code fragments which are too small, for example a line in the source code, then the precision of the power model may be drastically affected: a neglected amount of energy would probably be consumed. In fact, there is no tool that we can use capable of giving power consumption estimates at such a fine grained level, with reliable results. On the other hand, we should not consider fragments which are too large, since this will not give a precise indication where an abnormal energy consumption exists in the source code.

We choose to monitor application methods, as they are the logical code unit used by programmers to structure the functionality of their applications. To automatize the instrumentation of the source code of an application we use the JavaParser tool[2]: it provides a simple Java front-end with tool support for parsing and abstract syntax tree construction, traversal and transformation.

We developed a simple instrumentation tool, called jInst, that instruments all methods of all Java classes of a chosen Android application project, together with the classes of an Android test project. jInst injects new code instructions, at the beginning of the method and just before a return instruction (or as the last instruction in methods with no return), as shown in the next code fragment:

```
public class Draw{
...
public void funcA(){
  Estimator.traceMethod("funcA", "Draw", Estimator.BEGIN);
  ...
  Estimator.traceMethod("funcA", "Draw", Estimator.END);
}
```

This code injection allows the final framework to monitor the application, keeping trace of the called methods and consumed energy.

It is important to refer that not only is the Java source code instrumented. Since Android uses XML to define, among other things, the name of the project, the version of the API used, the name of the test runner, etc., we used the standard Java XML parser (DOM parser)[3] as well, in order to edit some necessary definitions, which are:

- The name of the project (both for application and test project): this is needed so if the instrumented projects are opened in Eclipse IDE they do not generate name conflict with the original projects.
- The list of permissions given to the application (for the application project): needed to execute the Power Tutor API.
- The test runner (for the test project): the JUnit test runner needs to be different than the one by default. The next chapter will explain the reason behind this.

So, to summarize, after instrumenting the source code of the application, jInst also edits the Android Manifest file of both the application and test projects.

2.4. Automatic Execution of the Instrumented Application

After compiling the instrumented source code, an Android application is produced. When executing this new application, the energy consumption metrics are produced. In order to automatically execute this application with different inputs, we use the Android testing framework[4] which is based on jUnit.

In order to use the instrumented application and the developed *Estimator* energy class, the application needs to call the start and stop methods before/after every test case is executed. Both jUnit and Android testing framework allow test developers to write a *setUp()* and a *tearDown()* method, which are executed after a test starts and after a test ends, respectively. So, our jInst tool only needs to instrument those methods so we can measure the consumption for each test, as shown in the following example:

```
public class TestA extends ActivityInstrumentationTestCase2<ActivityA>{
...
@Override
public void setUp(){
```

[2]Java parser framework webpage: https://code.google.com/p/javaparser.

[3]More information about DOM parser can be found here: http://docs.oracle.com/javase/tutorial/jaxp/dom/readingXML.html.

[4]Android testing web page: https://developer.android.com/tools/testing/index.html.

```
  Estimator.config("package", android.os.Process.myUid(), this.getContext());
  Estimator.start();
  ...
}
...
@Override
public void tearDown(){
  Estimator.stop();
  ...
}
```

With this approach, we assure that every time a test begins, the *Estimator.start()* method is called. This method starts a thread to collect information from the operating system and then apply the power consumption model to estimate the energy to be consumed. The *Estimator.config()* method is necessary since the energy monitor needs to know the UID of the application being tested, for each test. The *tearDown()* method is responsible for stopping the thread and saving the results.

3. Energy-wise Classification of Tests and Methods

In the previous section, we introduced a technique to measure the amount of energy spent by an Android application test execution. In this section, we will use this information to classify each test energy-wise. We will also introduce a technique to use this information to give some insight about the methods of the application to the programmer.

3.1. Classification of Test Executions

Knowing the exact energy consumption of a method or test is as important as understanding what such consumption means. As with household-electronics, where each new model is compared with its peers, we want to classify a program execution, but in the context of other program executions. For instance, to say that in a test execution the energy consumption is of 200 mW per second does not mean much. But to say that its energy consumption is lower than the energy spent by 90% of the other tests does have a meaning: that such a test is good when compared to others. On the other hand, if its consumption is above 90% of the existing tests, then something is probably wrong.

To automatically classify test runs regarding their energy consumption, we will use a methodology introduced in [22] to classify source code using metrics. This methodology compares each source code metric value, of the software under analysis, to the metric values of other software systems. For instance, it compares the size of a method with the size of the methods used in other systems of a software corpus. Usually, such corpus contains many different systems so it represents a wide variety of applications. This methodology allows the setting of thresholds to classify what a good or bad metric value is. This technique has also been successfully applied to find the thresholds for bad smells in spreadsheets [23]. We will use this technique to find the thresholds for low and high consuming tests. As in the previous applications of this technique, we will compare a given test run against a corpus of several other tests.

In our case we will be using 6 different applications: App Tracker, Catlog, Chord Reader, Connectbot, Google Authenticator, and News Blur. Using the technique presented in the previous section, we run 193 tests on these applications. We present the quantiles of the energy consumption of these test executions, per second, per number of method calls, in Figure 3.

As we can see, and as it happens in [22] and [23], the energy consumption of the test runs follows a power law distribution, with the highest variability in the tail. Thus, we use the same method to define the thresholds, which are at 70%, 80%, and 90%. A test that fits below 70% is considered *energy efficient* and is classified as *green*, between 70% and 80% *yellow*, between 80% and 90% *orange*, and above 90% is considered energy inefficient and is classified as *red* .

In this case, 70% corresponds to an energy consumption of 1.8 mW/s, 80% to 3.4, and 90% to 10.2. Table 1 summarizes this information.

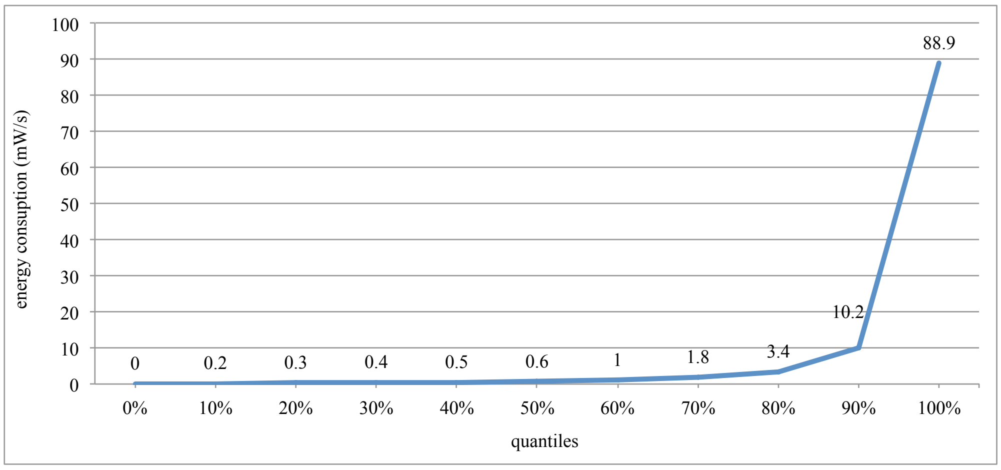

Figure 3: Quantiles of the energy consumption of test executions.

