GreenDroid: A Tool for Analysing Power Consumption in the Android Ecosystem

Marco Couto*†, Jácome Cunha*‡, João Paulo Fernandes§, Rui Pereira*†, and João Saraiva*†
* HASLab/INESC TEC
† Universidade do Minho, Portugal
‡ Universidade Nova de Lisboa, Portugal
§ RELEASE, Universidade da Beira Interior, Portugal
{marcocouto,ruipereira,jas}@di.uminho.pt, jacome@fct.unl.pt, jpf@di.ubi.pt

***Abstract*—**

This paper presents *GreenDroid*, a tool for monitoring and analyzing power consumption for the Android ecosystem. This tool instruments the source code of a giving Android application and is able to estimate the power consumed when running it. Moreover, it uses advanced classification algorithms to detect abnormal power consumption and to relate them to fragments in the source code. A set of graphical results are produced that help software developers to identify abnormal power consumption in their source code.

I. Introduction

While in the previous century computer manufacturers, software engineers and software developers were mainly looking for fast computer devices/software, this has drastically changed with the recent advent and wide use of mobile devices, like laptops, tablets and smartphones. In our mobile-device age execution time is not the only concern. In fact, nowadays one of the main computing bottlenecks is power consumption. Indeed, mobile-device manufacturers and their users are as concerned with the performance of their device as they are with battery consumption/lifetime.

This growing awareness on energy efficiency is also changing the way programmers develop their software. As shown by recent empirical studies [1], software developers are more and more concerned on developing energy efficient software. Unfortunately, developing energy-aware software is still a difficult task. While the programming language community has developed advanced and widely-used software tools, such us debuggers and fault localization tools [2], memory profiler tools [3], [4], testing tools [5], [6], [7], benchmark and runtime monitoring frameworks [8], compiler optimizations [9], etc there are no equivalent tools/frameworks to profile/optimize power consumption.

This paper presents a software tool, named *GreenDroid*[1], for monitoring and analyzing power consumption for the Android ecosystem, one of the largest software ecosystems for mobile devices. This tool uses a power consumption model to estimate the power consumed by an Android application. Moreover, *GreenDroid* also relates the power consumed by the application to fragments of the applications' source code. Thus, it can give software developers an indication by pointing to the source code, where their applications may be causing an abnormal power consumption.

To do this, *GreenDroid* combines several software engineering techniques: it uses a traditional compiler front-end that parses Android/Java programs and builds an Abstract Syntax Tree (AST).

Then, it uses generalized tree traversal in the AST in order to instrument a given application's source code and test cases with calls to the power model.

After that, the Android testing framework is used to execute the instrumented test cases with the instrumented version of the application, so that it collects information about power consumed at runtime. Then, our tool uses source code classification algorithms, proposed in [10], in order to relate power consumption to source code fragments.

Finally, the tool produces different graphical results relating execution time and power consumption (bar diagrams), source code methods/classes/packages and power consumption (sunburst diagrams), hardware devices and power consumption (pie chart).

It is important to say that this is only a tool demo. Here, we demonstrate how to combine several technologies to build a tool that is capable of analyzing power consumption of Android applications. The classification methodology, the results obtained from analyzing concrete applications and the discussion about them are presented in [10].

In the next sections we will explain how the tool was developed, what technologies were used and how does the tool works. We start by briefly presenting the Power Tutor Model: a power consumption model for Android devices, and the extension we performed so that it can be used as an API by other applications. In Section III we describe *GreenDroid*'s instrumentation front-end. Section IV presents the techniques used to execute the instrumented application and to analyze the collected power-related data. It also presents the different graphical results produced by our tool when analyzing a

This work is financed by the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013.

[1]The tool is available at https://github.com/greensoftwarelab/GreenDroid.

978-1-4673-9868-8/15/$31.00 ©2015 IEEE

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on October 29,2020 at 15:50:19 UTC from IEEE Xplore. Restrictions apply.

real Android application. After that, we present related work (Section V) and our conclusions (Section VI).

