UNIVERSIDADE D
COIMBRA

Daniel Veiga Ramos Ramalho

TRY TO BEAT THE SYSTEM

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Bruno Cabral and Professor João Correia, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July of 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Daniel Veiga Ramos Ramalho

Try to Beat the System

Exploiting Mobile Apps Energy Certificates Vulnerabilities

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Bruno Cabral and Prof. João Correia, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2023

Acknowledgements

I would like to thank my friends for all the good times we have been through.

I would like to mention my advisors for all their help.

Finally, but most importantly, a special thanks to my family for supporting me in this journey. Without them, I couldn't make it.

Daniel Veiga Ramos Ramalho

Funding

This work was financed by FEDER (Fundo Europeu de Desenvolvimento Regional), from the European Union through CENTRO 2020 (Programa Operacional Regional do Centro), under project CENTRO-01-0247-FEDER-047256 – GreenStamp: Mobile Energy Services.

Abstract

The importance of mobile devices in our lives makes it hard to imagine our daily activities without them. Mobile applications (apps) influence the autonomy of the devices. However, there is no indication of app energy consumption in app stores. Energy certificates are the solution to app stores' lack of energy efficiency information. Energetic certification must be robust to guarantee the fair evaluation of each app. In our work, we aim to research energy certification mechanisms, detect their vulnerabilities, and present how they can be exploited. We analyzed different types of mechanisms to reach our goal and understand how they work. We approach the problem by conceiving adversarial strategies targeted to energetic certification that would make an app rank higher than it should. Within the scope of the dissertation, we implemented experimental environments defined in the strategies, and then we validated the vulnerabilities identified. The analysis and experimentation of this dissertation allow the indication of guidelines for the developers of energy certification mechanisms to grant a more robust evaluation method.

Keywords

Energetic certification for Mobile Applications (apps), reliable rating system, exploit rating system.

Resumo

A importância dos dispositivos móveis na nossa vida dificulta imaginar as nossas atividades diárias sem eles. As aplicações móveis (*apps*) influenciam a autonomia dos dispositivos. Contudo, não há indicação do consumo de energia das aplicações nas lojas de aplicações. Os certificados energéticos são a solução para a falta de informação sobre a eficiência energética nas lojas de aplicações. A certificação energética deve ser robusta para garantir uma avaliação justa de cada aplicação. No nosso trabalho, visamos pesquisar certificados energéticos, detetar as suas vulnerabilidades e, apresentar como estes podem ser manipulados. Para atingir o nosso objetivo e compreender como funcionam, analisamos diferentes tipos de certificados. Abordamos o problema concebendo estratégias adversariais dirigidas à certificação energética que fariam com que uma aplicação tivesse uma avaliação mais elevada do que deveria. No âmbito da dissertação, implementamos os ambientes experimentais definidos nas estratégias, e posteriormente validamos as vulnerabilidades identificadas. A análise e experimentação desta dissertação permitem indicar diretrizes para que os desenvolvedores de mecanismos de certificação energética possam conceber um método de avaliação mais robusto.

Palavras-Chave

Certificação Energética para Aplicações Móveis (*apps*), sistemas de avaliação fiáveis, abuso de sistemas de avaliação.

Contents

Acronyms

APK Android Package Kit

AUT App Under Test

CFG Control Flow Graph

CPU Central Processing Unit

DUT Device Under Test

E-APK Energy-aware Android Patterns for Kadabra

EE Energy Efficiency

IDE Integrated Development Environment

WCEC Worst-Case Energy Consumption

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The importance of mobile devices in our lives makes it hard to imagine our daily activities without them. The use of smartphones, tablet computers, and, more recently, wearables such as smartwatches have changed and simplified how we communicate, how we individually or collectively have fun, and how we work or do business. Ultimately, the number and scope of mobile applications (apps) appear unlimited. The distribution of such applications is highly facilitated by app stores such as Google Play Store[1] and Aptoide[2], which democratize the opportunity to commercialize software for mobile devices. Notably, the number of mobile applications downloaded has increased significantly from 140.7 billion in 2016 to 230 billion downloads in 2021. This indicates a growing trend of mobile app usage among consumers[3].

While the app market targets mobile devices, which by nature most often run on batteries, the fact is that current marketplaces do not provide any indication of the absolute or relative Energy Efficiency (EE) of the applications they host. According to studies, users place the battery life as the most crucial feature when choosing a new smartphone, emphasizing the need for mobile applications to be energy efficient[4]. Considering this, energy certification mechanisms for apps were developed [1, 2, 3, 4], well as energy evaluation tools [5, 6, 7]. The energy certification mechanisms give users information regarding the app's EE of the apps they are about to install, which may influence the autonomy of their devices.

