Orka: A New Technique to Profile the Energy Usage of Android Applications

Benjamin Westfield[*] and Anandha Gopalan
Department of Computing, Imperial College London, 180 Queens Gate, London SW7 2AZ, U.K.

Keywords: Green Computing, Energy Profiling and Measurement, Energy Monitoring.

Abstract: The ever increasing complexity of mobile devices has opened new, exciting possibilities to both designers of applications and their end users. However, this technological improvement comes with an increase in power consumption, a drain that battery technology has not managed to keep up with. Due to this, application developers are now facing a new optimisation challenge not present for traditional software: minimising energy usage. Developers need guidance to help reduce energy usage while not compromising on the features of their application. Despite research identifying areas of code consuming high energy, developers currently don't possess the necessary tools to make judgements on their application's design based on this. This paper presents Orka, a new tool that analyses an Android application and provides feedback on exactly where the application is expanding energy, thus enabling developers to improve its energy-efficiency. Orka profiles an application using user-defined test cases, code injection techniques and bytecode analysis. Feedback provided is the energy usage at the method level as well as any consumption due to hardware used. Moreover, to be useful over the entire development life-cycle, this feedback is compared with feedback from previous versions of the application so as to monitor and improve the energy usage.

1 MOTIVATION

Mobile technology has changed dramatically in recent years. Devices can now display full 1080p real time videos, are capable of running games with high quality graphics, and contain enough sensors that they can be used as full virtual reality headsets with little extra hardware (Winchester, 2015).

Harnessing this potential, the application industry is forecast to be worth $54.89 billion by 2020 (Chaudhari, 2015). Research has shown that energy consumption of applications is of great concern to users of mobile devices (Heikkinen et al., 2012; Wilke et al., 2013). A glance at on-line repositories confirms this, with applications receiving lower ratings if users perceive them as consuming more of their device's battery (Jabbarvand et al., 2015). Due to this change in consumer habits, power optimisation is important for developers. Previous research has focused on time optimisation for software, as computers were earlier connected to a constant power source. Mobile application developers are now facing end-users who require software to be optimised to use as little of the battery as possible, less they drain this finite resource. This paper was inspired by the perceived lack of energy optimisation tools available to developers. Tools exist to optimise efficiency but energy optimisation can be orthogonal in nature to time optimisation (Bunse et al., 2009).

Research into Green Computing has started to identify the areas of code that consume the most energy. These findings allow testing methods to be constructed in order to provide developers with exciting new ways in which to optimise their code. However, most of these are still tied to hardware or require the understanding to interpret results from a multi-meter to provide execution costs (as current systems do, and which are discussed in Section 2). We believe that all developers should be able to have access to their code's energy performance regardless of whether they are professionals or coding on their own.

To address these issues, this paper introduces Orka, a novel approach to providing energy usage feedback to software developers. Orka provides feedback based on an application's API usage, as well as the energy usage of the app, down to the method level. It is important that software energy usage not be disassociated from hardware energy usage, so Orka also

[*]This work was done while this author was a student at Imperial College London.

Westfield, B. and Gopalan, A.
Orka: A New Technique to Profile the Energy Usage of Android Applications.
In *Proceedings of the 5th International Conference on Smart Cities and Green ICT Systems (SMARTGREENS 2016)*, pages 213-224
ISBN: 978-989-758-184-7

provides feedback on any energy consumption due to hardware usage. To the authors' knowledge, this is the first attempt to combine these two sources of energy usage. Previous research has focused on either one or the other. Orka tests the app using a dynamically created execution trace generated using a test script provided by the application's developer. Rather than running on physical devices, Orka performs the hardware analysis running on emulators. After running the application, Orka pulls the internal energy usage estimations from the emulator to provide feedback based on the different components used. Orka has been designed for applications on the Android OS, which has grown to be the most widely used OS for mobile technologies in the world (Rivera, 2015).

Despite the similarity, Orka is not a reference to large aquatic mammals. Dalvik (the original Virtual Machine used by Android OS) was named after the Icelandic village that was home to the ancestors of Dan Bornstein, the original developer of Dalvik. Smali and Baksmali (two tools that form part of the structure of Orka) were named after the Icelandic words for assembler and disassembler. In continuing with tradition, this system was also given an Icelandic name, Orka - meaning energy.

The remainder of this paper is organised as follows: Section 2 outlines work related to this paper while Section 3 gives an overview of the research challenges encountered while building Orka. Section 4 details the architecture of Orka, while Section 5 describes its prototype implementation. Section 6 provides an evaluation of Orka, while Section 7 lists some of the limitations of this work, and Section 8 concludes the paper and provides ideas for future work.

2 RELATED WORK

Orka is not the first attempt at providing feedback on energy usage for Android applications. Cycle accurate simulators (Brooks et al., 2000) can be used to accurately simulate a processor's cycle at an architectural level. While effective, these have been labelled as inefficient (Hao et al., 2012). Hao and colleagues (Hao et al., 2012) proposed a system, eCalc, to replace these. After creating an energy cost for each instruction, the eCalc system analyses the bytecode and each method was assigned a cost function. After capturing the execution trace of the program, a number of tests were then executed on this and compared to multi-meter readings.

Both eCalc and cycle accurate simulators suffer from the same issue: they only focus on the CPU. Other pieces of hardware are often more energy greedy than the CPU (Corral et al., 2013; Dong and Zhong, 2012). Research has shown that lowering the brightness of the screen can save up to 60% of an application's battery usage (Dong and Zhong, 2012). This has been supported by further research (Corral et al., 2013).

Elens, proposed by Hao and colleagues (Hao et al., 2012), extends the model of bytecode profiling further. This attempts to model the different energy usage by network transfers by including a stack trace. A call-graph is generated showing complete execution paths through each method. The stack size for transmitted data is used to model the linear growth of energy costs for network transmissions with the size of data sent. Both Elens and eCalc compare their models to actual measurements from a multi-meter. This presents an issue of how to accurately tie measured energy back to specific methods due to the fact that the Android operating system has asynchronous power states (Pathak et al., 2012) and also due to the phenomenon of 'tail energy' (Li et al., 2013), where a program or routine can draw power beyond the end of its execution. Both Elens and eCalc also require users to use multi-meter readings to estimate energy, and they were all run on specially developed hardware. Neither of these are readily available to developers, thus limiting the use of these models. In order for them to run correctly, each bytecode instruction needs to be accounted for in terms of energy cost. To do this, each instruction must be measured *for each* hardware and operating system combination. A developer would need to create this for whichever environment they want the application to run on. It is unlikely they would have the capacity to do this.