II. Energy Consumption Models for Android Applications

In a computer device, the hardware is what consumes power. However, the software that operates the hardware can have a significant impact on the power consumed, very much like the driver that operates a car. Thus, in order to measure the power consumed by a software system, we have to measure the power consumed by the hardware executing it. There are two main approaches to monitor power consumption: Firstly, by using an external data acquisition (DAC) device that monitors the power consumed by other electronic device. Secondly, by using power consumption models that estimate power consumption. The *GreenDroid* tool uses this second approach.

There are several power consumption models for the Android ecosystem [11], [12], [13], [14], [15] that consider the hardware components of Android devices (like for example, CPU, Display, GPS, WiFi, etc), their characteristics (number of cores), and possible states to provide a power model. *GreenDroid* uses the power tutor model [15]: a state-of-the-art power consumption model for smartphones [11].

This power consumption model associates to each hardware component a list of different state variables. These variables influence the operating mode that a particular component can have, and, thus, the power consumed by it. Table I shows an example of a power consumption model instance for a particular smartphone.

Component	Variable	Range	Power Coefficient
CPU	util	1-100	β_{uh}: 4.34
			β_{ul}: 3.42
	$freq_l$, $freq_h$	0,1	n.a.
	CPU_on	$0-1$	β_{cpu}: 121.46
Wi-Fi	npackets, R_{data}	$0-\infty$	n.a.
	$R_{channel}$	1-54	β_{cr}
	$Wi-Fi_l$	0,1	β_{Wi-Fi_l} : 20
	$Wi-Fi_h$	0,1	β_{Wi-Fi_h}: 720
Audio	Audio_on	0,1	β_{audio}: 384.62
LCD	brightness	0-255	β_{br}: 2.40
GPS	GPS_on	0,1	β_{Gon}: 429.55
	GPS_sl	0,1	β_{Gsl}: 173.55
3G	data_rate	$0-\infty$	n.a.
	downlink_queue	$0-\infty$	n.a.
	uplink_queue	$0-\infty$	n.a.
	$3G_{idle}$	0,1	β_{3G_idle}: 10
	$3G_{FACH}$	0,1	β_{3G_FACH}: 401
	$3G_{DCH}$	0,1	β_{3G_DCH}: 570

TABLE I: Power tutor instance for the HTC Dream smartphone, reprinted from [15]

Using this model, one can calculate the actual power consumption of the device at a given time, applying the following formula:

$$\begin{aligned} &Power = (\beta_{uh} \times \text{freq}_h + \beta_{ul} \times \text{freq}_l) \times \text{util}+ \\ &\beta_{CPU} \times \text{CPU}_{on} + \beta_{br} \times \text{brightness} + \beta_{Gon} \times \text{GPS}_o n+ \\ &\beta_{Gsl} \times \text{GPS}_s l + \beta_{\text{Wi-Fi}_l} \times \text{Wi-Fi}_l + \beta \text{Wi-Fi}_h \times \text{Wi-Fi}_h+ \\ &\beta_{3G_idle} \times 3G_{idle} + \beta_{3\text{G}_{FACH}} \times 3\text{G}_{FACH}+ \\ &\beta_{3G_DCH} \times 3G_{DCH} \end{aligned} \quad (1)$$

A. The Power Comsumption Model as an API

The Power Tutor Model is available as an Android standalone tool, which shows the current power consumed by an Android device. To be able to reuse this power model to monitor the power consumed by a known application, we need to update its implementation into an API-based software, so that its methods can be reused/called in the instrumented source code.

Thus, we introduce 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 the Power Tutor Model.
- start(): Starts the power monitoring thread.
- stop(): Stops the power monitoring thread and saves the results.

III. Energy Consumption Instrumentation

Having introduced an API for the power consumption model, we describe now the instrumentation phase of the *GreenDroid* tool: giving the source code of an Android application and its test cases, it embeds calls to the model's API in both the source code and test cases. This phase is shown in Figure 1. The idea is that when the (instrumented) test cases are executed by the (instrumented) application, then power consumption is monitored during that execution.