Table 1: Thresholds for energy consumption of test executions.

Quantile	Energy Consumption	Classification
$< 70\%$	< 1.8 mW/s	*green*
70–80%	1.9–3.4 mW/s	*yellow*
80–90%	3.5–10.2 mW/s	*orange*
$> 90\%$	> 10.3 mW/s	*red*

We have applied this methodology to 6 Android applications: AppTracker, Catlog, ChordReader, Connectbot, Google Authenticator and News Blur, open-source Android applications whose test cases are also available. Because this methodology greatly depends on a representative corpus of the programming language being analyzed, our classification should be updated as soon as new Android applications are available and/or analyzed by our tools.

3.2. Methods' Energy Consumption Information

We will now use the gathered information on the execution of the program tests to give some insight about the behavior of each method of an application, to the programmer.

Our main goal is to guide the programmer to find the worst methods in terms of energy consumption. To this end, we devised a visualization of the gathered consumption information of the tests, including which methods were used in each test, and present it to the programmer. For each method, we calculate the number of times it was used in a green test, in a yellow test, in an orange test, and in a red test. This is then represented as a vertical bar chart, where the y-axis represents the percentage of times a method was used in each test type, and the x-axis represents each of the methods of the underlying application. Figure 4 represents this information for one method, getOidFromPkcs8Encoded, in the Connectbot application:

In this case, we can see that this method was involved only in red and green tests; indeed, mostly in red ones. This information alone is probably not too helpful, but when compared to other methods, it gains another importance. In Figure 5 we show all the methods of the Connectbot application:

In this application, we can see that the methods had a tendency to be involved in more red tests, and as such, we can consider this application to be a high consuming one (in fact this is the worst application we used).

A second example is shown in Figure 6; here the application is the **Google Authenticator**.

In this case, we can see that most methods tend to be on the greener side. In fact, this is the best application in the corpus we used.

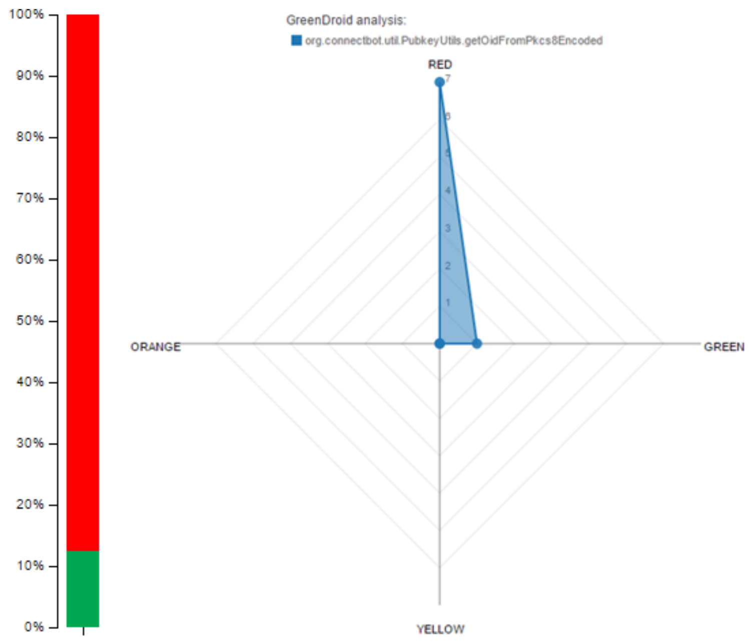

Figure 4: Example of the visualization of a method's energy consumption profile.

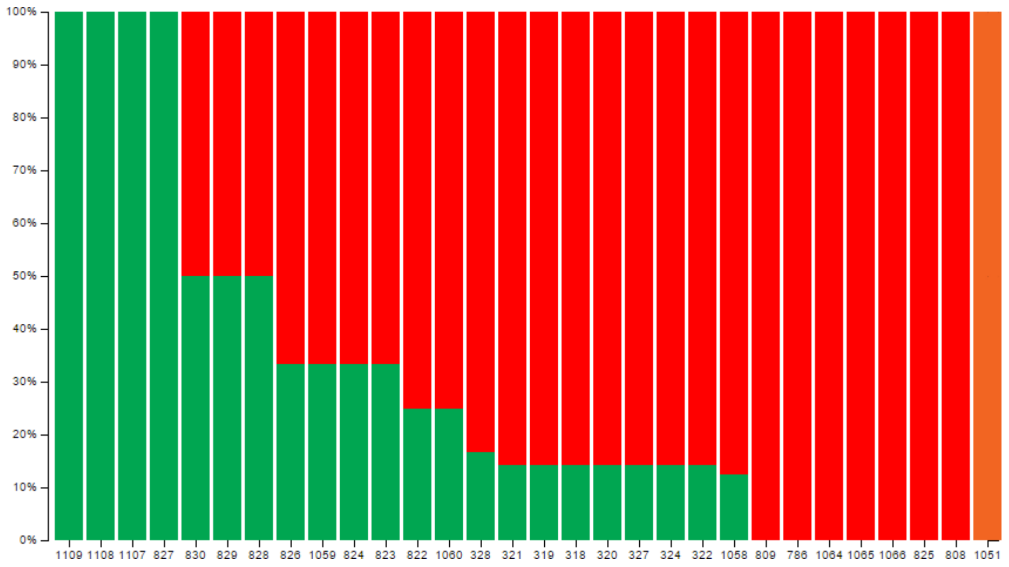

Figure 5: Energy consumption profile of the methods of the Connectbot application.

4. GreenDroid: An Android Framework for Energy Profiling

At this point, we defined how to adapt the consumption model to work as an API, what to instrument in the source code in order to enable the monitoring of both execution trace and consumption, and how tests execution can be used to perform an automatic execution of the application under test.

Nevertheless, this are all independent tasks, and we want to include them all in one final tool that executes them one after another. In this section, we will describe the work flow of that tool, GreenDroid[5],

[5]This tool is available at `https://github.com/greensoftwarelab/GreenDroid`

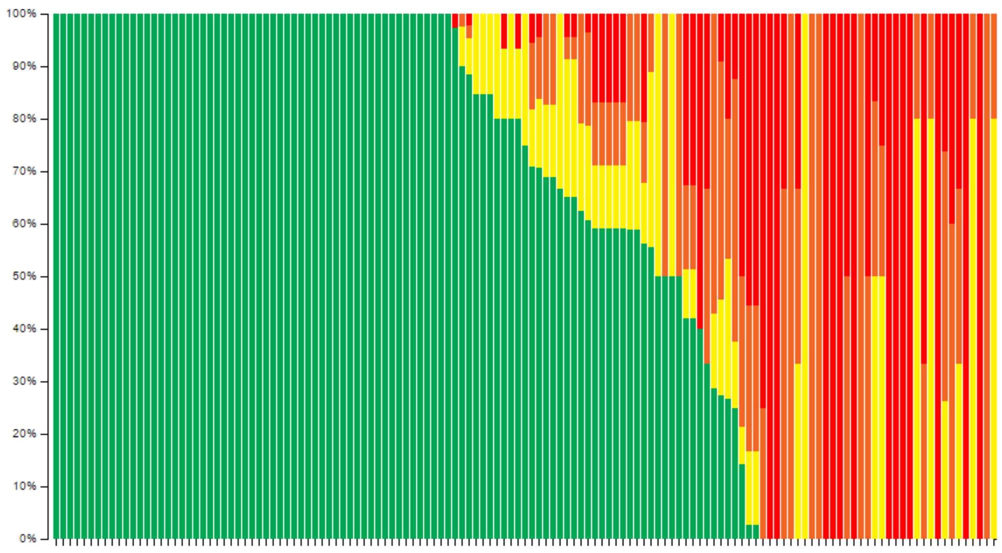

Figure 6: Energy consumption profile of the methods of the Google Authenticator application.

from the first task (instrumentation) to the last one (displaying the results), explaining what each task is expected to use as input and what to generate as output for the next one.

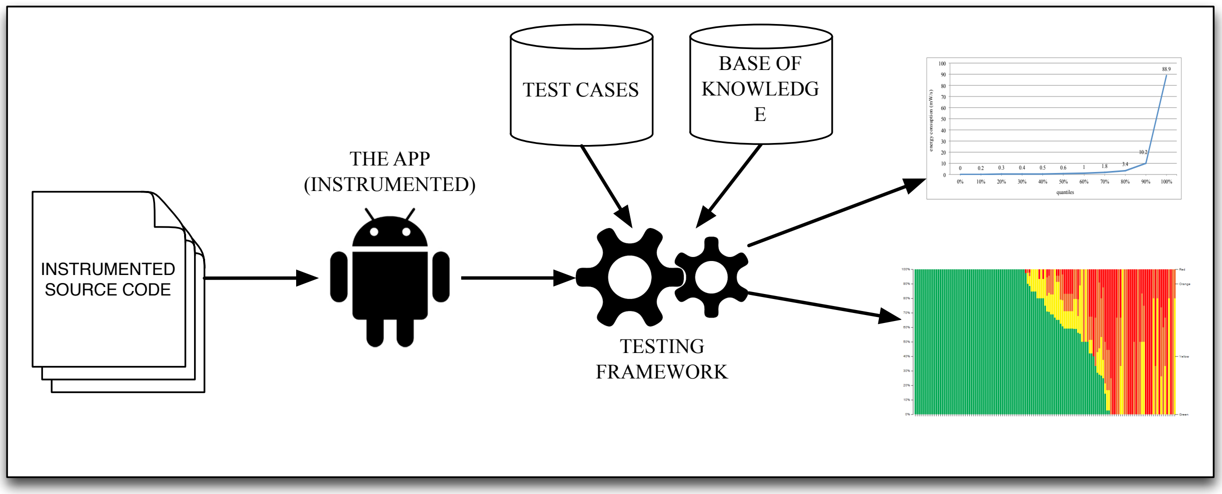

Figure 7: The behavior of the monitoring framework

4.1. Workflow

It is important to refer that GreenDroid expects as input some details about the application to be tested. Besides indicating the name of the tool, it must also be specified the path to both the application project folder and the application tests folder. It is also necessary to specify the package of the application, since this is the unique identifier of it and is needed for the process of compilation, installation and tests execution.

These four attributes that define the application under test will then be used as the starting point of the analysis. Each one of the next items will describe one task performed by GreenDroid and how it relates to the others.

1. *Instrument the source code*:
   - **Expected input**: path to the source code of the application under test and path to the tests.
   - **Expected output**: path to the instrumented version of the application under test and path to instrumented test project.

   This is the starting point for the tool. Using the ideas and techniques defined in Section 2.3, the tool takes the path to the application under test and flags the code as described, injecting the two enunciated method calls. Then, it takes the path to the tests and changes the *setUp()* and *tearDown()* methods (or creates them, if they were not already created), as explained before. The tests that do not fill the requirements (tests that do not use the Android testing framework, and so do not execute in a device) are simply ignored and will not be included in the instrumented test project.
   The tool produces as output in this phase the path of both the instrumented version of the application and the instrumented test project.
2. *Install the application and execute the tests*:
   - **Expected input**: path to the source code of the instrumented application under test (Application Under Test) and path to the instrumented test project.
   - **Expected output**: path to the folder with the testing results (in the device).

   With the source code instrumented, the next step is to install the instrumented application and execute execute the tests that will simulate the execution of the application. They will be executed twice: the first time to get the trace (list of called methods) and the second one to measure power consumption. We need the application trace so we can know exactly what methods are called in each test, and if we get the power consumption at the same time we may have a problem with the tracing overhead: the more methods we trace, the more power consumption we have, because tracing works as a part of the application. To avoid this, we first get the trace (by running tests over the instrumented application), and then we analyze the power consumption (by running the same tests over the original application, not instrumented).
   The tracing results will be saved in files (one for each test), containing a list of the methods called. We also saved the number of times each method was called. Although we did not consider this information t classify the methods, we can adapt our methodology later, and this information seems relevant.
   The execution time is also needed, and for that we used a different JUnit test runner than the usual Android Test Runner. This new runner is called Android JUnit Report Test Runner[6]. With this test runner we were able to generate a XML file containing the JUnit information for each test, and from that we took the execution times of each test.
   The total value of the energy consumed, in mW, is saved in a single file, where each line is a mapping between a test and the value of the energy consumed.
3. Pull files from the device
   - **Expected input**: path to the folder with the testing results (in the device).
   - **Expected output**: path to the folder with the testing results (locally).