Orka is not the first to leverage the findings of Linares-Vasquez et al (Linares-Vásquez et al., 2014). Jabbarvand et al (Jabbarvand et al., 2015) used these to create their modelling device, EcoDroid. By assigning energy costs to nodes of a generated call-graph, their system could estimate an application's energy consumption by API usage. They proposed including static tests, rather than just dynamic tests to ensure full coverage. Orka does not use static tests as it provides feedback on a specific test case, rather than attempting to provide a total code energy cost.

All models previously focused only on the energy usage of software. As far as we are aware, no other research has attempted to combine this with energy costs owing to hardware usage. Computers, after all, are just machines; all their different parts consume electricity. Therefore a complete testing tool should factor in both the drain due to software and that owing to the hardware.

3 RESEARCH CHALLENGES

In order to build Orka, the initial challenge encountered was to find out a method to measure the energy usage of the application's code. The actual discharge of the battery could not be measured as this would require physical hardware. Therefore, Orka would need to estimate energy usage from identifying areas of the source code with high energy costs. Linares-Vasquez et al (Linares-Vásquez et al., 2014) found that the majority of energy in applications was spent calling the Android Application Program Interface (API). In an extensive study, they documented the energy usage of these APIs. While the exact amount of energy usage (in Joules) might differ between different hardware combinations, it is reasonable to assume that these costs would remain in the order of these findings. Using these findings, Orka assigns this cost to each API call in the code. Orka is aware of flow control, taking into consideration that each call in the code could be made multiple times, in a loop for example.

To analyse the application's code, a number of different approaches were tried. Originally, we wanted to use a non invasive method that did not modify the source code. Xposed[2] is a module that modifies the script responsible for loading Zygote (the parent process for all applications that run on Android) at startup. Any interface added to this would be accessible from all other Android applications.

Other researchers (Jabbarvand et al., 2015) have had some success using Xposed, however we could not install it on emulated devices, an experience shared by others in the Android development community (XDADevelopers, 2014; StackOverflow, 2015). This restriction on emulators also ruled out a series of third party systems that could have been used for analysis, so we decided to focus on ways to create new logging methods.

When API calls are identified in the code, the system needs to identify and tally the number of calls. However, due to the twisting and dynamic nature of the execution of code, performing static analysis on an app's code would not provide an accurate view of the number of times each API is called, so a dynamic analysis would be necessary. In order to achieve this, we use Monkey Runner[3], a user interface testing tool for Android that simulates an end user interacting with the app.

A method was needed to tally the calls to the API during the execution of the app. As non-invasive methods had been ruled out, methods needed to be inserted to tally each API call. Orka achieves this by using an improved version of a key logger written for a popular Android application (Casey, 2013). The previously manual process had new logic inserted to automatically inject new logging methods without changing the underlying application (detailed below). These added methods insert the API calls to the global Android log, Logcat[4].

Estimating hardware energy usage presents its own pitfalls. Applications can draw power beyond the end of their execution due to a phenomenon called 'tail energy'(Li et al., 2013). The operating system will keep hardware components turned on for a period of time after an instruction to close. This allows it to amortise startup costs and occurs even if the system is idle. Another issue that can occur is with applications holding a 'wakelock' that does not allow the processor to enter a lower power mode. Orka uses Dumpsys[5], a tool that provides details about the status of the device's hardware in order to counteract this. One feature of this tool is the ability to view the contents of *BatteryStatsInfo.bin*, the battery estimation used by Android to populate the estimated battery usage functions stored in the device's settings menu. This is populated as long as the device's battery is not being charged.

The algorithms that populate *BatteryStatsInfo.bin* take into account that the end of a routine does not mean a used component will power down instantly. Using Dumpsys would, hence, solve the issues relating to tail energy. Furthermore, if another app runs because of a wakelock held by a specified application, then half the total energy usage of the second application is assigned to the one holding the wakelock (Google, b).

4 ARCHITECTURE

The architecture of Orka is illustrated in Figure 1. The system is divided into two separate modules: the injector, which is responsible for drawing the required information about the application from its .apk file, then injecting logging methods after ever API call; and the analyser, which runs the injected application on an emulated device, then computes the total energy usage from the execution trace. The sequence of steps followed by Orka (and detailed in the following sections) are:

- Convert the .apk file into Smali files.

[2]https://github.com/rovo89/XposedBridge/

[3]http://developer.android.com/tools/help/monkey.html

[4]http://developer.android.com/tools/help/logcat.html

[5]https://source.android.com/devices/tech/debug/dumpsys.html

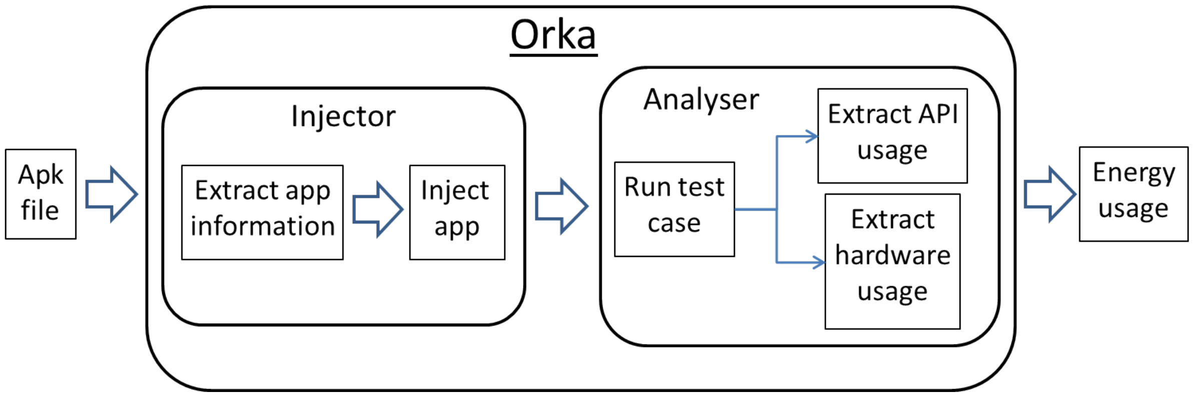


Figure 1: A system architecture overview of Orka.