Energy certificates for apps are like benchmarks that enable developers, providers, and users to compare apps concerning their power consumption behaviour. Benchmarks are often used to compare products' characteristics and performance. Hence, they must be robust; otherwise, vendors begin to develop their products to rank as high as possible with the benchmark artificially. Dieselgate[5] is a fine example of the abuse of evaluation mechanisms. Volkswagen implemented software to

[1] https://play.google.com/store/apps (Last Access: Jul 8, 2023)

[2] https://en.aptoide.com/ (Last Access: Jul 8, 2023)

[3] https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/ (Last Access: Jul 8, 2023)

[4] https://www.statista.com/chart/5995/the-most-wanted-smartphone-features/ (Last Access: Jul 8, 2023)

[5] https://www.bbc.com/news/business-34324772 (Last Access: Jul 8, 2023)

detect if the vehicle is under test by monitoring sensors such as speed and air pressure. Whenever the car is under test, the engine is adjusted to an operation mode that emits fewer pollutants, which allows the vehicle to meet the emissions requirements imposed by Environmental Protection Agency (EPA). However, Nitrogen oxides (NOX) emissions in regular driving were 15 to 35 times higher than allowed by EPA, Which led Volkswagen to an unfair advantage over other car manufacturers [8, 9].

With the implementation of mechanisms to analyze and profile the EE of apps in app stores, there are multiple benefits: developers are incentivized to improve their applications in terms of EE by increasing the concern about energy-efficient practices and users benefit from the capability to choose a more energy-friendly application. Energy certificates are already standard for household appliances (e.g., television and refrigerators), which motivated manufacturers to develop more energy-friendly appliances. By inserting energy certificates in the app market, it could also benefit from the increased concern for EE practices [5].

However, energy certification mechanisms must be reliable and ensure all apps have a fair rating. App providers could exploit the evaluation system for better evaluations if it has vulnerabilities. Then the rating slowly becomes less relevant since it may no longer represent reality. Developing innovative and robust mechanisms for evaluating the apps' EE is fundamental.

Moreover, project Greenstamp[6] is currently developing tools for analysis and profiling of the EE of the apps. The project Greenstamp aims to research and create innovative methods for evaluating and documenting the EE of mobile applications in marketplaces. Apart from detecting susceptibilities in the other certifications, we aim to explore also the evaluation methods developed by Greenstamp [5, 6, 7]. By reporting the limitations in the energy certification mechanisms for apps, new mechanisms can hopefully overcome them.

1.1 Objectives

We research energy certification mechanisms for apps and examine any potential susceptibilities they may have. These susceptibilities threaten the security of the evaluation system, which can lead to dishonest evaluations. Hence, we explore the vulnerabilities to determine how developers could design and develop their apps to rank higher than they should.

Our primary goal is to expose the flaws in the current app certification process so that creators of future energy certification may create a more robust version. To this end, we investigated and responded to the main research question:

RQ *How can developers create adversarial strategies against energy certification mechanisms that would make an app rank higher than it should?*

[6] https://greenstamp.caixamagica.pt/ (Last Access: Jul 8, 2023)

Our first objective is to research real examples of strategies influencing evaluations in a rating system. Following that, we aim to research energy certification mechanisms for applications and explore their problems and weakness. Then, the issues detected must be validated through experiments.

To reach our main goal and try to answer the RQ, we need to fulfil the following objectives:

- Survey of the state-of-the-art in software energy certification.
- Analysis of energy certification for mobile applications.
- Identify vulnerabilities in energy certification mechanisms.
- Design scenarios that simulate how app developers can exploit the vulnerabilities to obtain a better evaluation than they should in the energy certification mechanisms.
- Setup of the experimental environment.
- Implement and evaluate the proposed adversarial strategies designed in the scenarios.

1.2 Document Structure

In Chapter 2, we give background information about the methods for measuring or estimating the energy consumption of mobile applications. We overview diverse rating systems, their vulnerabilities, and how vendors exploit them to obtain unfair ratings on their products. In Chapter 3, we overview tools and certification mechanisms that allow an energetic app evaluation and identify susceptibilities in these methods. In Chapter 4, based on the vulnerabilities previously raised, we present scenarios to exploit them. In Chapter 5, we detail the experimental setup required for our experiments. In Chapter 6, we present the results obtained in the experiments, where they are also discussed. Finally, in Chapters 7 and 8, we summarize the work done in the dissertation.

Chapter 2

Background and State of the Art

This Chapter presents essential background knowledge to understand how energy certificates are attributed. To this end, we analyzed how an app's energy consumption can be measured to verify its efficiency. In Section 2.1, we distinguished two main approaches to measuring app energy consumption: energy monitors and energy profilers. In Section 2.2, we present four examples of vulnerabilities in different certifications and benchmarks, which allowed us to acquire a good foundation in the topic. These examples reflect vendors artificially developed their products to rank as high as possible in a particular rating system.