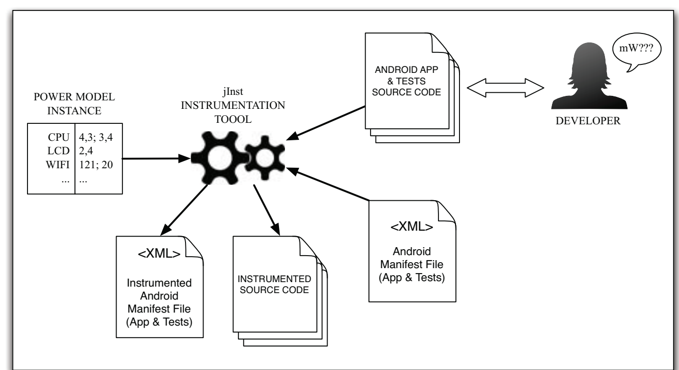

Fig. 1: *GreenDroid*: Instrumentation of the code and tests of an Android Application

A. Java front-end for source code transformation

Android applications are written in Java. Thus, in order to automatically instrument the source code of an Android application, we use the well-known Java front-end framework[2]: it consists of a parser (produced by JavaCC[3] from a Java

[2] https://code.google.com/p/javaparser.

[3] Java Compiler Compiler web page: https://javacc.java.net.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on October 29,2020 at 15:50:19 UTC from IEEE Xplore. Restrictions apply.

grammar), the construction of the corresponding AST, and generalized methods to traverse that tree. Next Java fragment shows the *GreenDroid* invocation to the generated Java parser.

```
//creates an input stream for the file to be parsed
  FileInputStream in = new FileInputStream("test.java");
  CompilationUnit cu;
  try { //parse the file
    cu = JavaParser.parse(in);
      } finally { in.close(); }
  ..
```

As we can see, the parse method receives as argument an input stream for a file containing java code, parses it and returns an object from the class CompilationUnit. This object is the root of the AST, and it consists of an entire class hierarchy representing the tree.

Once created this class hierarchy, we have an object oriented representation of the AST. Moreover, the Java front-end also supports a simple form of generalized tree traversals [16], making it easier to manipulate such AST as necessary in our instrumentation phase. This is described next.

B. Source code and Test Instrumentation

The AST returned by the Java parser has to be updated with calls to our API. Our goal is to relate power consumption with the applications' source code, therefore we have to instrument fragments of the source code. Since programmers structure their programs in methods, *GreenDroid* instruments source code methods.

However, because methods may consume a tiny negligent amount of power, that is not measurable by the power model, we decide to monitor power consumption in the test cases only. Source code methods are instrumented in order to trace their execution. Then, our classification algorithms [10] relate power consumption to method calls.

In order to achieve this transformation, we needed to implement a method that traverses through the AST and injects tracing instructions on the application methods (i.e., the traceMethod() method of our API). The following code fragment shows how to use the provided generalized tree traversal visitor to go through all the methods of an AST representing a Java source code file:

```
...
CompilationUnit cu;
...
//parse one java file, like in the previous listing
...
new MethodChangerVisitor().visit(cu, className);
```

The last instruction performs the traversal over some type of nodes of the AST. The name of the class implies that is a method declaration visitor. In order to enable method declaration visiting, we needed to overwrite the visit method. The next code fragment shows an implementation of that method that injects calls to our API traceMethod:

```
public class MethodChangeVisitor extends VoidVisitorAdapter {
...
@Override
public void visit(MethodDeclaration n, Object arg)
  ...
  String cName = (String)arg;
  Expression className = new StringLiteralExpr(cName);
  Expression method = new StringLiteralExpr(n.getName());
  ...
  MethodCallExpr mcB = new MethodCallExpr();
  mcB.setName("Estimator.traceMethod");
  ASTHelper.addArgument(mcB, className);
  ASTHelper.addArgument(mcB, method);
  ...
  n.getBody().getStmts().add(0, mcB);
  ...
}
...
}
```

This code injection allows to instrument the application so that it keeps trace of the called methods. Next, we explain how the source code of the tests is instrumented in order to monitor the power consumed.