- Inject logging functions into the code, and insert logging class into root directory of application.
- Recompile the application and resign it so it will run on device emulators.
- Load emulator and install the application on it.
- Run Monkey Runner script containing test execution.
- Extract Logcat files and hardware usage from Dumpsys.
- Process Logcat data by summing the number of API calls in each method and calculating the total cost of these calls.
- Present results to the user.

Orka was developed to run on Linux and the decision was made to write the application in Python. This is because the designed process was inherently procedural, and would require a number of different scripts to run (for example to pull data using the ADB).

4.1 Injector

The injector is responsible for injecting the app, and for any task required to complete this. To facilitate code injection, the app's .apk file is decompressed into an intermediate representation called Smali. Smali was developed to represent a human-readable version of the app's Dex code[6]. Orka uses Apktool, a tool to further aid automating the transformation process into Smali[7]. Apktool was chosen as extra media stored in the .apk hierarchy (required for recompilation) would not be lost during decompression. Once decompiled, the output directory contains all relevant files for the app, with the Smali representation of the code located in its own sub-directory. Rather than prompt the user for the application name to construct the path, the injector extracts this from the application's Android Manifest using command line tools developed by Google[8].

Any files that have been added to the directory will be included in the application once re-compressed to a .apk file. Leveraging this, Orka adds a Smali file with the others thus making it part of the application's package. This means that it can be referenced absolutely in the app's source code. This is important as foreign packages need to be loaded into a register in order to be referenced. By adding the logging methods to the app's package, it frees Orka from needing to load this into a register to reference it (reducing the number of changes to the underlying code). This helps keep the the injected application as similar to the original as possible, giving a truer picture of the energy costs of running the original app.

4.1.1 Logging Methods

Orka's logging interface contains two methods: one to be called after entering a new method that extracts the method's name from the stack trace, and a second that is passed an API name as a string constant to log a call. The former does not require any interference with the application's code as it uses the stack trace. The latter requires a free register to be inserted to the method as variables are passed by register reference (Google, a). The messages sent to Logcat have a unique tag so that the analyser can easily extract these later. These two methods are combined to create an execution trace by logging when the application enters a new method and then each call it makes to an API. Storing the method name makes it possible to attribute the API calls to the correct method.

[6]https://github.com/JesusFreke/smali

[7]https://www.georgiecasey.com/2013/03/06/inserting-keylogger-code-in-android-swiftkey-using-apktool/

[8]https://developer.android.com/tools/building/index.html

4.1.2 Code Injection

Code injection is done by manipulating the Smali code and expanding it by automating the process and adding further logic to automate the analysis of the code. While each Smali file is human-readable, it has a uniform representation due to previously being bytecode. The injector leverages this by opening each file in the Smali directory into a file stream, then analysing each line. Its built-in logic will then react to finding specific bytecode patterns. Each original line of Smali is output (along with any injected code) into new Smali files. Once inspected, the original file is deleted, leaving only the injected files to be recompiled into the injected application. It should be noted that this is not a destructive process. The original .apk file is not edited in any way.

Each file of Smali represents a different Java class file in the original source code, therefore containing a number of methods. Each method is checked individually and independently of the others. The injector pattern matches each line of the file checking for a function signature - signifying the start of a new method. On finding this, the injector checks to see if the method's signature matches that of a constructor (**method public constructor <init>**), ignoring the entire method on matching. After entering a new method, Orka looks at the following lines until one of two conditions are satisfied:

- An API call is found
- The end of method statement is found.

If there are no API calls found in the method, the injector will not inject any code into that method, instead reconstructing this unaltered in the output file. This limits the interference with the source code to when it is truly needed. If an API call is found, the injector calls a seek function to revert to the start of the method. Until the injector identifies the end of this method, it will now perform a number of behaviours depending on what it identifies in the source code.

4.1.3 Adding a Register

The Smali representation of Dex separates registers into two types: locals and parameters. Locals represent both the local variables of a method as well as storing any external classes used and the results of calculations. The number of registers available to a method is fixed, being declared on entering a method. This total is decided when the original Java code is converted into Dex (Smali, 2015). Of these registers, the registers numbered 0 to n-1 will contain all the local registers, where n is the number of locals declared. Parameters are always stored in the registers after the last local register. These represent the arguments passed to the method. Every method has at least one parameter representing a pointer to that method (*this). The total number of local registers is explicitly declared in the Smali code, followed by a declaration of the contents of each parameter register. As per Section 3, the injector adds a register to store the name of any API call. By increasing the number of explicitly declared locals, a new register will be added to the end of these (before the first parameter). The original locals declaration is then discarded with the new declaration output.

Using a naive approach that does not take into account the number and type of registers would cause a number of fatal bugs to be introduced to the app once recompiled. Android was built with the assumption that most methods will not have more than 16 registers (Google, a). As such bytecodes are separated into groups depending on whether they can reference a register whose number can be represented using 4 bits (the first 16) or whether it's number can only be represented by using 8 bits (the first 256) (Google, a).