There is a lack of research on vulnerability analysis in app certification mechanisms. Therefore, we can gather knowledge by analyzing vulnerabilities in other certifications and benchmarks, which will be helpful in the definition of our approaches. Moreover, we verify the importance of creating robust rating systems. If the rating system is not robust, it becomes less relevant since it may no longer represent reality. We found clear examples of rating systems exploited by vendors.

There is no flawless rating system, and apps' energy certifications are no exception. Our work is to detect the vulnerabilities and create adversarial strategies to make an app rank higher than it should.

2.1 Energy Consumption Measurement

Tracking energy consumption is fundamental to ensuring green software [10]. A range of tools can be used to measure the power consumption of software. These energy consumption tools have different configurations as well different applications. Moreover, some implementations only work with some platform versions (e.g., Android 6.0), and no procedure fits every situation. Therefore, it is vital to use the technique that best fulfils our needs.

Generally, the analyzed articles use Power Monitors or Energy profilers to measure apps' consumption. Each approach has advantages, disadvantages, and limitations that must be considered when choosing an energy measurement tool. We

analyzed these methods in the following subsections.

2.1.1 Power Monitor

The device's power consumption can be measured using power monitors, which are hardware tools, to obtain the most accurate energy consumption results. The power monitor is connected to the device's power source, allowing the measurement of the actual device's power usage [10].

However, they can be adamant about setting it up because it implies dismounting the device and removing the battery. Another difficulty is syncing the power information with the software tested. The power monitor only collects information about power usage but does not have information about the software running on the device [10]. The developer must start and stop the power consumption recording of the power monitor to synchronize with the application activity [11]. We found some hardware-based power monitors, Monsoon Power Monitor [12], ODROID used in [13], and NEAT [14]. Finally, hardware-based power monitors were used in the certificate [15].

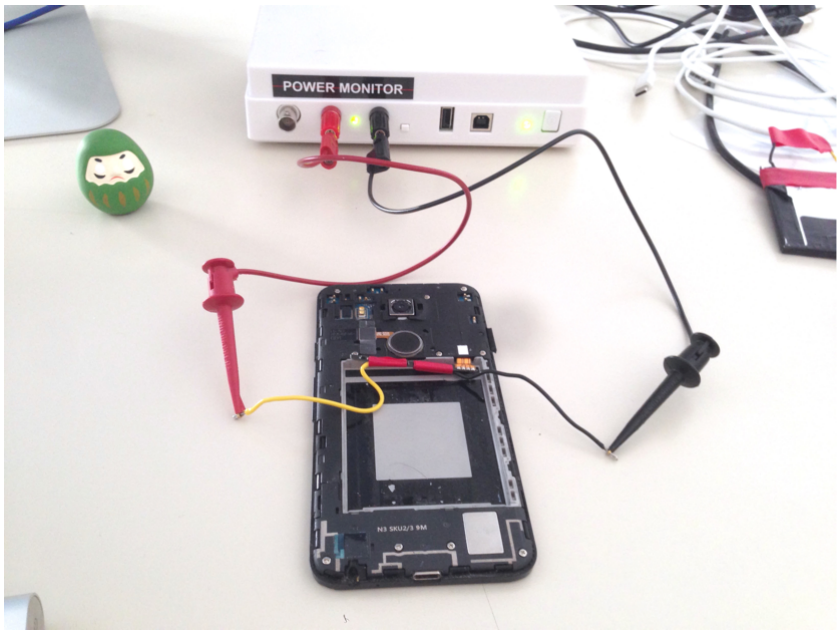


Figure 2.1: Example of a power monitor setup [10]

2.1.2 Energy Profiler

Energy profilers estimate the energy the application consumes instead of measuring the power usage like power monitors. They rely on power models adapted for each smartphone, and then based on which hardware components are active in the device, they estimate its energy consumption [10]. Energy profilers are more accessible to set up than Power meters because they do not need hardware modifications, additional hardware, or power sensors.

Energy profilers have limitations that can compromise energy profiling. They are mainly used for small sets, so they are not practical for profiling large applications [11]. Another limitation is that energy profilers are not reliable as power monitors. Besides, approximating the actual energy consumption does not ensure the measurement's reliability. Using PETrA, 95% of the analyzed methods' estimation error is within 5% of the actual values measured [16].

The certification mechanisms [1, 3, 2] used this approach, having these limitations. PowerTutor [2], Profiler[1] an Android Studio tool, Trepn Profiler[2] used in [17], GreenScaler [18], PETrA [16], Batterystats[3], GSam Battery Monitor[4], and Battery Guru[5], are energy profilers. PowerTutor, Trepn Profiler, Battery Guru, and GSam Battery Monitor are apps that must be installed on the device.

We explore Batterystats deeply because of its potential; the tool is included in the Android framework. Batterystats is easy to implement and presents the necessary information in a clear format. It collects all battery data in our development machine and creates a report, then using Battery Historian[3], we can analyze the report. Battery Historian converts the output from Batterystats into an HTML graphic that can be seen in the browser. In Figure 2.2, we present a piece of the report generated by Batterystats with the battery data collected from the device. In Figure 2.3, we can visualize a piece of the report converted by Battery Historian in the browser.

```
u0a321:
  Wake lock *launch* realtime
  Foreground activities: 1m 0s 558ms realtime (1 times)
  Top for: 1m 0s 582ms
  Cached for: 692ms
  Total running: 1m 1s 274ms
  Total cpu time: u=1m 2s 308ms s=335ms
  Total cpu time per freq: 35 1 0 0 137 2 239 74 124 118 35 39 30 0 2 833 0 0 0 0 0 0 0 0 0 0 0 0 0 0 61008
  Cpu times per freq at state Top: 35 1 0 0 137 2 239 74 124 103 35 31 30 0 2 197 0 0 0 0 0 0 0 0 0 0 0 0 0 0 60984
  Cpu times per freq at state Cached: 0 0 0 0 0 0 0 0 0 15 0 8 0 0 0 636 0 0 0 0 0 0 0 0 0 0 0 0 0 0 24
  Proc com.example.challenge1_A2:
    CPU: 0ms usr + 0ms krn ; 1m 1s 960ms fg
    1 starts

Estimated power use (mAh):
  Capacity: 2851, Typical: 5000, Computed drain: 0, actual drain: 0
  Global
    screen: 10.0 apps: 10.0 duration: 2m 22s 478ms
    cpu: 3.10 apps: 3.10
    system_services: 0.0896 apps: 0.0896
    mobile_radio: 0.0580 apps: 0
    sensors: 0.0891 apps: 0.0891
    wifi: 0.00167 apps: 0.00167
    idle: 0.170 apps: 0 duration: 2m 22s 478ms
  UID u0a321: 1.90 fg: 1.88 cached: 0.0180 ( cpu=1.90 (1m 2s 643ms) cpu:fg=1.88 cpu:cached=0.0180 system_services=0.00337 )
  UID u0a305: 0.468 bg: 0.0382 cached: 0.392 ( cpu=0.431 (26s 562ms) cpu:bg=0.0382 cpu:cached=0.392 system_services=0.0379 )
  UID 1000: 0.250 ( cpu=0.249 (20s 463ms) sensors=0.0891 (10m 41s 151ms) wifi=0.00166 reattributed=-0.08962500 )
  UID 0: 0.246 ( cpu=0.246 (19s 515ms) )
  UID u0a66: 0.0682 fg: 0.0573 ( cpu=0.0573 (2s 791ms) cpu:fg=0.0573 system_services=0.0109 )
  UID u0a82: 0.0463 fg: 0.0402 ( cpu=0.0405 (2s 667ms) cpu:fg=0.0402 system_services=0.00581 sensors=0 (1m 11s 239ms) )
  UID u0a320: 0.0404 fg: 0.0339 cached: 0.00345 ( cpu=0.0373 (1s 855ms) cpu:fg=0.0339 cpu:cached=0.00345 system_services=0.00304 )
```

Figure 2.2: Example of measurements collected using Batterystats

[1] https://developer.android.com/studio/profile/android-profiler (Last Access: Jul 8, 2023)

[2] https://developer.qualcomm.com/blog/introducing-trepn-profiler-60/ (Last Access: Jul 8, 2023)

[3] https://developer.android.com/topic/performance/power/setup-battery-historian (Last Access: Jul 8, 2023)

[4] https://play.google.com/store/apps/details?id=com.gsamlabs.bbm&hl=en_US&gl=US (Last Access: Jul 8, 2023)

[5] https://play.google.com/store/apps/details?id=com.paget96.batteryguru&hl=en_US&gl=US (Last Access: Jul 8, 2023)

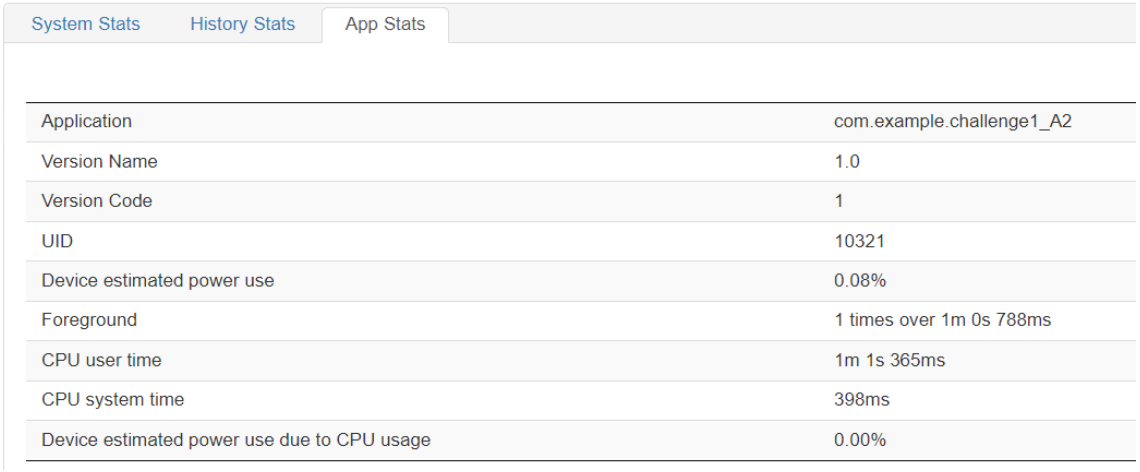


Figure 2.3: Example of the app stats visualization. Data was collected using Batterystats and visualized using the Battery Historian.

2.2 Cases of Rating Systems Vulnerabilities

Some organizations have tricked the evaluation system into obtaining a better evaluation of their products than they should. The rankings have vulnerabilities, and manufacturers put effort into abusing them. A ranking can show the user how the product will perform in certain circumstances, but sometimes it is a relative evaluation. If the user does not know how the rating is attributed, he can not make a thoughtful comparison. For example, the concept of Energy Efficiency (EE) can change between regions. In China, the EE of a television relies on the television's power consumption and the television's brightness. European Union (EU) opted for calculating the EE relating the power of the product to its size. Manufacturers try to make the product as efficient as possible in the market, leading to high-brightness televisions in China and TVs with larger screen sizes in Europe [19].

2.2.1 Dieselgate Case

The Dieselgate case was a worldwide fraud known thanks to his scandal's proportions. The German car manufacturer Volkswagen manipulated the emission test on their engines to obtain a lower emission value than it should have. With such low emissions, the motors were labelled as "Clean Diesel" motors putting the brand in an excellent position in the car market [20]. However, in reality, the emissions in regular driving of Nitrogen oxides (NOX) were 15 to 35 times higher than the Environmental Protection Agency (EPA) demanded [8, 9]. Only eight years later, this fraud was discovered, but the company sold 11 million defective cars in that period [8]. Moreover, the excess pollution provoked multiple health problems, most related to breathing, and also caused deaths [9].

It is important to know under what conditions the emission test is conducted. The test is done with the car in a dynamometer that fixes the car in place while

allowing the tires to rotate, and then an emission sensor is placed in the vehicle's exhaust. The sensor collects emissions information, which allows for checking whether the emissions are within limits. During the test, a car is made to adhere to a precise speed profile. The test requirements, including the speed profile, are standardized and available to the public [21].

The Engine Control Unit (ECU) regulates engine operation by constantly analyzing sensor data and adjusting engine aspects. Therefore, the engine manufacturer can program the vehicles' computers which decide how the car will perform [20]. They programmed it with two operation modes: the testing mode and the regular driving mode [9]. Since the test conditions are public, the company built software that predicts when the car is being tested, and if it is the case, the ECU activates the testing mode. The software uses various variables like the speed of the vehicle, the steering wheel, and traction control sensors to decide whether the car is being tested. This mode reduces the performance and increases the vehicle's consumption but lowers the emissions to an acceptable level by EPA [21]. Concluding, Volkswagen had passed the emissions benchmark test, although the values were much higher than allowed.

2.2.2 User Ratings and Reviews

Many applications in app stores have boosted their ranking to higher positions, like the top 50 trending applications. The ranking position in an app store can be induced by multiple factors and variate between different app stores. However, tendentially, the most critical aspects are the number of downloads and the ratings given [22]. So, collusive groups were created that perform attacks on the ranking of the app store, impacting the evaluation of one or multiple applications. These groups are formed by numerous people that give high ratings and good reviews in an application defined by the group leader. [22, 23]. Then when the app store ranking of the apps is updated, the application hit by the attack reaches the top positions until at least the following ranking update [22]. The app stores have improved at controlling these practices, but apps still have inflated ratings.

The reputation ranking of products in online stores also is susceptible to manipulation when some users give an unfair rating to a product. Reputation is the score of a product obtained through collective intelligence, i.e., the result of collaboration between many individuals [24]. The reputation of a product influences the customer's choice because it indicates the opinion of other customers. Like in app stores, companies can hire groups of people to rate their product with an excellent grade to increase their reputation. However, this type of evaluation given by the groups is dishonest and reduces the trustworthiness of the reputation.

2.2.3 Journal Rank Indicator

Multiple articles are published every year in science journals, and the quality of each article can be measured by the impact of the journal in which it was published [25]. A journal's citations are usually used in metrics to rank and compare

the journal's impact with others. There are multiple metrics used to perform the importance evaluation, like the Impact Factor (IF), Eigenfactor Score (ES), and SCImago Journal rank indicator (SJR). These indicators have the purpose of ranking the importance of the journals by using different metrics [26]. To calculate the IF of a journal in a given year, we divide the number of citations in that year by the number of citable papers published two years before the stated date. This metric leaves room for vulnerabilities because it does not consider other journal characteristics and citations [25]. The other two metrics are very similar, only variating in the data size used: SJR used much more information than ES. Both metrics consider the importance of the journal that made the citation; hence, the evaluations using these metrics increase when more prestigious journals cite the journal [26].

The IF metric has some vulnerabilities, and publishers and editors found strategies to boost the journal evaluation. One of the strategies used is publishing review articles that are usually more cited than original articles. Journals publishing only review articles can have a very high IF [25]. Another strategy is self-citation which consists in citing articles from the same journal where the article is published. Self-citing is more frequent in non-English journals because local communities tend to publish in their language in local journals [26]. Finally, journals also publish non-citable items such as commentary-type articles and correspondence articles that do not count toward the metric denominator but toward the nominator, increasing the IF of the journal [26].

2.2.4 Energy Labels for Household Appliances

Household appliances, like televisions and freezers, are responsible for much of the energy consumption in a house. The primary electric appliances in a household account for 25.6% of the total energy consumption of Spanish households [27]. Moreover, with the energy demand increasing in European Union (EU), there is a need to acquire and encourage the acquisition of more energy-efficient electric appliances [28]. Electronic appliances' energy labels help the customer choose the most energy-efficient alternative. These labels contain information about the product depending on the product's category; however, EE and the product's energy consumption are expected on all labels in the EU. The energy label information lets the customer make better energy-efficient decisions benefiting from energy savings. Besides that, the environment is protected because energy production, which causes much pollution, can be lowered. Due to EE requirements and more thoughtful customer choices, manufacturers are encouraged to manufacture more environmentally friendly appliances [29]. When several products reach the best EE classification, the energy labels are updated to leave room for improvement. If the EU did not update energy labels, there was no improvement margin, which may generate confusion for the customers by having such different products with the maximum EE rating.

The electric appliances to receive an energy label are submitted to a series of performance tests. The tests are carried out under similar conditions for the same product's category to ensure reliability, with a few exceptions. EE represents the

energy used concerning the size of the product. For example, the EE of televisions is calculated by the power consumption (Watts) per squared decimeter (dm^2) of the screen [28, 30]. So the EE of televisions depends on their size, and televisions with different screen sizes can have the same EE. In the previous labels, replaced in March 2021, the EE rating is attributed on a scale from D to A+++, with A+++ attributed to the most efficient products and D to the least [27]. Besides energy efficiency, we have annual energy consumption information, which is calculated based on the consumption per year of the electrical product, assuming a product's generic use. A television, for example, is considered to be used 4 hours per day for a year, but a freezer is supposed to be plugged in all year [28].

The EE of a product does not mean that the product will use low energy to perform. It just means that it uses low power in relation to the product size. Choosing an electric appliance with lower energy consumption can confuse the customer. The customer's perception of energy labels and their influence on the buying choice was investigated in [28]. The conclusion was that the energy label needs to be clarified, or the customers need more knowledge to interpret them. The customers do not give enough importance to the product's annual consumption, focusing only on EE, which is easier to understand and more appealing [28, 30]. This way, customers can be misled into thinking that two electric appliances have similar energy consumption because they have an equal EE rating; this incentivizes manufacturers to develop increasing-size products [31]. On the other hand, TV energy labels in China calculate EE based on power usage to produce a specific luminance. Therefore, TVs with higher luminance benefit from the Chinese labels [19].

2.3 Summary

We learned how an app's energy consumption could be measured and analyzed. Then, distinguished two different measurement methods: power monitors and energy profilers. Power monitors are hardware tools that allow a reliable measurement of the device's power usage but are complicated to implement. Energy profilers are software tools that are more accessible to implement but, due to the energy consumption estimates they use, are not as reliable as power monitors.

We gathered a good foundation of vulnerabilities in different certifications and benchmarks. Each certification mechanism has vulnerabilities that can be exploited. For example, when exploring energy certificates for household appliances, we acknowledge how EE is calculated based on the product's category, specs, and energy consumption. We comprehend the difference between EE and energy consumption; good EE doesn't always mean low energy consumption.

Concluding, vendors can design and develop their products to rank as high as possible with the benchmarks artificially. In the same way, app vendors could find gaps in the energy certifications that allow their apps to rank higher than they should. Therefore, we fulfil our first objective: *Survey of the state-of-the-art in software energy certification*, defined in Section 1.1.

Chapter 3

Energy Certification for Apps

This Chapter covers the energy certifications for apps and how developers can create adversarial strategies to make apps rank higher than they should.

Section 3.1 walks through the process of creating an energy certification mechanism for apps and highlights points that should be considered when developing a robust certification. The energy certifications must be reliable, enabling developers, providers, and users to compare apps concerning their power consumption behavior. Otherwise, developers can explore their products to (artificially) rank as high as possible.

Section 3.2 identifies vulnerabilities in energy certification mechanisms for apps and evaluation tools. To this end, it provides an overview of the techniques. The certification mechanisms analyzed were: [1, 2, 3, 4]. Moreover, we analyzed three tools that allow an energetic analysis of the apps [5, 6, 7].

3.1 Understanding Energy Certification Mechanism for Apps

Energy Efficiency (EE) could be decisive when discussing apps with similar goals and functionalities. Energy certificates for applications will allow users to compare similar apps' EE, which could help them choose the app that suits them better. It may also incentivize vendors to improve their app's EE, which leads to research and development of power-saving mechanisms [32].

There needs to be more information about apps' energy consumption or efficiency in app stores. Users can only rely on the app vendor's information, the number of downloads, and the user rating and reviews when choosing an app. Energy certificates function like benchmarks that can be used to evaluate the efficiency performance of an app. The objective of benchmarks is to provide a standardized way to measure the performance of a system or component so that it can be compared to others, which allows users to make informed decisions about which apps are best suited for their needs based on performance characteristics.

Benchmarks must be reliable, so they must be carefully designed and validated. Furthermore, desirable benchmark characteristics depend on the goals of the benchmark and the system being tested. The definition of desirable characteristics will be helpful in the analysis of benchmark limitations. This Section defines desirable characteristics and the process steps in creating a certification mechanism.

We can divide the evaluation methods into two categories:

- **Dynamic Analysis:** Dynamic analysis requires program execution on the target hardware or emulator, which enables us to obtain consumption values similar to the app's actual consumption. To this end, the testing team must create Test Cases that simulate the app's usage. The test case design is demanding because the developer must understand how the app must work. Furthermore, it has the limitation that only one use case can be tested each time [33], and not all possible input combinations can be assessed. Finally, the app is repeatedly tested to validate the collected consumption data. The evaluation methods [2, 3, 4, 7] are based on this type of analysis.
- **Static Analysis:** Using static analysis, we do not need to execute the app as an alternative to dynamic strategies. Instead, each part of the program is analyzed, producing an approximation of the entire program [34]. Static approaches can be easily applied in app stores and automated for every app, enhancing scalability. The static analysis is utilized in the evaluation methods [1, 5, 6].

3.1.1 Desirable Characteristics

In this Subsection, we will identify some general desirable characteristics that the energy certification mechanisms for apps should satisfy. These characteristics will be used to help identify limitations. The limitations can prevent the energy certification mechanisms from being applied in an app store.

- **Reproducibility:** Reproducibility is necessary to make meaningful comparisons between systems or components. The benchmark should produce consistent results when run under the same conditions. When the same app is run multiple times under the same conditions, the reported energy consumption values must be virtually the same with only minimal deviations allowed [32].
- **Fairness:** The benchmark should be fair, meaning it should not favour any particular system or component unfairly. For example, a benchmark that is biased toward a specific app may make it appear that the app is superior to others, even if they are not. Moreover, measurement devices must provide specific properties which guarantee comparable energy values [32].
- **Transparency:** The benchmark should be transparent, meaning the methodology and data used should be documented and available for review. The

benchmark needs to cover all possibilities and not allow any room for exploits.

- **Relevance:** The benchmark should accurately reflect the performance of the system or component being tested. It should be designed to represent the workloads and conditions likely to happen in real-world use [35]. If the tests performed on the app do not represent the actual use of the app, they have no relevance to users.

3.1.2 Process Steps Behind the Development

Designing energy certification mechanisms to allow comparisons between applications and finding a way to present energy-aware benchmarks in the marketplace is challenging [4]. The mechanisms must meet the previous subsection's desirable characteristics. Otherwise, app stores may prevent them from being accepted and implemented. And users may not trust the certificates or be misled by them.

After analyzing some suggested app certification mechanisms, we defined general critical points in their designing process. The critical points will be explored in the following section to identify limitations and vulnerabilities in the techniques used by the certification mechanisms.

- **Define the energy measurement method:** The energy measurement method involves choosing a tool to measure an app's energy consumption. We can distinguish between two different measurement methods, the usage of power monitors or energy profilers. Each technique has advantages, disadvantages, and limitations [10]. We have discussed this subject and identified which measurement method uses each energy certificate in Section 2.1.

- **Define test cases:** "To measure the energy consumption of a mobile application, one needs to design the use case in which the energy consumption will be tested" [11]. The use cases must be similar inside each app category to allow the comparison between apps inside the same category.

  The app categorization may be hard to define, but it is necessary to establish category-specific tests. The apps inside a category must be tested similarly to ensure benchmark fairness. However, some certification mechanisms did not categorize the apps, which allows comparing all apps between them.

- **Manage critical consumption details:** Energy labels for applications should address how applications drain devices' batteries, and all specific cases should be considered to ensure the robustness of the benchmark. The background service consumption, in some applications, impacts the total app consumption. Moreover, some applications need helper apps to provide a particular service. Energy labels must specify how to deal with helper apps. We will explore these two points in the following Section.

- **Design energy labels based on applications' energy consumption:** This step involves conceiving a transparent way to inform the user about a product's EE. The energy label must compare the various apps based on consumption measurements. The user must be able to search for energy-aware mobile applications quickly and comprehensively [4]. For example, the EE in the energy labels for household appliances is assigned from A to G, with A being one of the class's most efficient and G being the most inefficient [29].

The two first points presented, define the energy measurement method and define test cases, are unrelated to the static analysis. They are linked to the dynamic analysis, which implies running the app on the target device. There are various valuable uses for static analysis. Finding the routes that lead to undesirable behaviour is achievable (bug-searching). Additionally, static analysis techniques are frequently used in Integrated Development Environment (IDE) to help uncover code errors and on compilers to do automated optimizations. It also makes it possible to identify several errors in the source code, some of which are impossible to find using tests. For instance, uninitialized variables, invalid operations (such as division by zero and overflow/underflow in arithmetic expressions), resource leaks, and buffer overflow are some of the issues that may be detected by static analysis. Inside the static analysis, we emphasize two strategies: refactoring and path-tracing.

- **Refactoring Strategy:** Refactoring software involves altering the code's structure without changing its behaviour, which software maintainers can use to reorganize a program [36]. Software quality decays when developers make poor design choices or add new functionalities, which may lead to the origin of anti-patterns. Developers should avoid anti-patterns that cause battery drain, as documented [37]. Moreover, developers can manually refactor the app or use some refactoring tool to assist with the process. Refactoring must be done carefully to ensure the app continues working as expected on different devices and operating systems.

- **Path Analysis Strategy:** This strategy uses the app code or APK to trace all possible execution paths. Then, using all the data, the tools estimate the consumption of the app or give some evaluation. The tool may access the device consumption module, allowing a more accurate consumption estimation.

After analyzing various energy evaluation methods [1, 2, 3, 4, 5, 6, 7], we detect limitations and vulnerabilities in each process step. In the following section, we will explore each process step profoundly and focus on the vulnerabilities developers could exploit to obtain better evaluations in energy certificates.

3.2 Vulnerabilities and Limitations in the Certification Mechanisms

In this Section, we present vulnerabilities and limitations in the process steps of the design of a reliable energy certification mechanism. Moreover, we explore two approaches based on static analysis: refactoring and path-tracing strategies.

3.2.1 Design of Test Cases

Use cases are necessary to simulate the actual use of an application when we are using dynamic analysis. Then apps' energy consumption can be measured for each use case, exploring at least one application functionality [11]. The creation of use cases is a crucial step to allow the comparison between apps' energy consumption. They must be designed to ensure the reproducibility of results, i.e., the results must be similar in each run. If reproducibility is not guaranteed, we need more tests for each use case to reduce the measurement uncertainty. The human interaction also causes the measurement uncertainty to increase, with delays between interactions. So automating the test cases is more suitable to measure energy consumption [11].

We can detect two approaches relating to the articles that propose app energy labels. Some articles define the use cases oriented to app categories, allowing comparing apps inside the same category. In contrast, others design abstract test cases that are only time-limited without a precise definition of the interaction with the app. We will also make the distinction between automatized and human-made tests.

Application-Oriented Use Cases

Nowadays, apps are very complex and have plenty of functionalities, increasing the complexity of designing use cases. The developer must know the app's functionalities and objectives to define the use cases. Furthermore, app vendors need to describe app functionalities clearly because developers can only rely on the information they give.

Categorizing each app based on its functionalities and the main objective is necessary. So, we must create general use cases to be executed on apps that provide similar services. The general use cases allow energy consumption comparison. For example, Wilke et al. [4] made an investigation using two e-mail client apps. To test the energy consumption of these apps, they designed use cases that can be applied to both apps: (1) browsing an e-mail account's inbox and checking for new e-mails and (2) reading an e-mail. Therefore, they can compare the functionalities consumption of similar apps.

App Categorization is a complex process that depends on the developer's perception of the information given by the app vendor. The developer that designs

the use cases must understand well the app operation. If he fully understands the app's operation, he may create a correct path for addressing the use case. Otherwise, the path designed can be inefficient, which is unfair to the evaluated app, i.e., be more extensive or time-consuming to reach the same objective, which is unfair to the evaluated app.

The article [3] used this approach and avoided the difficulties by choosing a set of 6 apps from the Dictionary category. They selected this category because it has simple and constrained use case scenarios that can be investigated and carried out in a fair amount of time, enabling them to produce precise estimations. They designed use cases such as looking up, adding, and deleting words in a dictionary. One limitation referred to by the author is the recognition of typical use case scenarios in apps from other categories. Hence, developers must have specific subject expertise.

Sophisticated apps can cause a more elaborate design process and more complex use cases. