

BACHELOR'S THESIS COMPUTING SCIENCE

Reducing the Energy Usage of Android Applications

An Assessment of the Available Tools

PETER KLEIN HANEVELD
S1078661

October 13, 2025

First supervisor/assessor:

dr. Mart Lubbers

Second assessor:

dr. Bernard van Gastel

Radboud University



Abstract

Most programmers do not know how to optimize their code for energy efficiency. This is a problem, because most personal devices run on a battery these days, like Android-based phones. To support developers, there are tools that are made to help find energy related issues within the developers' software. In this thesis, we test and analyze the available tools on open-source Android apps and those tools on their performance, both in terms of *usefulness* and *usability*. We conclude that there are two good tools, both useful in their own right. This helps programmers pick the tool that fit their needs, but we also expose the weaknesses of the tools in general and show what still needs to be improved.

Contents

1	Introduction	3
1.1	The Research Question and Structure	3
2	Energy Problems	4
2.1	Hotspots or Bugs	4
2.2	Code Smells	4
2.3	Batching and Compression	5
2.4	Implementation	6
2.5	UI	6
2.6	Leaks	6
2.7	Wake Locks	6
3	Energy Reduction Tools	7
3.1	Shapes and Sizes	7
3.1.1	Methods	7
3.1.2	Presentation	8
3.1.3	Environment	8
3.2	The Tools to be Tested	9
3.2.1	Android Lint	9
3.2.2	aDoctor	9
3.2.3	EcoCode Android	9
3.2.4	Android Power Profiler	10
3.2.5	PowDroid	10
4	Methodology	11
4.1	Case Study	11
4.2	Testing	12
4.2.1	Usefulness	12
4.2.2	Usability	12
4.2.3	The Process	13
5	Results	14
5.1	Android Lint	14
5.2	aDoctor	15

5.3	EcoCode	16
5.4	Android Power Profiler	16
5.5	PowDroid	17
5.6	Comparisons	18
6	Related Work	20
6.1	Smells	20
6.2	Other Energy Issues	20
6.3	Some Solutions	21
6.4	Other Tools	21
7	Conclusions	22
7.1	Discussion	22
7.2	Conclusion	23
7.3	Future Work	23
	Bibliography	24
A	Tools & Apps	27
B	Results	28
C	Hardware	29

Chapter 1

Introduction

Unsurprisingly, most personal devices nowadays run in an energy limited environment, like smartphones, laptops and smartwatches. Where Android alone is used on more than three billion devices [10]. For these devices it is important that the software running on them is not using too much energy, since a quick draining battery means that it has to be plugged in more often, hindering the core feature of being a mobile device. Traditional methods of measuring the energy used by mobile devices are non-trivial and require expensive tools. Thus, these options are not always accessible for development teams. Despite this, one would expect that developers know the importance of energy efficiency and keep that in mind when making the software for energy limited devices. However, most programmers have not learned to optimize for energy efficiency and are not up-to-date on the biggest pitfalls and green coding practices [23].

In this thesis we test a number of currently available tools that promise to help find energy issues with Android apps (from here on called Energy Reduction Tools) and discover how well they work. This sheds light on the current situation of energy debugging and helps Android developers pick a suitable tool to help them understand and debug their code from an energy-focused point of view. Furthermore, it shows us the quality of the current tools and whether or not there is more research needed to make better tools.

1.1 The Research Question and Structure

After explaining different kinds of energy problems in Chapter 2 and discussing the tools we test in Chapter 3, we answer our research question:

RQ: What available energy reduction tool for Android is the best and what still needs to be improved.

How we accomplish this, is explained in Chapter 4, afterwards, we present our results in Chapter 5. Then we cover the related work in Chapter 6 and finally we draw and discuss our conclusion in Chapter 7.

Chapter 2

Energy Problems

There are many kinds of energy problems, i.e. software issues that cause an increase in energy consumption. Some problems have a significant impact and others less so. It is however highly dependent on the specifics of the software. This makes it difficult to properly compare different kinds of issues. In this chapter we discuss and explain some common issues which, ideally, energy reduction tools help discover.

2.1 Hotspots or Bugs

First of all, we differentiate between two kinds of energy wastage, *energy hotspots* and *energy bugs*. The difference is fairly simple, an energy hotspot is a process that uses more energy than it should and an energy bug is when energy is spent on something without function.

Let us take an hiking app as example. Imagine starting a walk and the app enables the GPS to track the route. However, after completing the walk, the GPS is not turned off. This would be a case of an energy bug, as the GPS is still consuming energy but its functionality is no longer used.

Now let us say that during the walk, every second, the app is sending data from that walk to its server for online storage. This unnecessarily uses a lot of energy, as the app could also save up a minute's worth of data and send it as a batch. This is an example of an energy hotspot.

2.2 Code Smells

The term *code smells* is another term for bad practices, which can impact performance or indicate a deeper problem with the code. Unlike a bug, a code smell does not alter the outcome of the code, whereas conventional bugs cause unintended behavior.¹ There are a large number of code smells,

¹Please note the difference between a conventional program bug and a energy bug as explained above.

some code smells increase the energy consumption and thus are classified as an energy hotspot. Code smells can be corrected by refactoring. Here are some smells that, in certain cases, have been shown to decrease energy consumption when refactored [4, 7, 22, 25].

- **Duplicated Code (DC)** is when the same piece of code is present on multiple places when it could just be a method that is called from those places.
- **Type Checking (TC)** is when the type of objects or variables is repeatedly checked where it is not necessary.
- **HashMap Usage (HMU)** is when a hashmap is used while better alternatives are available. Especially for Android, there are other maps that are more memory efficient and cause less garbage collection like `arrayMap` and `sparseArray`.
- **Internal Getter/Setter (IGS)** is when trivial getters and setters are used. In Android these use more resources than just directly accessing the variable.
- **Member Ignoring Method (MIM)** is when a non-static method does not access any attributes. Calling a static method is faster, so when possible, it should be used.
- **Feature Envy (FE)** is when a method is constantly getting values from another class, indicating that it maybe should be part of that class instead.

2.3 Batching and Compression

As aforementioned, batching data before processing it decreases energy usage. This is because starting and finishing an operation costs energy, so called tail energy consumption. Combining multiple operation into one minimizes how often a process has to be started and thus reduces the total tail energy consumption.

When transmitting data it is good practice to, in addition to batching, compress the data beforehand. Because data compression is relatively energy efficient in comparison to the services generally used for transmitting data: Wi-Fi, Bluetooth or cellular data. These services require a significant amount of energy. Compressing the data, decreases the duration these services are active and thus decreases the energy used [9, 17].

2.4 Implementation

The way an algorithm is implemented also has impact on the energy performance. For example, it has been shown that recursive algorithms are not as well supported by Android as iterative solutions, and can have a negative impact on the (energy) performance. Whereas using parallelization can greatly decrease the energy consumption [8]. These implementation choices are highly dependent on the kind of software that is written and it is up to the developer to make a well educated decision.

2.5 UI

While not being computationally or algorithmically heavy, the UI has a large impact on energy usage as it directly relates with what is shown on the display. Certain screens are optimized to consume less power when displaying darker colors [8, 16]. So accounting for that when designing an program prevents a significant amount of battery drain for some devices.

2.6 Leaks

A leak occurs when a resource is used but not released afterwards. Not properly closing such a resource can cause it to still consume energy in the background while not being used, i.e. an energy bug. An example of such a resource is a database, before interacting with a database, a connection needs to be made. If that connection is not closed afterwards, it still uses power. The same goes for sensors, not disabling the GPS or camera after using it will quickly drain the battery [1, 17].

Another kind of leak is a context leak, which occurs when a context is no longer needed but is still accessible from an active part of the code, preventing it from being cleared from memory and still consuming energy.

2.7 Wake Locks

A wake lock allows a program to take direct control over the power state of a device. For example explicitly increasing the screens brightness or disabling power saving mode. Developers have to be careful when using wake locks, as misuse can significantly increase the energy usage. Problems can arise if the wake lock is acquired too early or released too late. Not releasing it at all (i.e. a leak) is especially bad [2, 17].

Chapter 3

Energy Reduction Tools

Energy reduction tools differ in quite some ways from each other. These differences makes it hard to objectively compare them all. Focusing on just the tools that are suitable for this case study, makes it easier to properly compare them. Despite that, we discuss many different kinds of tools, so we have a more complete picture of energy reduction tools in general, after which we discuss the tools tested in this thesis.

3.1 Shapes and Sizes

Just like there are many different kinds of bugs, there are different kinds of tools that try to detect them. The main functional differences between these tools is their method and their presentation of the results. Where some tools work by scanning the source code and highlight problems within specific lines of code, others monitor and analyze the energy usage and indicate inefficient packages. These differences are highly related to the level of abstraction each tool operates on. All of these features have their benefits and drawbacks. The kind of tool that is useful for a developer is highly dependent on the kind of software, the kind of work environment and kind of issues they want to detect. Another important aspect is the programming language each tool supports. Some tools work independently of the language, while others just support one.

3.1.1 Methods

There are different methods to analyze software. A common approach is static analysis (SA), a simple example of this is the default error checker of your everyday IDE, like Android Lint which is part of Android Studio. This method analyzes the source code for possible bugs and generally provides some kind of suggestion on how to fix it.

Another method is measuring the energy usage when the code is running.

This can either be done externally or internally. External measuring (EM) is done by hooking up some kind of wattage meter to the device running the software and is generally quite an elaborate process, requiring expensive tools. Alternatively, internal measuring (IM) is done within the device, similar to how smartphones can show the battery usage of each app. Internal measuring is less convoluted and does not require expensive equipment, making it more accessible for developers. Either way, both kinds of measurement techniques require a device to run the app, in the context of this research, an Android based smartphone.