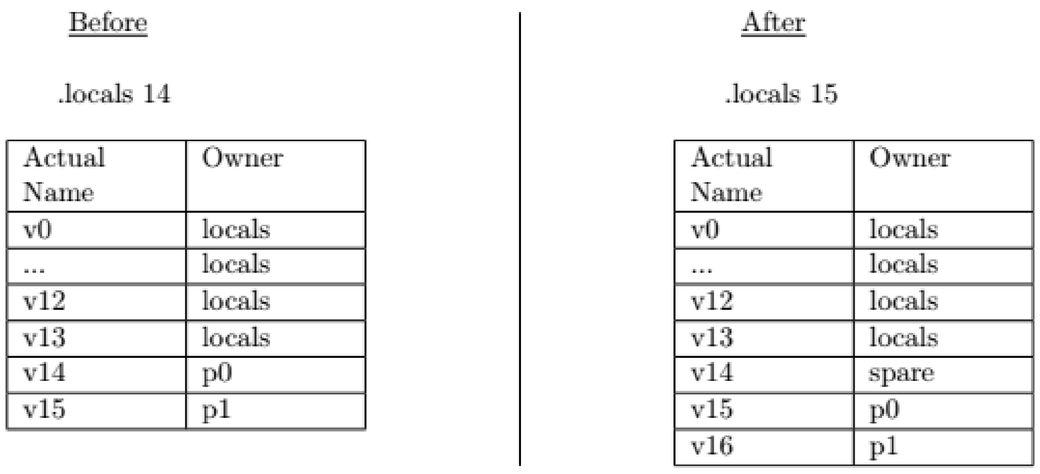
This produces the following two issues:

- Added instructions must be able to reference the inserted register. If the newly added register is outside of the 4 bit address registers, then it cannot be referenced using 4 bit address bytecodes.
- By increasing the number of locals by one, all parameters start one register higher than when they were originally converted to bytecode. A parameter that was previously inside the 4 bit address registers could be pushed out of them. The 4 bit bytecodes that were used previously to reference this would now cause the app to crash.

In Figure 2, adding an extra local register means that p1 used to reference a 4 bit address register. Instead this now references an 8 bit address register so must use the correct bytecodes to reflect this. Due to the numerical overlap, bytecodes for registers with an 8 bit address can reference all of the first 16 registers (those covered by the 4 bit address bytecodes). Consequently, when Orka inserts bytecodes it uses those that can address registers with 8 bit addresses. This way it can be sure the inserted bytecode will be able to access the register. The second issue proved tougher to solve.

It only presents itself in two situations:

- The sum of the registers used for parameters and locals is greater than or equal to 16, and the number of local registers is less than 16.
- If the sum total of the locals and parameters is 15, and the last parameter stores a double or long



Before

.locals 14

Actual Name	Owner
v0	locals
...	locals
v12	locals
v13	locals
v14	p0
v15	p1

After

.locals 15

Actual Name	Owner
v0	locals
...	locals
v12	locals
v13	locals
v14	spare
v15	p0
v16	p1

Figure 2: Demonstrating the effect of the bug in Orka, causing parameters to go beyond 4 bit address registers.

(both requiring two registers). Adding to the locals will push the second of the parameter's two registers out of the 4 bits address registers.

The injector resolves this issue with minimal intrusion into the code. After all of the declarations of the parameters within the method, the injector inserts instructions to move each back into their original register. This returns the registers to their original states. While a number of move statements are inserted, the injector does not change the instructions used (from 4 bit to 8 bit addresses). These move statements are only inserted if the injector detects a potential issue with the registers (as detailed above), therefore only changing the code if necessary.

Once converted to Smali, all parameters in a method are referenced using their *pX* name. This relates to the parameter number of the contents of the register (Smali, 2015). These are static, tied to the original register number containing the parameter's data when entering the method. If the data is moved to another register, the pX name would still refer to the original register. Using Figure 2 as an example, even if the contents of p0 is moved back to v14 and p1 to v15, the labels p0 and p1 will always reference registers v15 and v16 respectively.

The injector overcomes this by leveraging the naming convention of the registers in Smali code. Every register, no matter the contents, has an absolute name that can reference it. This is always of the format *vX*, with X being a unique number (Smali, 2015). The second scheme relates to registers that hold a parameter when the method begins. These can also be referenced with *pX* format. All *pX* registers have a *vX* reference that relates to them but not all *vX* registers have to have a *pX*. The *pX* names are static, and will always relate to the register they pointed to at the beginning of the method.

The injector creates a hash table to store each original *pX* mapping for the parameters against its type. It then uses this to calculate the required number of registers for the parameters and therefore the correct bytecode to move this (as different types require different bytecodes) (Google, a). Special consideration is needed for the first parameter, as this is not explicitly declared in the code. The injector maintains these mappings in another hash table, also taking care, if a parameter requires two registers, not to overwrite the contents of the second with a later parameter. This mapping hash table is needed as all references to parameters will use the *pX* name. As stated, this will now face the incorrect register. To solve this, as the injector parses each line of Smali code for a method, it changes all references of *pX* to their new corresponding *vX* name (using the mapping hash table).

A simple string matching algorithm was written to aid with this. As all keys have the same first character, the algorithm searches for the first occurrence of a 'p' in the line. Starting with the character after this, the characters are appended to a substring until the next non numeric character. As the substring contains all of the following numeric characters, there can be no mismatches (for example, p10 incorrectly matching with p1). The substring is then checked to see if it is a key in the mapping hash table. Positive matches have these characters replaced in memory with the corresponding mapping before being written to the output files. If the 'p' is not followed by a number, this step is ignored and the next instance of 'p' is searched for instead. This means that, 'p's appearing in bytecodes or method names are instantly discarded without having to check the map. As this algorithm acts on each line of the file, this should run with time complexity O(n). However, it is extremely unlikely that the worst case input (a string where every character matches) will occur due to the nature of the input. By searching for just ‘p’ large chunks of the line should be skipped, thus making the actual run time much faster.

4.1.4 Injection Logic

Having resolved all register issues, the injector will parse each line to look for API calls. Following locals and parameter declarations in Smali code is the ‘.Prologue’ declaration. Adding Smali code after this is the equivalent of adding a line to the source code. Specifically, adding code after this means that the freshly inserted code will be the first operation run. As this is the criteria for the *methodLog* method in the logging class, the injector will insert a call to this here.