   All the information referring to test execution is obviously stored in the device. After the test execution phase, they need to be pulled out from the device in order to be properly analyzed and computed. Android SDK offers a tool that can easily do this task if we passed it two arguments: the source folder in the device you wish to pull out, and the destination folder in your computer. We included an instruction in our tool that simply does the invocation of the command as if it was invoked from a shell, and it does all necessary work associated with this task.
4. *Classify the tests*:
   - **Expected input**: path to the folder with the testing results (locally).

[6]More information and tutorials can be found at `http://www.alittlemadness.com/2010/07/14/Android-testing-xml-reports-for-continuous-integration/`.

- **Expected output**: list of the tests from the application under test classified.

At this point, the tool will analyze the files previously pulled out from the device.
In first place, it creates a list containing the tests executed. Each entry in that list contains the information of the corresponding test, i.e. the methods traced, the execution time and the total power consumption. Then, with this information it calculates the energy consumed (mW per second per number of method calls), which is the standard comparison reference for the tests. As we already referred, this value is saved in a base of knowledge that contains the consumption per second per number of method calls of every test of every application analyzed so far.
Considering the classification approach described in Section 3, the tests are then classified, according to their consumption, execution time and number of method calls, as Red, Orange, Yellow or Green.

5. *Display the results*:
   - **Expected input**: list of the tests form the application classified.
   - **Expected output**: a radar diagram for each method.

   The last task is responsible for generating two visual representations of the results. First, a bar chart indicating the percentage of Red, Orange, Yellow or Green tests in which a method was called, where each bar corresponds to a method, and is divided in 4 sections (the four test types). This representation gives information about the general quality of the methods, and an example of it was already shown in Figure 5 and 6. Second, the tools generates a diagram that displays the information about the involvement of each method in the tests of the application. This means that the method has associated to it the number of times that it was called in a test with a particular classification. We choose to display this information in a radar diagram (as seen before in Figure 4), where each axis represents one of the four classification for the tests. The ideal scenario for a method would be to have a single line in the Green axis (meaning that it was only called when the consumption was not considered high, but completely normal.

5. Results

In this section we analyze in detail the results produced by our techniques when analyzing and automatically classify (energy-wise) two Android applications. To validate these results, we manually inspect the methods involved in green/red executions and we describe which source code characteristics may positively/negatively influence energy consumption.

5.1. Case study

As a case study we compare the results generated by our GreenDroid tool using two of the six applications analyzed. We selected the applications Google Authenticator and Connectbot, since they are the ones with the greatest number of tests. In order to better understand the differences between both applications, we first performed a statistical analysis considering the consumption, number of called methods and execution time per test, of both applications. This analysis is shown in Table 2 for Google Authenticator and Table 3 for Connectbot.

Table 2: Statistical analysis for the Google Authenticator application.

	Consumption (mW)	Number of calls	Execution time (s)
Average	162,71	228,88	0,58
Standard Deviation	197,61	93,93	0,56
Maximum	1333,00	812,00	2,84
Minimum	17,00	10,00	0,02

As we can see in these tables, apparently Connectbot is the one with more probability to have problematic methods, between the two applications. It has much higher values for the maximum and average consumption, and the column regarding execution times is not so different.

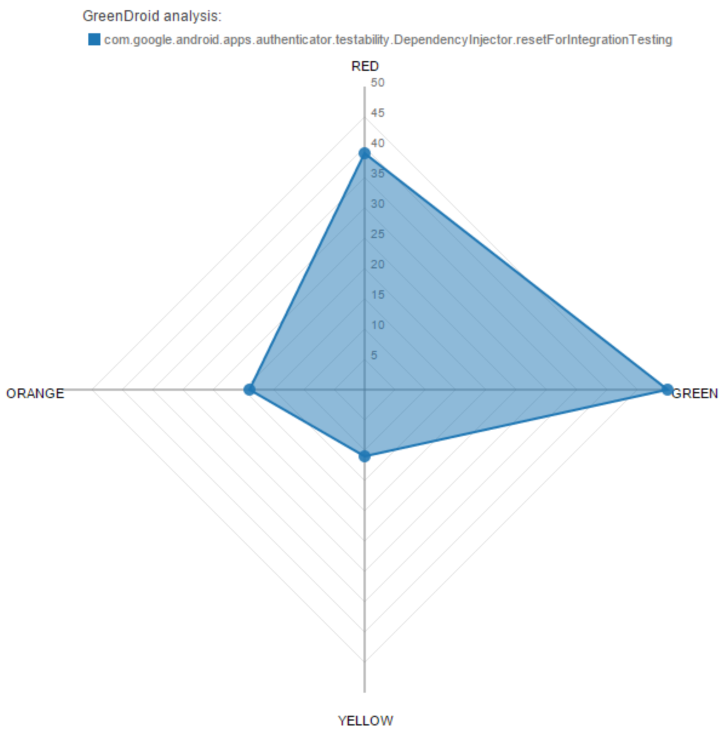

Figure 8: A radar diagram displaying the results of one method

Table 3: Statistical analysis for the Connectbot application.

	Consumption (mW)	Number of calls	Execution time (s)
Average	919,53	228,88	0,17
Standard Deviation	765,14	213,03	0,58
Maximum	3555,00	1041,00	3,24
Minimum	16,00	15,00	0,02

The first thing we tried to understand at this point is how different the global classification results are for both applications, i.e., verify if Connectbot actually has more tests classified as Red when compared with Google Authenticator, and if these red tests contain few or many associated methods.

Our tool first generates a normalized stacked bar chart, for each project, where each bar represents one analyzed method and the percentage of Red, Orange, Yellow and Green tests when it was called. This gives a clear view of the influence of an application and its methods in the consumption classified as anomalous. If there is one application with a clear high red area, it means that it is most likely composed of methods which are leading to anomalous consumption. We generated the results for our tested applications, and obtained the graph represented in Figure 5 for Connectbot and the graph for Google Authenticator represented in Figure 6.

The results for Connectbot are according to our expectations. We see that there are several methods with very short green bars and very long red bars, which indicates they are most likely the less energy efficient ones. Additionally, there are only three that have at least 50% of their calls associated with green methods, which are most likely methods which are more energy efficient than the others in the applications. Regardless, we have three particular methods (the three left-most bars) that have a 100% green bar, so this either means that they were called several times, where none of them were related to an anomalous consumption test, or that they were rarely called in the Red tests and always in Green tests.

Regarding Google Authenticator, we expected to have better results with a wider green area but did not anticipate how wide that area would be. With the obtained results, where we can see a very wide green area, it is fair to conclude that the majority of this application's tests were not related to anomalous consumption. Nevertheless, we also have indication that, between the analyzed methods, there are some with a strong evident relation with Red and Orange tests (situated in the right half of the chart).