1) Test Cases Instrumentation: In order to use the instrumented application and the developed *Estimator* power class, the application needs to call the start and stop methods before/after every test case is executed. This will enable a power consumption sampling during the execution of a test. Both jUnit and Android testing framework allow test developers to write a *setUp* and a *tearDown* method. These two methods are executed after a test case (test method) starts and after it ends, respectively. All test cases belonging to the same test suite (test class) will first call setUp, then execute and at the end call tearDown

Thus *GreenDroid* needs to instrument those methods (or create them if they do not exist) with calls to the API methods, like the following example:

```
public class TestA extends
  ActivityInstrumentationTestCase2<ActivityA>{
...
public void setUp(){
  Estimator.config(
    "package", android.os.Process.myUid(), this.getContext());
  Estimator.start();
  ... }
...
public void tearDown(){Estimator.stop(); ... }
```

This approach ensures that every time a test begins, the *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 power to be consumed. The *config* method is necessary, since the power monitor needs to know the UID and the context of the application being tested, for each test. The *tearDown* method is responsible for stopping the thread and saving the results.

To perform this instrumentation, we follow the same strategy as in the source code instrumentation. First, we define a new type of visitor and implement a visit method, and then we call it to perform the AST traversal. In this specific case, we need the visit method to first check if there is a setUp and tearDown method. If that is true, then it injects the instructions, as shown in the next code fragment:

```
public class TestChangerVisitor extends VoidVisitorAdapter{
@Override
public void visit(MethodDeclaration n, Object arg) {
  Definitions defs = (Definitios arg);
  Expression className =
    new StringLiteralExpr(cDef.getDescriptor());
  ...
  if(n.getName().equals("setUp")){
          defs.setSetUp(true);
          MethodCallExpr mStt = new MethodCallExpr();
    mStt.setName("StaticEstimator.start");
    ...
    MethodCallExpr mCon = new MethodCallExpr();
```


Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on October 29,2020 at 15:50:19 UTC from IEEE Xplore. Restrictions apply.

```
    mCon.setName("StaticEstimator.config");
    ...
    n.getBody().getStmts().add(0, new ExpressionStmt(mCon));
    n.getBody().getStmts().add(1, new ExpressionStmt(mStt));
    ...
  }else if(n.getName.equals("tearDown")){ ... }
  ... }
... }
```

If at the end of the visiting the AST we have not found a setUp and/or a tearDown method, we need to create new methods and inject them directly in the class (test suite). The next code fragment shows how this operation is performed:

```
CompilationUnit cu = JavaParser.parse(in);
new TestChangerVisitor().visit(cu, defs);
if(!defs.hasSetUp()){ //create the setUp method
        MethodDeclaration newSetUp = new MethodDeclaration();
        newSetUp.setName("setUp");
    newSetUp.setModifiers(ModifierSet.PUBLIC);
    newSetUp.setType(new VoidType());
    newSetUp.setBody(new BlockStmt());
    ...
    //add the setUp method
    ...
    MethodCallExpr mcStart = new MethodCallExpr();
    mcStart.setName("StaticEstimator.start");
    ...
    MethodCallExpr mcConfig = new MethodCallExpr();
    mcConfig.setName("StaticEstimator.config");
    ...
    ArrayList<Statement> body = new ArrayList<Statement>();
    body.add(new ExpressionStmt(mcConfig));
    body.add(new ExpressionStmt(mcStart));
    ...
    newSetUp.getBody().setStmts(body);
    cu.getTypes().get(0).getMembers().add(newSetUp); }
if(!defs.hasTearDown()){ ... }
```

IV. Energy Consumption: Monitoring and Analysis

After both the application source code and test cases are instrumented, *GreenDroid* uses the Android Testing framework to execute the tests so that it monitors and analyzes the Power consumption of the underlying application. Figure 2 shows the architecture of this phase of our tool.

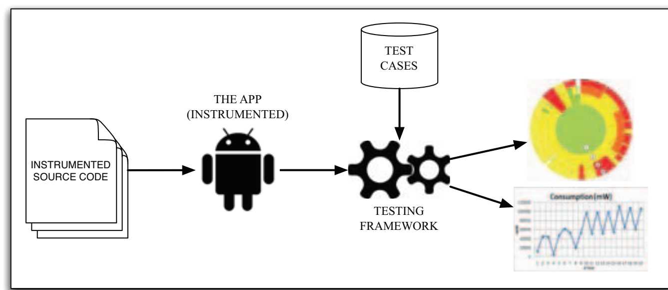

Fig. 2: *GreenDroid*: Power monitoring and analysis

A. Framework for Test Execution

In order to execute the instrumented tests of the Android application, *GreenDroid* uses the Android testing framework[4] which is based on jUnit.