Different bytecodes are used to invoke different types of methods (Google, a) and all of these begin with ‘invoke-’. As the injector parses each line, it attempts to match it to this. On succeeding, it passes this to a function that checks if the called method is an API. To invoke a method in Smali, the entire package name is passed in the instruction as well as the method. This helps the VM identify the method to be called (Ehringer, 2010). As all Android API’s belong to the Android package, the injector checks if the invocation references this package. Methods found to be API calls have their name extracted from this. By matching the package name, it frees Orka from the requirement of keeping an up to date documentation of the ever changing Android API names.

One line of source code does not necessarily relate to one line of Smali as Smali is a lower-level interpretation of the code. To cause as little interference with the code as possible, the injector stores all API calls in memory until its internal logic finds a safe place in the underlying bytecode. Smali retains information relating to the line numbers in the original source code. When the injector finds these, it knows it can safely insert methods to log all the API calls currently in memory. Before this is written to the output file, the injector inserts two new lines per API. The first adds the API name (a string) to the newly added register. The second adds an invocation of the API logger function, passing it the register containing the API’s name. This is done for every item in the list (the corresponding list entry is then deleted). This behaviour also occurs on finding statements corresponding to returning a value or leaving a method. As all API calls are tied to a method, they are logged before leaving it less they be attributed to the incorrect routine.

Special consideration was given for flow control statements. The injector is designed to place logging methods inside of loops, so that this is logged as many times as the loop runs. Loops are represented in Smali with a goto statement. The injector is designed to insert logging methods on finding statements closing a loop. Furthermore, the injector can correctly handle nesting loops. The list of API calls is implemented as a stack, being a list of lists. Orka maintains a pointer to the index representing the top of the stack. This is incremented on entering a new loop, pointing to the index in the list where new found API invocations should be added. On finding the end of a loop, the logging methods are inserted for all APIs in the list at the top of the stack. This list is then cleared, and the pointer is decreased, representing the top of the list being popped off the stack. Any further API call would then be added to the list now at the top of the stack (until the corresponding end of the loop is found).

Due to how Smali represents ‘if’ statements, the outlined logic correctly handles these. The bytecodes representing ‘if’ take the condition given in source code and reverse it. Those that now pass this reverse condition (failing the original) are ‘jumped’ over the code relating to conditions that satisfy the equality. The flags which are jumped to are placed after a ‘.line’ statement in the Smali, so Orka will insert logging methods before this.

When there are no more files to inject, the injector runs Apktool to recompile the application. It then digitally signs this using Jarsigner - a tool that allows digital signing of Java jar files, which can also sign .apk files (Google, c). This is required to run an application on an Android device (Google, c).

4.2 The Analyser

Once the app has been injected and recompiled, the analyser takes over. It is responsible for generating and analysing the results. First, it loads an emulated version of a Nexus 7 (2013 model) and the injected application is then installed on this.

During the course of development, we noted that emulators are considered as ‘charging’ when they initially load. *BatteryStatsInfo.bin* will not populate its data in this state as it only shows data since the last charge. The analyser addresses this by running a bash script that logs into the device via *telnet* and then changes the power settings to discharging. This uses Expect, a scripting language to script the response to and from *telnet*. To help increase the speed of the emulators, these are run as Kernel-based Virtual Machines (KVM). This moves the control of the hardware resources used by virtual machines from software to hardware, greatly improving the execution time (Stylianou, 2013). As the majority of the analyser’s execution time is spent loading and executing applications on these, the decision was made to use

KVMs to speed up execution.

4.2.1 Monkey Runner Testing

With the injected application installed on the emulator, the analyser loads the app and runs the user's provided Monkey Runner script. As this is using an injected version of the application, the data about API usage is being output to Logcat. This tests the programme for a scenario of the user's own design; they are free to use this tool for a variety of scenarios, from average use cases to stress testing their applications. This freedom will give Orka a greater flexibility as a testing tool. This does place the onus on the user to write good tests. However, the users of Orka will be developers who should be knowledgeable about testing and know how to write tests for their own applications.

On completion of the Monkey Runner script, Logcat contains the execution trace from the test execution and *BatteryStatInfo.bin* will be populated with the hardware costs. As all these have a unique tag, the relevant logs that relate to this can be easily accessed and pulled from Logcat (using Google's own tools). The analyser then calls Dumpsys, saving the human-readable output as a file, and then opens and retrieves the relevant lines relating to estimated hardware usage.

4.2.2 Handling the Data

Using the cost as per the findings of Linares-Vasquez et al (Linares-Vásquez et al., 2014), the equation to calculate software energy usage was:

$$\sum_{M_i \in Program} \left(\sum_{API_i \in M} (API_i \times c) \right) \qquad (1)$$

with M being each method in the program, API representing each API call made, and c being the energy usage. Hardware costs are stored in a table within the output from *BatteryStatsInfo.bin*. This provides a breakdown of the total energy usage of the application by component of the system, from which the analyser pulls the values.

5 IMPLEMENTATION PROTOTYPE

For the purpose of this research, Orka was deployed as a web application. This method circumvented restrictions placed on the developer regarding operating systems and hardware due to using KVMs. The web application was written using Flask[9], utilising Bootstrap[10] for aesthetics. Developers would upload their application and Monkey Runner script to Orka via the website. This generates a usage request that was added to a remote procedure call (RPC) queue. All results were converted to a serialisable data format in order to be transferred over via this queue.

Using the Pygal[11] module, the analyser generates graphs to present the results to the developer. Pie-charts are generated to show each method's energy usage as a percentage of the total usage as well as the breakdown of the energy usage for each hardware component used. An example of these using the simple application (used to fine tune Orka) which involved just one button which called the API to display a 'toast' message is shown in Figure 3 (showing breakdown with respect to methods in the application's code) and Figure 4 (showing breakdown in terms of hardware components used).