Another feature that our tool provides is a more detailed view of the method that the programmer wants to analyze. For a selected method, we display a radar chart showing exactly the number of times that the method was called. We again chose to analyze one situation for each of the applications, and so we selected one method for each. The selection criteria was simple: we wanted to know the detailed information of one method of each application with the biggest number of calls in Red tests. Figure 9 shows the results of the resetForIntegrationTesting method (from Google Authenticator) and Figure 10 shows the radar graph generated for a method called getOidFromPkcs8Encoded (from Connectbot). Comparing both methods we can understand why the majority of the methods from Connectbot had higher red areas, and the methods from Google Authenticator were considered more energy efficient.

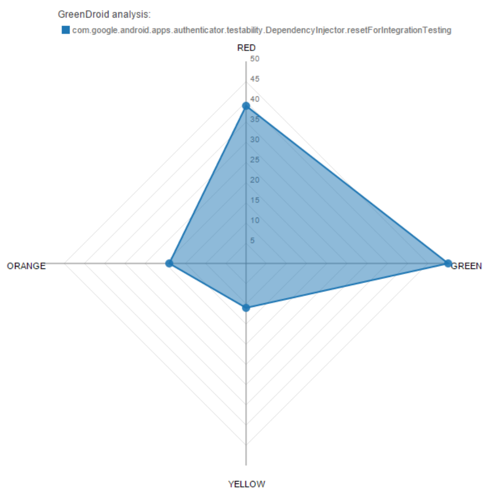

Figure 9: Green, Yellow, Orange and Red test calls of the method resetForIntegrationTesting (Google Authenticator)

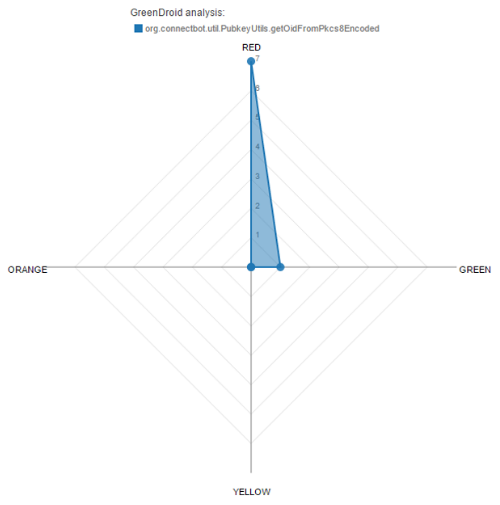

Figure 10: Green, Yellow, Orange and Red test calls of the method getOidFromPkcs8Encoded (Connectbot)

Another feature we thought would be interesting to include in our tool was a way to visually compare methods from the same applications. For the programmer, it would be interesting to not only know if a method has more calls in Red or Green tests, but also how good is a method when compared to others. For example, if a method **A** has 6 calls in total, 3 of which are in Red tests, and a method **B** has 10 calls where 2 of are in Red tests and the other 3 are in Orange tests, then perhaps it is wiser to improve method **B** since it is called more times and half of its calls are in tests with anomalous consumption. Figure 11 shows an example of a comparison between three methods of the Connectbot application.

In this particular case, it might be useful to focus the attention in optimizing the onCreate method (blue area), since it appears to be the one with the worst ratio (higher values for Red and Orange, and smaller values for Yellow and Green), although it is the one with higher number of calls (larger area). The method with higher value in the Red axis (close) is also the one with the higher values in the Orange, Yellow and Green axis, so the ratio is not as bad as the previous one. The deleteAllData method is apparently the less problematic one, since around 50% of its calls were in Green tests.

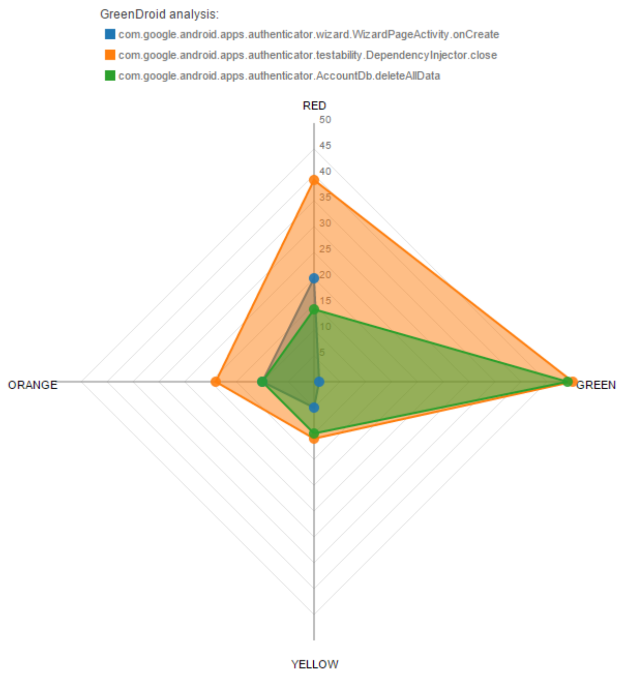

Figure 11: Comparison between 3 methods from Connectbot

5.2. Empirical Study of Energy-wise Methods Features

In order to validate the results presented in Section 5.1, we manually inspect the source code of the two applications to identify program characteristics hich influence the energy consumption of the methods. With this analysis, we aimed to understand if there were similarities among methods mostly called in Red and/or Orange tests, and methods mostly called in Green and Yellow tests. We also aimed to find differences between methods called mostly in different types of program tests.

For this study, we considered 30 methods of each application: 15 of them were the ones with more calls in Red and Orange tests (which we will refer to as +Red methods), and the remaining 15 were the ones with more calls in Green and Yellow tests (which will be called for now +Green methods). This makes a total of 60 methods analyzed.

We were interested in investigating if there was a relation between certain software metrics/code characteristics (or a set of metrics) and the +Red/+Green methods. We started by investigating with a common set of metrics, such as number of parameters, or cyclomatic complexity. Unfortunately, we could not find any relation between such metrics and the energy consumption of the methods. We then manually inspected the code and collected a set of metrics and characteristics that we believe form a pattern. For the purpose of this analysis, we considered the following set of metrics/characteristics:

- Number of instructions;
- Method contains loops;
- Number of calls to other methods;
- Method contains operations over databases;
- Method contains operations over HTTP connections;
- Method contains operations over data structures;
- Method is the static, synchronized, or final;

- Return type;

Table 4 and Table 5 show the results of the analysis for the 30 +Red and +Green methods, respectively.