Before running the application on a specific Android device, first *GreenDroid* creates the files for both projects, using the android test tools, and installs them. This is done by the following command lines:

```
#Generate the installation files for the project and test project
$ android update project -p "path/to/project" -n Green
$ android update test-project -p "path/to/test/project"
--main "path/to/project"
$ ant -f "path/to/test/project/build.xml" clean
$ ant -f "path/to/test/project/build.xml" debug

#Install previous generated files in the device
$ adb install -r "path/to/project/bin/Green-debug.apk"
$ adb install -r "path/to/test/project/bin/GreenTest-debug.apk"
```

The tests can now be executed using the Android testing framework. The following example shows how to run previously installed tests in the device by invoking an android tool:

```
$adb shell am instrument
"test.package/com.zutubi.android.junitreport.JUnitReportTestRunner;
```

Since the instrumented tests run on the instrumented application, both the program trace and power consumption metrics are generated. With this information we produce different graphical views on those results so that software developers can easily identify abnormal consumption in their source code.

B. GreenDroid: Power Consumption Graphics

To help software developers in analyzing the power consumption of their applications, *GreenDroid* produces three different graphical results relating execution time and power consumption, hardware devices and power consumption, and source code methods/classes/packages and power consumption.

a) Execution Time and Power Consumption:: GreenDroid generates a bar diagram where per executed test it displays the total execution time of the test, and the power consumed per second. Figure 3 shows an example produced by executing the twenty test cases available for an open source Android application (*0xBenchmark* application).

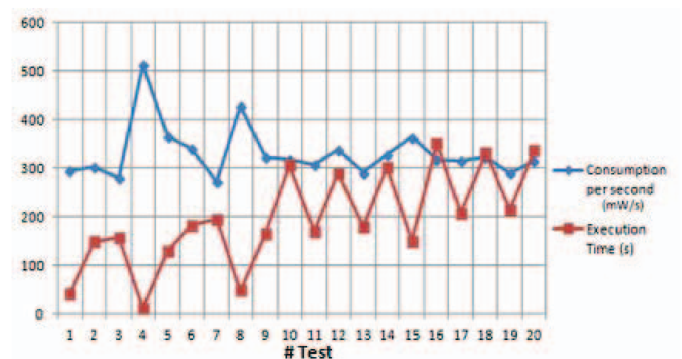

Fig. 3: Consumption per second versus Execution time per test case.

This diagram shows which test cases have higher power consumption per second, and thus, indicating an abnormal power consumption. For example, in Figure 3 we can see that two of the fastest test cases are the ones consuming more power per second. This goes against the usual intuition that faster is greener!

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

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on October 29,2020 at 15:50:19 UTC from IEEE Xplore. Restrictions apply.

b) Hardware Devices and Power Consumption:: Some hardware devices, like for example GPS and Bluetooth, are usually very power consuming. In order to analyze power consumption of an Android application it is very useful to identify which hardware device is responsible for it. Thus, *GreenDroid* generates a pie chart (as shown in Figure 4), that represents the total power consumed per hardware component.

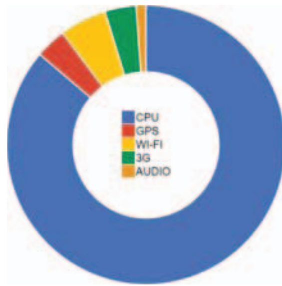

Fig. 4: Power Consumption per Hardware Device.

c) Source Code and Power Consumption:: In [10] we have presented a methodology to classify program methods according to power consumption. We defined a simpler version of the spectrum-based fault localization [17] technique to classify methods as green/yellow/red classification, where green is a power efficient method and red a power inefficient one. A method classified as yellow means that it has been used in both power efficient and inefficient program executions. *GreenDroid* produces a sunburst diagram where the methods of the monitored application are shown according to this classification. Figure 5 shows the sunburst diagram resulting of analyzing the *0xBenchmark* application.

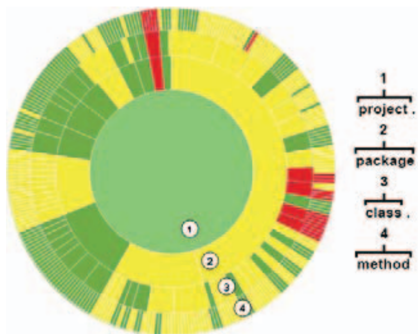

Fig. 5: Sunburst diagram (and how to interpret it)

The outermost circle represents the methods, and they are separated by classes, which are separated by packages. The classification of classes and packages depends on the classification made to the methods included in them. This diagram also gives information about the hardware device responsible for the power consumption (by pointing to a method/class), and to its source code. Thus, a software developer can immediately look at the source code of red methods in order to optimize them in terms of power.

V. 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 power for a *smartphone* application. The majority of these tools, however, focus on Android based *smartphones*, mostly because it is an open source OS[5] and statistics reveal that the sale of Android devices are much higher than any other *smartphone*[6]. In fact, in the second quarter of 2013 almost 80% of the market share belonged to Android devices.

As previously mentioned and described, Power Tutor [15] was our starting point, as it was for many other research works. For example, DevScope [14] 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 [13] to estimate the power consumption of an application, and by UserScope [18] 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) [19] 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 [12], [20], [21], are not as powerful as the previously mentioned ones. SEMO [22] is a power monitoring system and application for Android smartphones which profiles application power usage based only on the battery discharge level, and unfortunately produces less reliable results due to this. JouleTrack [23], a web based tool for software energy profiling, which allowed developers to upload their code to be executed in a specific machine, where the average energy consumption by processor instruction was already known. This tool was discontinued in 2006.

More recent works are also focused on defining reference models with metrics and criteria for green computing and energy sustainable software engineering. The GREENSOFT model [24] proposes a definition of "Green and Sustainable Software" and "Green and Sustainable Software Engineering", and it also defines a model that helps software developers and users in creating, maintaining, and using software in a more sustainable way, providing metrics and criteria for measuring software quality and classifying a software product's sustainability. In a similar work, it was developed a new language, named *Eco* [25], that is a minimal extension to Java, and enables the possibility to develop a program that adaptively adjusts its own behaviors to avoid leading to battery drain and/or CPU overheating. This new language is based on a novel energy-aware and temperature-aware programming model that, like the previous model, is focused on improving software sustainability.

The closest work to our paper is Hao' and colleagues' work with *eCalc* [26]. They also estimate Android application's energy through the execution of software artifacts with a series of test cases, alongside previously created power consumption

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

[6]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.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on October 29,2020 at 15:50:19 UTC from IEEE Xplore. Restrictions apply.

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 [10] where we presented a simple method classification algorithm: a method is considered as having an abnormal power consumption whenever it is called in a program execution which consumes more power than the average of all monitored runs of that program. As a consequence, that approach compares power of the different executions of the same application.

VI. Conclusions

This paper is a tool demo of *GreenDroid*: a tool to analyze the power consumption of Android applications and detect possible power leaks in the source code. The tool focuses on providing to the developers several representations of the analysis made to the energy efficiency of Android applications. In fact, we have already tested this tool with 6 different Android open-source applications. The results are presented and discussed in our previous works [10], and they indicate that it is possible to locate which parts of the code may be leading to energy inefficiency. We are focused on testing our tool with other applications to check if it is still able to present the same kind of results, and we still want to validate our classification method to test its consistency and accuracy. Moreover, we also want to extend this tool to be able to analyze other types of applications, and not only Android-based ones.

References

[1] G. Pinto, F. Castor, and Y. D. Liu, "Mining questions about software energy consumption," in *Proc. of the 11th Working Conf. on Mining Software Repositories*. NY, USA: ACM, 2014, pp. 22–31.

[2] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'12*, 2012, pp. 378–381.

[3] T. Ball and J. R. Larus, "Optimally profiling and tracing programs," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 4, pp. 1319–1360, Jul. 1994.

[4] C. Runciman and N. Röjemo, "Heap Profiling for Space Efficiency," in *2nd Int. School on Advanced Functional Programming*, vol. 1129. sv, 1996, pp. 159–183.

[5] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," in *Proc. of the 5th ACM SIGPLAN Int. Conf. on Functional Programming*, NY, USA, 2000, pp. 268–279.