Instead of measuring, some tools estimate (ES) the energy usage, by keeping track of what hardware is triggered to run each piece of code.

The way IM, EM and ES tools are used is quite different from SA tools. These three methods are mainly useful for A/B testing, a research method where two implementations are tested and compared to find out which is better. In the context of green coding, an example is comparing two implementations of an algorithm to find the most energy efficient one.

3.1.2 Presentation

The way the results of the tool are presented is highly dependent on the method the tool uses. An SA tool generally just gives error/warning messages coupled with highlighted source code, like the conventional error checker in most IDEs. However, this is not necessary, as some tools also incorporate graphs to convey additional insights.

Generally graphs are used to display EM, IM or ES based methods. Sometimes these graphs are made to compare with similar graphs from other versions of the application. Other times they are linked in some way to the executed code, making it possible to easily analyze the energy profile of the software and detect energy intensive parts.

These different presentations can all be useful, suggesting that it is optimal to use a combination of tools that complement each other.

3.1.3 Environment

Not all tools function in or are suitable for all work environments, some tools are built into the IDE whereas others are part of the deployment pipeline. These differences do not directly impact their effectiveness, but they do matter in terms of usability. Some tools work on the device of the individual programmer, so it is their responsibility to use it correctly. Whereas a tool that automatically scans all code before it is deployed is accessible for a whole team, requiring each member to keep the energy usage in mind and allows the whole team to view the results. This method also makes it easy to compare different versions of a program.

Not all methods from Section 3.1.1 work in all environments. For example, it is impossible to run a static analysis on a production server. This further suggest that ideally, a developer uses multiple tools in tandem with one another for the most complete overview.

3.2 The Tools to be Tested

We only extensively analyze tools that are suitable for this case study. Which are tools that employ either SA, IM or ES, are usable for an individual developer and work for Android development with Kotlin.¹ There is however one exception, namely EcoCode Android, as it seems likely it will get Kotlin support at some point in time.

3.2.1 Android Lint

Android Lint is a build in error checker for Android Studio, it is not designed for energy debugging but since it is likely used by all Android developers already, it serves as a nice baseline for other static analysis tools.

3.2.2 aDoctor

aDoctor [21] is a plugin for Android Studio, accessible from the plugin manager. This makes it really easy to get started. It checks the source code for 6 specific energy issues: IGS, MIM, HMU, inefficient data structures, leaking threads and wake locks. It also suggests a refactor for each issue.

3.2.3 EcoCode Android

EcoCode² [18] Android is part of the Green Code Initiative, a French initiative that develops energy reduction tools for a range of different languages. They mainly make and maintain plugins for SonarQube, a professional software analysis platform that automatically performs code-quality checks. The EcoCode Android plugin adds 42 rules for energy related issues, covering a big part of the issues mentioned in Chapter 2. It is important to know that SonarQube requires the source code to be compiled, which means it functions a bit different from other SA tools.

Unfortunately EcoCode does not (yet) support Kotlin, as such it is not possible to test it as extensive as the other tools. Despite that, we feel it is worth it to include this tool, due to the apparent quality and the chance that a Kotlin version will be released at some point.

¹Since 2019 Google recommends the use of Kotlin over Java for Android development, meaning that not all Android apps are written in the same language. Not all tools support Kotlin (yet), but as it is the recommended language, it will be the focus of our study. Unfortunately, that excludes the analysis of some promising tools.

²Also known as Creedengo.

3.2.4 Android Power Profiler

This feature of Android Studio can measure the energy usage of Google Pixel smartphones when running an app. After the run it compiles the data into a timeline, showing how much energy each part of the device used each moment.

3.2.5 PowDroid

PowDroid [5] is a stand alone program that can estimate the energy usage of a mobile device. It can do this by connecting both before and after a test run, which means that there is no overhead during testing. It returns a table with relevant energy metrics, like energy used while an app was running on the foreground and whether certain services were used. Unfortunately, this is the only selected tool that does not run on windows.

Chapter 4

Methodology

As made clear in Chapter 3, there is a significant difference between different tools. This means that it is not possible to have a one-size-fits-all approach. In this chapter we explain the method used to analyze each of the kinds of tools appropriately to allow for an as fair as possible comparison. Furthermore, we explain the process used for picking the case studies used for testing the tools on.

4.1 Case Study

It is important to use suitable case studies to test the tools on. To ensure that, the following criteria are used:

1. The apps used need to be representative of applications that are actually commonly used. For example, a calculator app is not nearly as often used as a messaging service, so it makes sense to prioritize that the latter is properly optimized.
2. The apps used need to be relatively large apps. Because the chance that there are energy issues in a large code base is higher than in a small app.
3. The apps need to be open source. Since we need access to the code in order to run the SA tools.
4. The apps need to be written in Kotlin, as it is the recommended language for Android development.

In order to get representative cases, we took one app related to the five most popular used categories of apps. Picking five apps also increases the total amount of code. A study from Han et al. [15] found that the five most common categories are *communication*, *web*, *game*, *music* and *social network*. This led us to pick the following apps: Thunderbird, Duck Duck Go, Freebloks, Fossify Music Player and Signal.

4.2 Testing

We use the available energy reduction tools on the selected apps' source code and score them. The criteria we use to score them consists of two parts: *usefulness* and *usability*.

4.2.1 Usefulness

A *useful* SA tool should be able to catch or detect many kinds of issues with the code, preferably the ones with the largest impact if not all of them. Like we discussed in Chapter 2 there are a lot of different kinds of possible problems with software that increase the energy consumption. We keep record of all the problems a certain tool helps finds so we can compare the kinds of energy issues each tool detects. This methods allows us to adapt to the issues we find instead of looking for specific problems. However, we have to keep in mind that when using this method, we assume that the source code the tests are run on is not fully optimized in the first place.

IM, EM or ES based tools work significantly different from SA tools, these tools require a baseline to compare their results to, human interpretation and knowledge of the code base it is used on. So a *useful* IM, EM or ES tool give insight in how the energy is spent, as to give the user an idea of wether a part of the program is using an excessive amount of energy and where possible issues might be.

4.2.2 Usability

A *usable* tool should be user friendly, i.e. easy to use and set up. It should give good insight on the energy usage of the software. It should be relatively quick, and be easily incorporable into a proper workflow. There should be a good balance between specificity and global overview. These factors, from here on called *setup*, *runtime*, *output*, *workflow* and *knowledge*, are fairly subjective and are judged by assigning each a score (- ; ±; +), accompanied by a short argumentation, backing up the score. The criteria for usability factors can be seen below:

Setup How easy it is to install and set up the tool.

- + It takes a couple of clicks to install the tool and no specific hardware is required.
- ± It takes significant effort or specific hardware to set up the tool.
- It takes significant effort and specific hardware to set up the tool.

Runtime How long it takes to run the tool.

- + The tool detects problems immediately.
- ± The tool takes up-to a couple of minutes to run.
- The tool takes longer than a couple of minutes to run.

Output How good the output is in terms of readability and specificity.

- + The tool clearly shows the results and allows for in depth analysis.
- ± The tool shows the results.
- The tool does not properly show the results.

Workflow How well it incorporates into a workflow, taking into account aspects like Git support and usability for teams.

- + Using the tool fits right in the development process.
- ± Using the tool require a couple extra steps in the development process.
- Using the tool requires an alternative development process.

Knowledge How much knowledge is assumed the user has.

- + The user does not need to know anything about green coding.
- ± The user needs to know the basics of green coding.
- The user needs advanced knowledge of green coding.

4.2.3 The Process

The steps we go through for each tool to test them is as follows:

1. Install the tool, score *setup*.
2. Run the tool on the selected versions of the case studies, score *runtime*.
3. Analyze the output, score *usefulness*.
4. Evaluate the output, *output*.
5. Evaluate the process, score *workflow* and *knowledge*.

After testing and scoring all tools, we compare them and discuss what tools is the best and what further improvements need to be made.

Chapter 5

Results

In this chapter we show the result of using the approach discussed in Chapter 4. For each tool there is a section covering the analyzed criteria accompanied by relevant comments about the tool and testing process. Additional quantitative results are available in Appendix B and the specifications of the hardware used is available in Appendix C.

5.1 Android Lint

Usability

- Setup + Android Lint comes preinstalled with Android Studio, so most Android developers already have access to this tool. If not, it is just a case of downloading Android Studio and opening your project.
- Runtime ± The average runtime is 71 seconds, however, the runtime differs significantly per project, although, consecutive runs cost less time. As it would not have to index all the files anymore. Also it is unlikely that for large projects a developer regularly runs Lint over the whole project rather than the part they are working on.
- Output + The tool clearly indicates the issues it found, where they are located and generally suggests a solution.
- Workflow + The tool can easily be ran after a developer makes some changes, so it does not require a different workflow. It can also be made to run automatically during the build process for even better incorporation into the workflow.
- Knowledge ± The knowledge required is highly dependent on what you want to use the tool for. Just some basic scans are understandable for every developer. However, with enough knowledge, a developer can write

their own checks to include. Which means there is a lot of potential and a high skill ceiling.

Usefulness

Android Lint not a useful tool for energy debugging. None of the issues it found were related to energy usage.