Table 4: Metrics analysis for the +Red methods

Method	**Green**	**Yellow**	**Orange**	**Red**	**#Instructions**	**Loops?**	**#Calls**	**DB operations?**	**HTTP?**	**Data Structure operations?**	**(St)atic/(Fi)nal/ (Sy)ncronhized**	**Return**
1	50	11	19	39	18		3		x		St/Sy	String
2	50	11	19	39	9		7	x			St/Sy	int
3	1	5	10	20	40		17					byte
4	1	5	10	20	1		1					byte[]
5	49	10	10	18	4		0				St/Sy	Object
6	49	10	10	14	2	x	1	x			St	int
7	49	10	10	14	12		6	x		x		byte[]
8	49	10	10	14	13		3	x				void
9	49	10	10	14	3	x	2	x			St	String
10	49	10	10	14	7		2	x			St/Sy	boolean
11	2	1	4	7	10		4					int
12	0	5	9	5	4		4					void
13	48	9	8	3	1		1				St	Object
14	39	6	7	3	15		8			x		void
15	4	4	4	3	5		3					void
16	1	0	0	7	14		7				St	String
17	1	0	0	6	1		0					int
18	1	0	0	6	4		0			x		byte
19	1	0	0	6	7		1			x		byte[]
20	1	0	0	6	9		5			x		Object
21	1	0	0	6	14	x	2					int
22	1	0	0	6	8		4			x		byte[]
23	1	0	0	6	3		0			x		void
24	1	0	0	5	31	x	11			x		String
25	0	0	0	4	28		15					boolean
26	0	0	0	4	8		4					int
27	1	0	0	3	2		0					void
28	1	0	0	3	18		14			x	St	Object
29	0	0	0	2	1		0					void
30	0	0	0	2	7	x	1			x		void

As we can see from the tables, there are some patterns which can be identified in the +Red methods and others in the +Green methods. Moreover, we can also see that there are some similarities among the majority of the +Green methods which are not verified in the majority of the +Red methods, and vice versa.

Considering the previously referred metrics, we identified a set of similarities that we believe are the reason why methods are considered +Red or +Green.

Regarding +Green methods, we can observe that:

- Only 2/30 have loops;
- 11/30 never call other methods;
- 23/30 are void or return values from native types (int, boolean or long);
- 25/30 do not have operations over data structures, databases or HTTP connections;
- The average number of instructions per method is 5;
- The average number of method calls per method is 2.7.

Regarding +Red methods, we can observe that:

- Only 5/30 have loops;
- 16/30 have operations over data structures, databases or HTTP connections;
- 12/30 are methods with return type Object or data structures;
- 23/30 have calls to methods external to the application (libraries or Android API), or methods from the application usually among the +Red ones.

Table 5: Metrics analysis for the +Green methods

Method	Green	Yellow	Orange	Red	#Instructions	Loops?	#Calls	DB operations?	HTTP?	Data Structure operations?	(St)atic/(Fi)nal/ (Sy)ncronhized	Return
1	36	0	0	0	17		4					void
2	36	0	0	0	6	x	2					void
3	36	0	0	0	1		1					Object
4	36	0	0	0	11		3				St/Sy	Object
5	36	0	0	0	4		3				St/Sy	void
6	36	0	0	0	4		1				St/Sy	Object
7	36	0	0	0	3		1					void
8	36	0	0	0	3		1					void
9	36	0	0	0	4		3					void
10	36	0	0	0	11		5					void
11	36	0	0	0	4		4					void
12	36	0	0	0	4		1					void
13	38	0	0	0	7		1					long
14	38	0	0	0	1		0				St/Fi	long
15	38	0	0	0	1		0				St/Fi	long
16	1	0	0	2	6		0					int
17	1	0	0	2	1		0					void
18	1	0	0	2	2		0					void
19	1	0	0	2	5		6				St	Object
20	0	0	0	1	15		11					void
21	0	0	0	1	1		0			x		void
22	0	0	0	1	1		0			x		void
23	1	0	0	1	4		0					void
24	1	0	0	1	4		0					void
25	1	0	0	1	4		0					void
26	0	0	1	0	1		1					byte[]
27	1	0	0	0	4		0					void
28	1	0	0	0	8		15			x	St	Object[]
29	1	0	0	0	8		15			x	St	Object[]
30	1	0	0	0	6	x	3			x		void

- The average number of instructions per method is 10;
- The average number of method calls per method is 4.2.

From these observations, we can see a relation between methods with complex operations (such as connections to databases or data structure manipulation) and anomalous power consumption. Calls to external methods also make a method less energy efficient. Moreover, analysis done to the number of instructions per method seem to indicate that methods with a higher number of instructions are prone to be associated with higher consumption, as the average number of instructions and calls to other methods is higher for +Red. The return type also seems to play an important role in the energy efficiency. About 76% of the +Green methods have a void return type or return native values, while 40% of the +Red return objects or data structures.

While these results are somewhat expected, they also support our approach as it correctly signals +Red and +Green methods.

6. Related Work

In the past few years, research on the power consumption of *smartphones* has been increasing. Several research works indicate that power consumption modeling and energy-aware software are becoming important and gaining much interest. It is possible to find different tools designed to estimate the required energy for a *smartphone* application. The majority of these tools, however, focus on Android based *smartphones*, mostly because it is an open source OS[7] and statistics reveal that the sale of Android devices are much higher than any other *smartphone*[8]. In fact, in the second quarter of 2013 almost 80% of the market share belonged to Android devices.

[7]An Android overview can be found at `http://www.openhandsetalliance.com/android_overview.html`.

[8]Information about global smartphone shipments can be found at `http://techcrunch.com/2013/08/07/android-nears-80-market-share-in-global-smartphone-shipments-as-ios-and-blackberry-share-slides-per-idc`.

As previously mentioned and described, Power Tutor [17] was our starting point, as it was for many other research works. For example, DevScope [21] is a tool which creates a power consumption model relating the different hardware components of a device to its different states and consequent power consumption values. This model is used by AppScope [20] to estimate the energy consumption of an application, and by UserScope [24] to create a user-specific profiler for a *smartphone*. However, instead of an independent Android application to create the power consumption model for these tools, they use a Linux kernel module. Additionally, these tools are neither an API library, nor are they open-source.

ADEL (Automatic Detector of Energy Leaks) [25] uses an external power consumption meter to detect unnecessary network communication by tracing the indirect use of received data. However this, and other examples of works based on power consumption models [19, 26, 27], are not as powerful as the previously mentioned ones. SEMO [28] is an energy monitoring system and application for Android smartphones which profiles application energy usage based only on the battery discharge level, and unfortunately produces less reliable results due to this.

Corral et al. [29] measured the amount of energy required to complete different tasks in Android applications using Java, Native C, and Regular C and report in what cases it is advisable to reallocate the job from the Java environment to an external execution environment. Noureddine et al. [30] introduced a software framework which infers the energy consumption model of software libraries from measuring the execution traces through a large number of tests. They also support that these models can be used for energy classifications, but they themselves do not classify the applications methods, and the energy values are CPU based only. These works demonstrate that it is possible to have different values on energy consumption for different software designed to do the same tasks. So this can be a very good indicator that helping developers choose the most energy-aware solution for a software implementation is of great importance. In fact, this has been demonstrated in [7].

The closet work to our paper is Hao' and colleagues'work with *eCalc* [31]. They too estimate Android application's energy through the execution of software artifacts with a series of test cases, alongside previously created power consumption models/CPU profiler. Unfortunately, these models only define the cost functions at the instruction level, and the application itself is not publicly available. Additionally, while *eCalc* only predicts the energy value and returns that same value, we take this a step further. We visually present our estimated values to the developer, showing which are the most critical methods in their code and classifying this information in an easy to understand format.

This paper builds on our previous work [32] where we presented a simpler method classification algorithm: a method is considered as having an abnormal energy consumption whenever it is called in a program execution which consumes more energy than the average of all monitored runs of that program. As a consequence, that approach compares energy of the different executions, but only of the same application.

7. Conclusions

This paper presented a methodology for monitoring, analyzing and classifying energy consumption for Android applications. To monitor energy consumption we have extended the power tutor energy model so that it can be (re)used as an API. We have also developed the GreenDroid tool, which given the source code of an Android application automatically instruments it with calls to the power tutor API. This instrumented version of the application is then executed with test cases to monitor its energy consumption.

We have also presented a methodology to classify (energy-wise) different executions of the instrumented Android applications. Using the GreenDroid tools, we have instrumented and analyzed the energy consumption of a corpus of open source Android applications. Thus, we have defined energy thresholds to classify program executions in four greenware categories. Moreover, we presented the results of analyzing and manually inspecting the source code of the applications' methods associated with the more energy efficient/inefficient executions. Our first results show that methods consisting of few lines of loop-free code are associated with green executions, while methods that manipulate collections or arrays and access databases/HTTP connections are associated to red executions.

These results show that our methodology using our (energy-wise) classification of android applications produces good results, and thus, can be of great help in the development of greener software.

Acknowledgments

This work is integrated in the project GreenSSCM - Green Software for Space Missions Control, a project financed by the Innovation Agency, SA, Northern Regional Operational Programme, Financial Incentive Grant Agreement under the Incentive Research and Development System, Project No. 38973. The last author is supported by CAPES through a *Programa Professor Visitante do Exterior (PVE)* grant.

References

[1] N. D. Jones, An introduction to partial evaluation, ACM Comput. Surv. 28 (3) (1996) 480–503. doi:10.1145/243439.243447. URL `http://doi.acm.org/10.1145/243439.243447`

[2] U. A. Acar, G. E. Blelloch, R. Harper, Adaptive functional programming, ACM Trans. Program. Lang. Syst. 28 (6) (2006) 990–1034. doi:10.1145/1186632.1186634.
URL `http://doi.acm.org/10.1145/1186632.1186634`

[3] A. Krall, Efficient javavm just-in-time compilation, in: International Conference on Parallel Architectures and Compilation Techniques, 1998, pp. 205–212.

[4] P. Wadler, Deforestation: transforming programs to eliminate trees, Theoretical Computer Science 73 (1990) 231–248.

[5] J. Saraiva, D. Swierstra, Data Structure Free Compilation, in: Stefan Jähnichen (Ed.), 8th International Conference on Compiler Construction, CC/ETAPS'99, Vol. 1575 of LNCS, 1999, pp. 1–16.

[6] J. P. Fernandes, J. Saraiva, D. Seidel, J. Voigtländer, Strictification of circular programs, in: Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM '11, ACM, 2011, pp. 131–140. doi:10.1145/1929501.1929526.
URL `http://doi.acm.org/10.1145/1929501.1929526`

[7] G. Pinto, F. Castor, Y. D. Liu, Mining questions about software energy consumption, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 22–31. doi:10.1145/2597073.2597110.
URL `http://doi.acm.org/10.1145/2597073.2597110`

[8] D. A. Padua, M. J. Wolfe, Advanced compiler optimizations for supercomputers, Commun. ACM 29 (12) (1986) 1184–1201. doi:10.1145/7902.7904.
URL `http://doi.acm.org/10.1145/7902.7904`

[9] J. Campos, A. Riboira, A. Perez, R. Abreu, Gzoltar: an eclipse plug-in for testing and debugging, in: M. Goedicke, T. Menzies, M. Saeki (Eds.), IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012, ACM, 2012, pp. 378–381. doi:10.1145/2351676.2351752.
URL `http://doi.acm.org/10.1145/2351676.2351752`

[10] T. Ball, J. R. Larus, Optimally profiling and tracing programs, ACM Trans. Program. Lang. Syst. 16 (4) (1994) 1319–1360. doi:10.1145/183432.183527.
URL `http://doi.acm.org/10.1145/183432.183527`

[11] C. Runciman, N. Röjemo, Heap Profiling for Space Efficiency, in: J. Launchbury, E. Meijer, T. Sheard (Eds.), Second International School on Advanced Functional Programming, Vol. 1129 of LNCS, 1996, pp. 159–183.

[12] K. Claessen, J. Hughes, Quickcheck: a lightweight tool for random testing of haskell programs, in: Proc. of the Fifth ACM SIGPLAN International Conference on Functional Programming, ACM, New York, NY, USA, 2000, pp. 268–279. doi:http://doi.acm.org/10.1145/351240.351266.

[13] H. Wu, J. Gray, Automated generation of testing tools for domain-specific languages, in: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05, ACM, New York, NY, USA, 2005, pp. 436–439. doi:10.1145/1101908.1101993.
URL `http://doi.acm.org/10.1145/1101908.1101993`

[14] P. Godefroid, N. Klarlund, K. Sen, Dart: Directed automated random testing, in: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, ACM, New York, NY, USA, 2005, pp. 213–223. doi:10.1145/1065010.1065036.
URL `http://doi.acm.org/10.1145/1065010.1065036`

[15] T. Mens, T. Tourwé, A survey of software refactoring, IEEE Trans. Softw. Eng. 30 (2) (2004) 126–139. doi:10.1109/TSE.2004.1265817.
URL `http://dx.doi.org/10.1109/TSE.2004.1265817`

[16] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, B. Wiedermann, The dacapo benchmarks: Java benchmarking development and analysis, SIGPLAN Not. 41 (10) (2006) 169–190. doi:10.1145/1167515.1167488.
URL `http://doi.acm.org/10.1145/1167515.1167488`

[17] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, L. Yang, Accurate online power estimation and automatic battery behavior based power model generation for smartphones, in: T. Givargis, A. Donlin (Eds.), Proceedings of the 8th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2010, part of ESWeek '10 Sixth Embedded Systems Week, Scottsdale, AZ, USA, October 24-28, 2010, ACM, 2010, pp. 105–114. doi:10.1145/1878961.1878982.
URL `http://doi.acm.org/10.1145/1878961.1878982`

[18] M. Dong, L. Zhong, Self-constructive high-rate system energy modeling for battery-powered mobile systems, in: A. K. Agrawala, M. D. Corner, D. Wetherall (Eds.), Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, June 28 - July 01, 2011, ACM, 2011, pp. 335–348. doi:10.1145/1999995.2000027.
URL `http://doi.acm.org/10.1145/1999995.2000027`

[19] M. B. Kjærgaard, H. Blunck, Unsupervised power profiling for mobile devices, in: A. Puiatti, T. Gu (Eds.), Mobile and Ubiquitous Systems: Computing, Networking, and Services - 8th International ICST Conference, MobiQuitous 2011, Copenhagen, Denmark, December 6-9, 2011, Revised Selected Papers, Vol. 104 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer, 2011, pp. 138–149. doi:10.1007/978-3-642-30973-1.
URL `http://dx.doi.org/10.1007/978-3-642-30973-1_12`

[20] C. Yoon, D. Kim, W. Jung, C. Kang, H. Cha, Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. (2012) 387–400.

[21] W. Jung, C. Kang, C. Yoon, D. Kim, H. Cha, Devscope: a nonintrusive and online power analysis tool for smartphone hardware components, in: Jerraya et al. [33], pp. 353–362. doi:10.1145/2380445.2380502.
URL `http://doi.acm.org/10.1145/2380445.2380502`

[22] T. L. Alves, C. Ypma, J. Visser, Deriving metric thresholds from benchmark data, in: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 1–10. doi:10.1109/ICSM.2010.5609747.
URL `http://dx.doi.org/10.1109/ICSM.2010.5609747`

[23] F. Hermans, M. Pinzger, A. v. Deursen, Detecting and visualizing inter-worksheet smells in spreadsheets, in: Proceedings of the 34th International Conference on Software Engineering, ICSE '12, IEEE Press, Piscataway, NJ, USA, 2012, pp. 441–451.
URL `http://dl.acm.org/citation.cfm?id=2337223.2337275`

[24] W. Jung, K. Kim, H. Cha, Userscope: A fine-grained framework for collecting energy-related smartphone user contexts, in: IEEE 19th International Conference on Parallel and Distributed Systems, ICPADS 2013, Seoul, Korea, December 15-18, 2013, IEEE, 2013, pp. 158–165. doi:10.1109/ICPADS.2013.33.
URL `http://dx.doi.org/10.1109/ICPADS.2013.33`

[25] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. A. Dinda, L. Yang, ADEL: an automatic detector of energy leaks for smartphone applications, in: Jerraya et al. [33], pp. 363–372. doi:10.1145/2380445.2380503.
URL `http://doi.acm.org/10.1145/2380445.2380503`

[26] D. Kim, W. Jung, H. Cha, Runtime power estimation of mobile AMOLED displays, in: E. Macii (Ed.), Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013, EDA Consortium San Jose, CA, USA / ACM DL, 2013, pp. 61–64.
URL `http://dl.acm.org/citation.cfm?id=2485305`

[27] A. Carroll, G. Heiser, An analysis of power consumption in a smartphone, in: P. Barham, T. Roscoe (Eds.), 2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010, USENIX Association, 2010.
URL `https://www.usenix.org/conference/usenix-atc-10/analysis-power-consumption-smartphone`

[28] F. Ding, F. Xia, W. Zhang, X. Zhao, C. Ma, Monitoring energy consumption of smartphones, CoRR abs/1201.0218.
URL `http://arxiv.org/abs/1201.0218`

[29] L. Corral, A. B. Georgiev, A. Sillitti, G. Succi, Method reallocation to reduce energy consumption: an implementation in android OS, in: Cho et al. [34], pp. 1213–1218. doi:10.1145/2554850.2555064.
URL `http://doi.acm.org/10.1145/2554850.2555064`

[30] A. Noureddine, R. Rouvoy, L. Seinturier, Unit testing of energy consumption of software libraries, in: Cho et al. [34], pp. 1200–1205. doi:10.1145/2554850.2554932.
URL `http://doi.acm.org/10.1145/2554850.2554932`

[31] S. Hao, D. Li, W. G. J. Halfond, R. Govindan, Estimating android applications' CPU energy usage via bytecode profiling, in: R. Kazman, P. Lago, N. Meyer, M. Morisio, H. A. Müller, F. Paulisch, G. Scanniello, O. Zimmermann (Eds.), First International Workshop on Green and Sustainable Software, GREENS 2012, Zurich, Switzerland, June 3, 2012, IEEE, 2012, pp. 1–7. doi:10.1109/GREENS.2012.6224263.
URL `http://dx.doi.org/10.1109/GREENS.2012.6224263`

[32] M. Couto, T. Caro, J. Cunha, J. Fernandes, J. Saraiva, Detecting anomalous energy consumption in android applications, in: F. Quinto Pereira (Ed.), Programming Languages, Vol. 8771 of Lecture Notes in Computer Science, Springer International Publishing, 2014, pp. 77–91. doi:10.1007/978-3-319-11863-5_6.
URL `http://dx.doi.org/10.1007/978-3-319-11863-5\_6`

[33] A. Jerraya, L. P. Carloni, N. Chang, F. Fummi (Eds.), Proceedings of the 10th International Conference on Hardware/-Software Codesign and System Synthesis, CODES+ISSS 2012, part of ESWeek '12 Eighth Embedded Systems Week, Tampere, Finland, October 7-12, 2012, ACM, 2012.
URL `http://dl.acm.org/citation.cfm?id=2380445`

[34] Y. Cho, S. Y. Shin, S. Kim, C. Hung, J. Hong (Eds.), Symposium on Applied Computing, SAC 2014, Gyeongju, Republic of Korea - March 24 - 28, 2014, ACM, 2014.
URL `http://dl.acm.org/citation.cfm?id=2554850`