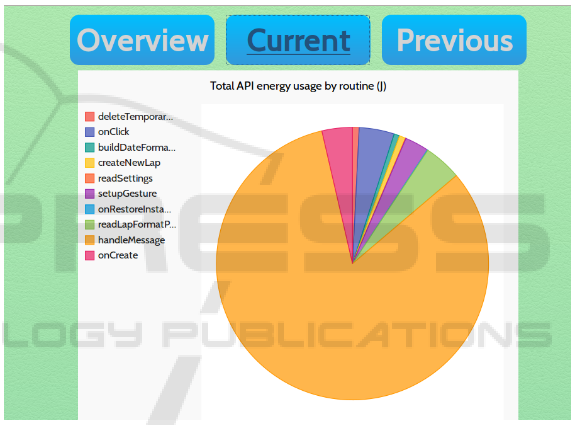


Figure 3: Screen-shot of Orka showing the breakdown by method of the application's energy usage.

Originally developers were also shown the total usage in terms of Joules. Following user feedback, these results were also presented in a manner that does not require an understanding of electronics. As such, Orka converts the total Joules used by a method into how long it would power the device for the following activities (shown in Figure 5), such as:

- Browsing the internet
- Watching a high definition video
- Playing a 3D game

Independent performance tests provided the number of hours each of these activities take to drain the battery (Shimpi, 2013). The number of Joules used

[9]http://flask.pocoo.org/docs/0.10/
[10]http://getbootstrap.com/
[11]http://www.pygal.org/en/latest/

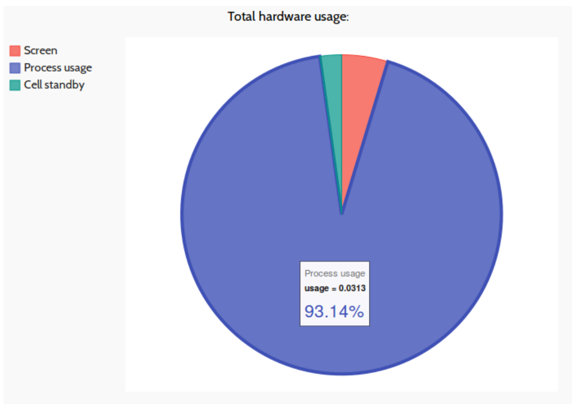
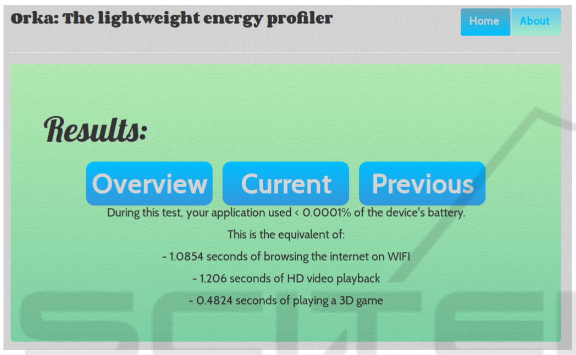


Figure 4: A pie-chart from an application's results. It shows the breakdown of energy consumption by components.

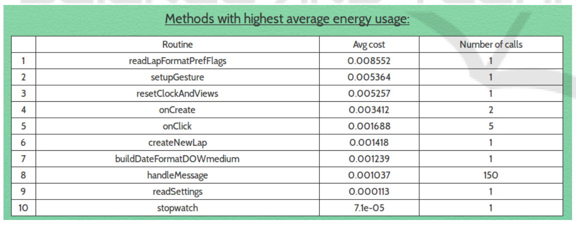


During this test, your application used ‹ 0.0001% of the device's battery.

This is the equivalent of:

- 1.0854 seconds of browsing the internet on WIFI
- 1.206 seconds of HD video playback
- 0.4824 seconds of playing a 3D game

Figure 5: Screen-shot of Orka showing the test case's energy usage in terms of other potential applications.



Methods with highest average energy usage:

	Routine	Avg cost	Number of calls
1	readLapFormatPrefFlags	0.008552	1
2	setupGesture	0.005364	1
3	resetClockAndViews	0.005257	1
4	onCreate	0.003412	2
5	onClick	0.001688	5
6	createNewLap	0.001418	1
7	buildDateFormatDOWmedium	0.001239	1
8	handleMessage	0.001037	150
9	readSettings	0.000113	1
10	stopwatch	7.1e-05	1

Figure 6: An example of the table showing the routines with the highest energy usage (results from Anstop application).

by both the entire program and a breakdown of each method is still presented to the developer, if they wish to use this.

When extracting data from Logcat, the analyser will maintain the number of times each method is called. Based on this, developers are presented with a table containing the ten routines with the highest average usage. This allows the analyser to highlight the routines that most likely contain energy bugs, rather than those with a low average usage cost but a high number of calls. An example of this (using the Anstop stopwatch application[12]) is given in Figure 6.

[12]https://code.google.com/p/anstop

The analyser also has the facility to track energy usage changes over different versions of the code. Cookies stored within the browser contain the previous total energy cost of the program and the ten methods with the highest energy usage. Developers are presented with a graph plotting these against the previous runs to allow comparison across versions. These are indexed by application name, allowing developers to store results from different applications yet only see those relevant to their current tests. This is very useful for developers to track how the energy usage of their application changes over the development life-cycle and improve it as necessary.

6 EVALUATION

By using the findings of Linares-Vasquez et al (Linares-Vásquez et al., 2014) and the Android OSs own internal hardware usage estimations, Orka was developed to provide new a new metric on which Android developers can test their code. Using Orka, a developer can have feedback on the energy consumption of their code, in both raw data and by comparing this to other uses of the device. These results are persistent across their development cycle, allowing them to see how changes to their code affect their application. However, what sets Orka apart from other energy profiling research and the systems currently available[13] is its independence from hardware. Previous research has focused on measuring energy usage with multi-meters attached to real devices (Hao et al., 2012). While this project could not have been completed without their findings to act as foundations, this research has decoupled the hardware by using emulators. Additionally, this system has been deployed so that it is not limited to users of specific software or operating system. This research shows that tools can be made to provide this feedback without requiring a developer to purchase any equipment.

In regards to the testing of the application, one of the goals was that tests should be valid and representative of real life, so Orka was tested using a variety of applications, such as:

- Anstop, a stopwatch application[14]
- Alarm Klock, an alarm clock[15]
- Acrylic Paint, a painting application[16]

[13]https://developer.qualcomm.com/mobile-development/increase-app-performance/trepn-profiler

[14]https://code.google.com/p/anstop

[15]https://f-droid.org/repository/browse/?fdid=com.angrydoughnuts.android.alarmclock

[16]https://github.com/valerio-bozzolan/AcrylicPaint

- Accordion, an accordion music program[17]

These applications were all downloaded from F-droid[18]. For each application, a Monkey Runner script was written to run tests upon it. These applications were chosen as they represented a variety of different styles of applications and had different developers, the latter being an important issue as different developers use different programming styles, and any of these styles could have caused issues with Orka. Applications were taken from F-droid rather than the Google play store as they were all open source. Having the source code was necessary to confirm that Orka was working correctly and for analysing problems in how its logic was treating the Smali code.

Being designed with the intention to be used by developers, Orka's website was tested by a few developers in order to get their feedback. Due to a departmental security policy, Orka's website could only be accessed by those on the internal campus network. This meant that all the testers were students of Imperial College London, rather than opening the tool to a worldwide beta testing by inviting members of online development communities for feedback. The chosen testers were all students in the Department of Computing. The decision to use Computer Science students was made since they would all have significant experience programming. This tool was designed to be used by developers, so prior programming knowledge was required to get meaningful feedback.

Initial user feedback was very positive. All those surveyed liked the idea behind Orka and thought that knowing energy usage of their code would be useful. Most importantly, all said they would use such a tool. Feedback was also positive towards the simple user interface. Testers liked that it only required that they upload their files, instead of having to configure the settings of the tool for it to work.

Testers were presented with a choice between the pie charts and animations to display energy usage (such as having leaves falling from a tree depicting the energy usage). The tester's feedback was that they preferred the scientific style of using pie charts and also felt the animations could lead to ambiguity.

In order to ascertain whether Orka would work in a real-life scenario, a simple proof of concept experiment was carried out. The scenario chosen was that of a developer who would build a simple Android application and use Orka to measure its costs and use the feedback to improve their application's energy usage. To measure the real costs of the application on the device (to check whether the changes really worked),

[17]https://github.com/billthefarmer/accordion
[18]https://f-droid.org/

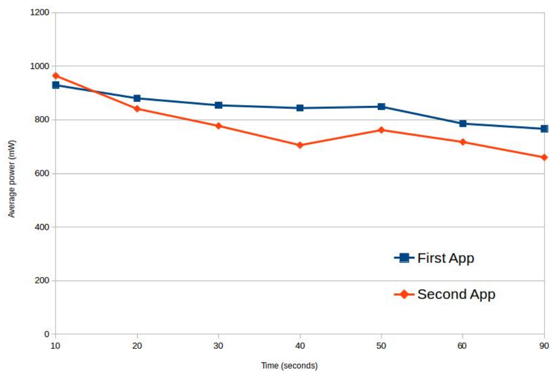


Figure 7: Average measured power consumption of the applications used for testing the proof on concept.

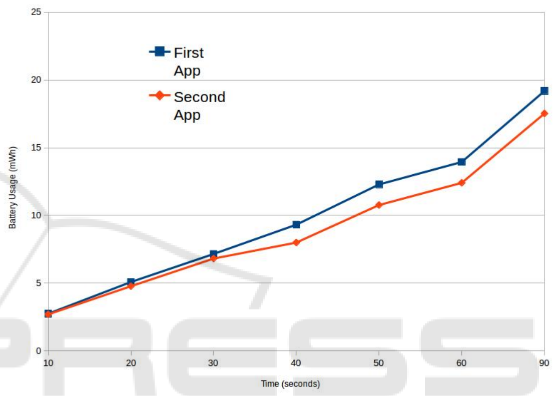


Figure 8: Measured energy use of the applications used for testing the proof of concept.

we use Trepn[19], a power profiling tool that measures an Android application's battery consumption.

Two simple test applications were written for this. 'First App' contained a 'programming error' in that a routine meant to be called only once was inadvertently called within a loop. Orka measured the applications energy usage as 0.42193334J and highlighted the higher than expected number of calls to this method. Using this feedback, this error was corrected by the developer in the 'Second App', which Orka registered as having an energy usage of 0.008439J (significantly lesser).

To measure the real energy costs, both applications were installed onto a Sony Xperia Z1 smartphone (available at the time) and the actual energy consumption was checked using Trepn. Even though Orka ran on an emulator running Nexus 7, the information provided (since it is likely the API costs will be in the order of magnitude across different devices) was useful to the developer so that they could reduce

[19]https://developer.qualcomm.com/software/trepn-power-profiler

the energy costs of their application, as shown in Figures 7 (measuring actual power usage) and 8 (measuring actual energy consumed).

7 LIMITATIONS

This approach has produced a trace of the energy usage of an application using tests as designed by the developer of the application. Orka handles the instrumentation of the application, removing the need for developers to learn any new skills to receive feedback. However, as Orka is not making readings based of discharge from the battery, this approach leads to estimation based on values. As stated, this comes from the findings of previously published research which have been further used in other studies.

The approach used in this paper attempts to identify the areas of source code with the highest energy usage. This approach does not attempt to accurately estimate the energy usage of an application, but rather to highlight potential trouble areas. This does lead to a known error in the total estimation. This was deemed acceptable as precise reading would require measuring the usage on an actual device.

Due to the branching nature of computer programmes, a programme's energy usage will change depending on the input and its effect on the flow of the program. Orka has pushed the writing of tests onto the developer of the application. This opens up the system to those who might try to 'game' it by writing tests that do not represent a typical usage of their application. Such concerns can affect any software testing that asks the developer to write their own tests. By allowing this, developers have a greater flexibility in the types of tests that can be run on their software (e.g. average use, stress testing).

By using code injection, extra code is being added to the applications by Orka, which will in turn have an effect on the application's total energy cost. The injected code calls an API, a process which is known to be the most energy costly code. While this method is not counted in the analysis of the energy cost of the application, it is factored into the data taken from Dumpsys and therefore in the breakdown by components. The method of the Log API was chosen as it had lower energy consumption than other log methods (Linares-Vásquez et al., 2014). This also only affects one part of the results.