[6] H. Wu and J. Gray, "Automated generation of testing tools for domain-specific languages," in *Proc. of the 20th IEEE/ACM Int. Conf. on Automated Software Engineering*, NY, USA, 2005, pp. 436–439.

[7] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proc. of the 2005 ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, ser. PLDI '05, NY, USA, 2005, pp. 213–223.

[8] 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, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," *SIGPLAN Not.*, vol. 41, no. 10, pp. 169–190, Oct. 2006.

[9] D. A. Padua and M. J. Wolfe, "Advanced compiler optimizations for supercomputers," *Commun. ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986.

[10] M. Couto, T. Car cão, J. Cunha, J. P. Fernandes, and J. Saraiva, "Detecting anomalous energy consumption in android applications," in *Programming Languages*, ser. LNCS, F. Quintão Pereira, Ed., vol. 8771. Springer, 2014, pp. 77–91.

[11] M. Dong and L. Zhong, "Self-constructive high-rate system energy modeling for battery-powered mobile systems," in *Proc. of the 9th Int. Conf. on Mobile Systems, Applications, and Services (MobiSys 2011), Bethesda, MD, USA, 2011*, 2011.

[12] M. Kjærgaard and H. Blunck, "Unsupervised power profiling for mobile devices," in *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, ser. LNCS SITE, A. Puiatti and T. Gu, Eds., vol. 104. Springer, 2012, pp. 138–149.

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

[14] W. Jung, C. Kang, C. Yoon, D. Kim, and H. Cha, "Devscope: A nonintrusive and online power analysis tool for smartphone hardware components," in *Proc. of the 8th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. NY, USA: ACM, 2012, pp. 353–362.

[15] L. Zhang, B. Tiwana, R. Dick, Z. Qian, Z. Mao, Z. Wang, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *Proc. of Int. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Oct 2010, pp. 105–114.

[16] J. Visser and J. Saraiva, "Tutorial on strategic programming across programming paradigms," in *8th Brazilian Symposium on Programming Languages (SBLP)*, 2004.

[17] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-based multiple fault localization," in *Proc. of the 2009 IEEE/ACM Int. Conf. on Automated Software Engineering*, ser. ASE '09. Washington, USA: IEEE Computer Society, 2009, pp. 88–99.

[18] W. Jung, K. Kim, and H. Cha, "Userscope: A fine-grained framework for collecting energy-related smartphone user contexts," in *Proc. of the 2013 Int. Conf. on Parallel and Distributed Systems*. Washington, DC, USA: IEEE Computer Society, 2013, pp. 158–165.

[19] L. Zhang, M. S. Gordon, R. P. Dick, Z. M. Mao, P. Dinda, and L. Yang, "Adel: An automatic detector of energy leaks for smartphone applications," in *Proc. of the 8th IEEE/ACM/IFIP Int. Conf. on Hardware/Software Codesign and System Synthesis*. New York, NY, USA: ACM, 2012, pp. 363–372.

[20] D. Kim, W. Jung, and H. Cha, "Runtime power estimation of mobile amoled displays," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, March 2013, pp. 61–64.

[21] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *2010 USENIX Annual Technical Conference, Boston, USA, June 23-25*, 2010.

[22] F. Ding, F. Xia, W. Zhang, X. Zhao, and C. Ma, "Monitoring energy consumption of smartphones," *CoRR*, 2012.

[23] A. Sinha and A. P. Chandrakasan, "Jouletrack: A web based tool for software energy profiling," in *Proc. of the 38th Annual Design Automation Conference*. ACM, 2001.

[24] S. Naumann, M. Dick, E. Kern, and T. Johann, "The greensoft model: A reference model for green and sustainable software and its engineering," *Sustainable Computing: Informatics and Systems*, 2011.

[25] H. S. Zhu, C. Lin, and Y. D. Liu, "A programming model for sustainable software," in *Proceedings of the 37th International Conference on Software Engineering*, 2015, to appear.

[26] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating android applications' CPU energy usage via bytecode profiling," in *First International Workshop on Green and Sustainable Software, GREENS 2012, Zurich, Switzerland*, 2012.

Authorized licensed use limited to: b-on: UNIVERSIDADE DO MINHO. Downloaded on October 29,2020 at 15:50:19 UTC from IEEE Xplore. Restrictions apply.