5.2 aDoctor

Usability

- Setup + As it is an extension for Android Studio, it is just a case of browsing the extension manager and clicking install.
- Runtime ± The average runtime is 119 seconds, however the runtime greatly differs for each application. Where the smallest apps take virtually no time and the largest nearly 10 minutes. However, big projects are rarely made alone, so as long as each developer runs the tool on the part of the code they are working on. It should not take a significant amount of time.
- Output ± The results are easily viewable. For each issue it highlights the corresponding code and presents a possible solution. However, it can be difficult to see all issues when there are a lot of them, since it is a single dropdown list with no option to filter or sort the issues.
- Workflow + The usage does not require an alternative workflow, the developer can just run the tool every so often or before committing as a final check.
- Knowledge ± While the tool does briefly explain what the issue is, it does sometimes recommend fixes that would break the code. Therefore a developer cannot blindly trust the suggestions.

Usefulness

The tool found energy issues in three of the tested applications (Thunderbird, DuckDuckGo and Signal). It mainly found cases of MIM across all three apps. Furthermore it found a case of IGS and HMU in Signal.

The tool found two cases of IGS in Thunderbird, however, manual inspection indicated that these were false positives since the setters in question were nontrivial. An incorrect refactor was suggested for a wake lock issue, due to the complex usage of the wake locks. Even though the fix was not useful, it is an indicator that the developer should take a look and see if there is a better way to implement the wake locks.

5.3 EcoCode

Usability

- Setup ± It took significant effort to get SonarQube to run, after which it also took quite some time to integrate the EcoCode plugin. This was mostly due to few and conflicting online documentation.
- Runtime ± Since EcoCode does not support Kotlin, it could not be tested on the selected apps. As such, there is no fair comparison to be made. However, based on limited tests, it seems fall in the ± category.
- Output + The results are shown in a neat report, including an easily navigable list with issues. The results are compared to previous runs, allowing the user to keep track of the progress.
- Workflow + While using SonarQube requires the application to be built, they have great support for managing issues across a development team. In addition, there is support for integrating with platforms like GitHub.
- Knowledge + The results come with a detailed explanation and are given a severity score. This means that the user does not need to be up to date with green coding to make good use of this tool.

Usefulness

Since EcoCode does not support Kotlin, there is not a list of issues it found. As such, it is not useful for energy debugging. However, its extensiveness does give us insight in what a proper SA energy reduction tool looks like.

5.4 Android Power Profiler

Usability

- Setup ± The tool is only useful for energy analysis when using a Google Pixel 6 or newer. Apart from that the setup is fairly easy.
- Runtime ± The runtime is directly related to how long the user is testing the application for. As the runtime includes the time spent recording and the time spent generating the report. However, a user generally wouldn't record for longer than a couple of minutes, as a short recording already results in a lot of data.
- Output + The report shows how much energy each part of the device used every quarter second. This allows the user to properly analyze the results.

Workflow ≠ To run the tool, the app has to be built and installed on the mobile device, thus making a change and quickly checking its performance is not possible, especially when working on large projects. Using the profiler, the code is a bit slower due to the overhead. But whenever the user would run the project, they could run it using the profiler and get more insight.

Knowledge - The tool cannot show what issues there might be, it only allows the user to identify actions that use a lot of energy. So the developer needs to have a lot of knowledge about green coding and the code base itself to use the tool optimally.

Usefulness

The results are very in depth. It shows a timeline of the test run, indicating the global battery stats (capacity, charge and current). Furthermore it shows how much power each part of the device (like the CPU, camera and display) was using at any point during the run. This data is very useful for finding leaks, especially since it also shows what each thread is doing and what events it handled during a given time.

There is not an easy way to compare consecutive runs to each other, making it somewhat difficult for in depth A/B testing. In addition, the energy measurements are done at the device level, as such, there is some background noise. However, due to the extensive thread report, it is possible to filter out moments of excessive energy usage from background processes.

5.5 PowDroid

Usability

Setup + While a mobile phone is needed to use PowDroid, no specific brand is required. The rest of the setup is fairly simple.

Runtime ≠ The runtime is directly related to how long the user is testing the application for, but a test run of at least five minutes is suggested in the documentation. Then, after recording, the tool needs some time to generate the report.

Output ≠ The output is a simple table.

Workflow - To run the tool, the app has to be built and installed on a mobile device, thus making a change and quickly checking its performance is not possible, especially when working on large projects. Additionally, to get repeatable and comparable results, one might need to use a macro for testing. Which takes a lot of time to set up.

Knowledge \neq The results themselves are not hard to interpret, however using them to decrease the energy usage is not trivial. Because PowDroid only suggest there might be problems instead of concretely pointing them out. Thus some knowledge is required to find and fix the related issues.

Usefulness

The results of the tool are very rudimentary, indicating data like estimated energy used during the test run. Besides that it indicates wether certain services were used like: screen, GPS, WiFi, camera, audio and wake lock. It is important to note that PowDroid estimates the energy usage of the whole device and correlating it to the application that was on the foreground at that time.