Currently, Orka runs on the emulator running Nexus 7 (the 2013 model) and the results produced are a result of running it on this particular hardware. An extension to Orka here would be to enable it to perform this analysis on multiple different pieces of hardware. The proof of concept could also certainly be extended to include more complex applications.

8 CONCLUSIONS AND FUTURE WORK

Mobile technology has improved by leaps and bounds over the last few years, which has in turn led to a an increase in new, exciting applications. However, with this technological improvement comes an increase in power consumption. Increasingly, users are aware of this and hence, application developers need to ensure that their applications expand minimal energy. In order to achieve this, developers need tools to aid them in optimising their application's energy usage. This paper introduced Orka, which attempts to bridge this gap. Taking a developed Android application, Orka injects code into this to track calls made to APIs. This injected application is then run on an emulator, removing the need for the developer to purchase hardware. By running a specific test case written by the developer, Orka creates an execution trace from which the energy usage is calculated. Furthermore, energy usage due to hardware use is also captured. In this deployment, Orka was a web-based service, however in future iterations it could be a command-line system in order to allow it to function on games.

Orka stands as a platform on which a number of new research ideas could be formulated. It could be expanded to test an application on emulators representing different pieces of hardware. Due to the different sizes and resolutions of devices, a mapping would be needed to allow one Monkey Runner script to run on many devices. Furthermore, Orka could be expanded to allow developers to compare their application's performance to others of a similar category. As such, future research could also try to identify behaviours that all category of applications need to perform. These could be used to create a series of standardised tests for different types of applications, which would help mitigate the problem of malicious developers writing their own tests. This would also help novice developers who are less familiar with testing, as well as aiding more experience developers trust the rankings of their applications.

REFERENCES

Brooks, D., Tiwari, V., and Martonosi, M. (2000). *Wattch: a framework for architectural-level power analysis and optimizations*, volume 28. ACM.

Bunse, C., Höpfner, H., Roychoudhury, S., and Mansour, E. (2009). Choosing the "Best" Sorting Algorithm for Optimal Energy Consumption. In *ICSOFT (2)*, pages 199–206.

Casey, G. (2013). Inserting keylogger code in Android SwiftKey using apktool. https://www.georgiecasey.com/2013/03/06/inserting-keylogger-code-in-android-swiftkey-using-apktool/. Accessed: 2015-08-24.

Chaudhari, A. (2015). Mobile Applications Market Expected to Reach US$ 54.89 Billion by 2020 Transparency Market Research. http://globenewswire.com/news-release/2015/02/19/707887/10120995/en/Mobile-Applications-Market-Expected-to-Reach-US-54-89-Billion-by-2020-Transparency-Market-Research.html. Accessed: 2015-06-05.

Corral, L., Georgiev, A. B., Sillitti, A., and Succi, G. (2013). A method for characterizing energy consumption in Android smartphones. In *Green and Sustainable Software (GREENS), 2013 2nd International Workshop on*, pages 38–45. IEEE.

Dong, M. and Zhong, L. (2012). Chameleon: a color-adaptive web browser for mobile OLED displays. *Mobile Computing, IEEE Transactions on*, 11(5):724–738.

Ehringer, D. (2010). The Dalvik Virtual Machine Architecture. http://show.docjava.com/posterous/file/2012/12/10222640-The_Dalvik_Virtual_Machine.pdf. Accessed: 2015-08-24.

Google. Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. Accessed: 2015-08-24.

Google. Keeping the Device Awake. http://developer.android.com/training/scheduling/wakelock.html. Accessed: 2015-08-24.

Google. Signing Your Applications. https:// developer. android.com/ tools/ publishing/ app-signing.html. Accessed: 2015-08-24.

Hao, S., Li, D., Halfond, W. G., and Govindan, R. (2012). Estimating Android applications' CPU energy usage via bytecode profiling. In *Proceedings of the First International Workshop on Green and Sustainable Software*, pages 1–7. IEEE Press.

Heikkinen, M. V., Nurminen, J. K., Smura, T., and Hämmäinen, H. (2012). Energy efficiency of mobile handsets: Measuring user attitudes and behavior. *Telematics and Informatics*, 29:387–399.

Jabbarvand, R., Sadeghi, A., Garcia, J., Malek, S., and Ammann, P. (2015). EcoDroid: An Approach for Energy-Based Ranking of Android Apps. In *Proceedings of the 4th International Workshop on Green and Sustainable Software*, pages 8–14. IEEE Press.

Li, D., Hao, S., Halfond, W. G., and Govindan, R. (2013). Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89. ACM.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D. (2014). Mining energy-greedy API usage patterns in Android apps: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 2–11. ACM.

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

Rivera, J. (2015). Gartner Says Tablet Sales Continue to Be Slow in 2015 . https://source.android.com/devices/tech/power/index.html. Accessed: 2015-08-24.

Shimpi, A. (2013). The Nexus 7 (2013) Review - Platform Power and Battery Life. http://www.anandtech.com/show/7231/the-nexus-7-2013-review/2. Accessed: 2015-08-30.

Smali (2015). Registers. https://github.com/JesusFreke/smali. Accessed: 2015-08-24.

StackOverflow (2015). How to use Xposed framework on Android emulator. http://stackoverflow.com/questions/18142924/how-to-use-xposed-framework-on-android-emulator. Accessed: 2015-08-24.

Stylianou, C. (2013). Speeding Up the Android Emulator on Intel Architecture. https://software.intel.com/en-us/android/articles/speeding-up-the-android-emulator-on-intel-architecture. Accessed: 2015-08-14.

Wilke, C., Richly, S., Gotz, S., Piechnick, C., and Aßmann, U. (2013). Energy consumption and efficiency in mobile applications: A user feedback study. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, pages 134–141. IEEE.

Winchester, H. (2015). The best VR headsets. http://www.wareable.com/headgear/the-best-ar-and-vr-headsets. Accessed: 2015-06-05.

XDADevelopers (2014). Installing Xposed on the Android Emulator. http://forum.xda-developers.com/xposed/installing-xposed-android-emulator-t2794768. Accessed: 2015-08-24.