PowDroid can be useful to detect services that should not be used. The energy data can be used for A/B testing, however, the app needs to be tested with a macro in order to ensure consistent tests, adding more difficulty. Still then the results are not completely accurate due to noise from the device and the fact that it is based on estimations.

5.6 Comparisons

Here we compare the scores of each criteria.

Setup

In terms of set up, Android lint takes the least effort. As it is already part of the IDE many Android developers use. With only a couple clicks more, aDoctor is nearly as effortless. PowDroid is also straight forward to set up, while it needs a smartphone to work, it does not require a specific model. While Android Power Profiler is integrated into Android Studio, but does require a specific make smartphone to be used, which can be a hurdle for solo developers. EcoCode took the most effort to get working, which was mostly due to SonarQube's unclear installation documentation.

Runtime

The runtimes of all tools are the same order of magnitude. For the SA tools it only takes time to scan the source code, whereas the IM and ES tools need time to do a reading and time to compile the data into the output.

Output

For the measurement tools there is a difference in quality of output, but this directly related to the amount of data there is to show. PowDroid does not measure much data, so it is acceptable to use a simple table, whereas

Android Power Profiler collects magnitudes more data points, as such, it has a smarter way to display said data. For the SA tools we see a difference in quality as well. While all tools do display a list of found issues, aDoctor's list is not as easy to navigate. Whereas EcoCode's list was clear and also provides a lot of support for keeping track of said issues throughout different versions of the code.

Workflow

The measurement tools are inherently less easily incorporable into a workflow, as you need to run the tools to test them. Whereas Android Lint and aDoctor can be quickly ran before committing to check for issues. EcoCode, while it needs compiled code in order to work, has great support for teamwork and issue assignment, allowing for professional use.

Knowledge

The knowledge required to fully operate each tool varies. EcoCode gives the best insight into green coding, clearly pointing out the issues and giving a clear explanation as to why it is an issue. Android Lint and aDoctor both point to the issues and give some explanation. However, Android Lint's explanation is not always clear and aDoctor gets some false positives meaning the user needs to pay attention. PowDroid's output is easy to understand due to the rudimentary nature of the data. Android Power Profiler expects the user to be knowledgeable. In order to understand the data and make the code more efficient one needs a solid understanding of green coding.

Usefulness

As expected, both Android Lint and EcoCode are not useful energy reduction tools for Android, because they are not made for energy related issues or do not support Kotlin respectively. Whereas aDoctor was able to find issues and generally provided a solid solution. As for the measurement tools, both Android Power Profiler and PowDroid have their use cases. Android Power Profiler provides lots of data, allowing a skilled user to find and fix a lot of energy issues. PowDroid is useful for detecting needlessly used services and for A/B testing.

Chapter 6

Related Work

To our knowledge there is no prior research on the comparison of energy reduction tools for Android, even though the Android guidelines indicate the importance of energy efficiency [3] and studies show that programmers are generally not knowledgeable on the subject of energy efficiency and could use proper tools [23, 24]. This chapter indicates the research that has been done surrounding our topic.

6.1 Smells

There has been quite some research done into the effect of code smells on energy usage. Anwar et al. [4] found that refactoring certain code smells increase energy efficiency, but that the order in which they are refactored matters. The DC and TC smells proved to have the biggest impact. They also found that refactoring did not affect the execution time. Carette et al. [7] researched the effect of three smells (HMU, IGS, MIM) and found that refactoring these smells is generally beneficial for the energy usage. Likewise, Palomba et al. [22] and Verdecchia et al. [25] also found that refactoring certain code smells benefits the energy usage.

6.2 Other Energy Issues

Other kinds of energy issues were also investigated. Khan et al. [17] studied and discussed different kinds of leaks, and Liu et al. [19] found that leaks are the second most common kind of bugs in apps after GUI lag. Hort et al. [16] covers a wide range of performance metrics, the part about energy mentions anti patterns, bad practices, UI colours and power states as factors with an impact on battery life. Chen and Zong [8] tested aspects of an Android application like program language, compiler and implementation choices. They found that Android is better at handling iterative code than recursive and that Android software can benefit a lot from parallel execution. Cruz

and Abreu [9] compiled a large list of green coding practices after analyzing a lot of mobile applications.

6.3 Some Solutions

Some studies explored solutions for improving the energy usage of software. Gupta et al. [14] developed a method of comparing power traces with execution logs to find out what modules have the biggest impact on energy consumption. And Bunse et al. [6] made an optimization that automatically chooses the best sorting algorithm for a sorting task. Striking a balance between runtime and energy efficiency.

6.4 Other Tools

There are more energy reduction tools available than the ones covered in this thesis. However, these tools are not suitable because they either do not support Kotlin or are old and did not work anymore.

Green Android Lint is an expansion of Android Lint, which adds checks for inefficient energy practices [12]. AEON is an Android Studio plugin that adds an energy debugging environment into the IDE, with features like SA, ES and profiling [13]. PETrA is an ES tool that uses random interactions to automatically stress test an application [11]. Finally, E-Debitum is a plugin for SonarQube that employs SA to give the analyzed code an *Energy Depth* score in order to quantify the energy efficiency of applications throughout their versions [20].

Chapter 7

Conclusions

The conclusion of this thesis is threefold: a discussion, the conclusion and future work. We mention the assumptions and choices we made and explain why they are justified. Then we answer our research question by interpreting the results. Finally we lay out what still needs to be done in the future.

7.1 Discussion

During this thesis, we made some choices and assumptions. Firstly, we chose to focus on Android development using Kotlin. We feel this choice is justified because Kotlin has been the recommended language for six years now. Despite this decision, we still included EcoCode Android in our research. We did this because we deemed both the tool and platform it ran on interesting enough to analyze it to some degree.

We also chose to apply different usefulness criteria to the SA and the measurement based tools. This was needed because of the inherent differences between the approaches to handle energy issues, as outlined in Chapter 3.1.1

Another decision was that we tested all tools on a personal device, this was out of necessity, as there were no other devices available. Also, PowDroid did not work on Windows, so it was tested on a different (less powerful) device. This is of course not ideal for testing because it makes the research less repeatable. However, we feel that these decisions have not significantly impacted the results, as the criteria that is impacted the most is the runtime, which is scored in such a way that some variance does not make a difference.

Lastly, we made the assumption that the apps we selected had some energy issues. We think this is a fair assumption, as most developers do not take energy usage into account when programming, as mentioned in Chapter 1. Furthermore, the tests are not about the selected applications, but are about the tools. Since all tools were used on all same apps, no single tool was benefited by this.

7.2 Conclusion

There is no single best energy reduction tool, since both static analysis and measurement solutions are required for exhaustive testing, and there is currently no tool that provides both. Which means, that developers will need a combination of at least two tools for proper energy debugging.

Android Power Profiler is the best measurement tool. The set up is trivial and it is incorporable into a proper workflow. Its extensive output gives a lot of insight and allows for in depth analysis. This extensiveness is also its main drawback, because interpreting and using the data requires a good understanding of the code base and knowledge of green coding practices.

The best static analysis tool for Android is aDoctor. The easy set up, manageable runtime and easy to understand output make simplicity its biggest strength. Allowing all Android developers to make use of this tool and learn something about green coding in the process. Its main drawback is the fact that it only detects six energy issues and seems to not be updated anymore.

Despite the fact that aDoctor and Android Power Profiler are very useful tools, there are still some things left to be desired. Android Power Profiler would benefit from more documentation, lowering the required knowledge. Further more, a way to properly compare runs will allow for easier A/B testing. Concerning aDoctor, what has in simplicity, it lacks in extensiveness, making EcoCode a great alternative once it has Kotlin support.

7.3 Future Work

There needs to be more research done into energy reduction tools, as most tools are already fairly old and are not regularly updated. Furthermore, a deeper analysis of Android Power Profiler would be interesting. As this thesis did not allow us to use it to its fullest potential.

Bibliography

- [1] Android. Android developer - sensormanager. <https://developer.android.com/reference/android/hardware/SensorManager.html>. Last accessed: 2025-01-30.
- [2] Android. Android developer - wake lock. <https://developer.android.com/develop/background-work/background-tasks/awake>. Last accessed: 2025-04-2.
- [3] Android. Android developer - core app quality. <https://developer.android.com/docs/quality-guidelines/core-app-quality>. Last accessed: 2025-01-23.
- [4] Hina Anwar, Dietmar Pfahl, and Satish N. Srirama. Evaluating the impact of code smell refactoring on the energy consumption of android applications. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 82–86, 2019.
- [5] Fares Bouaffar, Olivier Le Goer, and Adel Noureddine. PowDroid: Energy Profiling of Android Applications. In *2nd International Workshop on Sustainable Software Engineering (SUSTAINSE)*, Melbourne, Australia, November 2021.
- [6] Christian Bunse, Hagen Höpfner, Suman Roychoudhury, and Essam Mansour. Choosing the "best" sorting algorithm for optimal energy consumption. volume 2, pages 199–206, 01 2009.
- [7] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. Investigating the energy impact of android smells. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 115–126, 2017.
- [8] Xinbo Chen and Ziliang Zong. Android app energy efficiency: The impact of language, runtime, compiler, and implementation. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BD-Cloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom) (BDCloud-SocialCom-SustainCom)*, pages 485–492, 2016.

- [9] Luís Cruz and Rui Abreu. Catalog of energy patterns for mobile applications. *Empirical Software Engineering*, 24, 08 2019.
- [10] David Curry. Android statistics (2025). <https://www.businessofapps.com/data/android-statistics>. Last accessed: 2025-07-14.
- [11] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Petra: A software-based tool for estimating the energy profile of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 3–6, 2017.
- [12] Olivier Le Goaër. Enforcing green code with android lint. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 85–90, 2020.
- [13] David Gonzalez. Aeon: Automated android energy inspection. <https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection>. Last accessed: 2025-09-24.
- [14] Ashish Gupta, Thomas Zimmermann, Christian Bird, Nachiappan Nagappan, Thirumalesh Bhat, and Syed Emran. Mining energy traces to aid in software development: an empirical case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14*, New York, NY, USA, 2014. Association for Computing Machinery.
- [15] Sang Pil Han, Sungho Park, and Wonseok Oh. Mobile app analytics: A multiple discrete-continuous choice framework. *MIS Quarterly*, 40(4):983–1008, 2016.
- [16] Max Hort, Maria Kechagia, Federica Sarro, and Mark Harman. A Survey of Performance Optimization for Mobile Applications. *IEEE Transactions on Software Engineering*, 48(08):2879–2904, August 2022.
- [17] Muhammad Umair Khan, Shanza Abbas, Scott Uk-Jin Lee, and Asad Abbas. Energy-leaks in android application development: Perspective and challenges. *Journal of Theoretical and Applied Information Technology*, 98:11, 11 2020.
- [18] Olivier Le Goaer and Julien Hertout. ecocode: a sonarqube plugin to remove energy smells from android projects. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, ASE '22*, New York, NY, USA, 2023. Association for Computing Machinery.

- [19] Yepang Liu, Chang Xu, and Shing-Chi Cheung. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering, ICSE 2014*, page 1013–1024, New York, NY, USA, 2014. Association for Computing Machinery.
- [20] Daniel Maia, Marco Couto, João Saraiva, and Rui Pereira. E-debitum: Managing software energy debt. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 170–177, 2020.
- [21] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Lightweight detection of android-specific code smells: The adocor project. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 487–491, 2017.
- [22] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105:43–55, 2019.
- [23] Gustavo Pinto and Fernando Castor. Energy efficiency: a new concern for application software developers. *Commun. ACM*, 60(12):68–75, November 2017.
- [24] Mark Rietveld. Empowering embedded systems developers to reduce the energy consumption of their software through visualizations: uncovering their information needs. Master’s thesis, Utrecht University, 2024.
- [25] Roberto Verdecchia, René Saez, Giuseppe Procaccianti, and Patricia Lago. Empirical evaluation of the energy impact of refactoring code smells. 02 2018.

Appendix A

Tools & Apps

Here are tools and apps used in this study, together with the source:

name	source
Android Lint	https://developer.android.com/studio/write/lint
aDoctor [21]	https://plugins.jetbrains.com/plugin/13443-adoctor
EcoCode [18]	https://github.com/green-code-initiative/ecoCode-android
Android Power Profiler	https://developer.android.com/studio/profile/power-profiler
PowDroid [5]	https://www.nouredine.org/research/powdroid

Table A.1: Sources of tools tested in this study

name	version	source	downloads
Thunderbird	Release 10.0	https://github.com/thunderbird/thunderbird-android	100K+
DuckDuckGo	Release 5.233.0	https://github.com/duckduckgo/Android	50M+
Freebloks 3D	Release 1.6.4	https://github.com/shlusiak/Freebloks-Android	500K+
Fossify Music Player	Release 1.1.0	https://github.com/FossifyOrg/Music-Player	10K+
Signal	Release 7.42.1	https://github.com/signalapp/Signal-Android	100M+

Table A.2: Sources of the apps tested in this study

Appendix B

Results

App	Android Lint	aDoctor
Duck Duck Go	123	9
Fossify Music Player	8	0
Freebloks	11	0
Signal	133	9
Thunderbird	79	575
Average	71	119

Table B.1: Table showing the runtime of the SA tools in seconds for each application they were tested on.

App	Finding	Solution
Thunderbird	70 cases of MIM	add <code>static</code> keyword
DuckDuckGo	4 cases of MIM	add <code>static</code> keyword
Signal	290 cases of MIM	add <code>static</code> keyword
Signal	1 case of IGS	replace <code>setAtrb(x)</code> with <code>this.atrb = x</code>
Signal	1 case of HMU	replace <code>HashMap<></code> with <code>SparceArray<></code>

Table B.2: The energy issues and suggested solution aDoctor found.

Appendix C

Hardware

Personal desktop

This device is used for all tests except for testing PowDroid.

CPU Intel Core i7 8700

RAM 2x16 GB DDR4 3200MHz

OS Windows 10

Storage Samsung 970 EVO Plus

GPU GigaByte GeForce RTX 3060 TI EAGLE OC

Acer Aspire 7 A715-75G-56GB

This device is only used for testing PowDroid.

CPU Intel Core i5 9300H

RAM 2x4 GB DDR4

OS MX Linux (Dual Boot)

Storage SSD

GPU Intel UHD Graphics 630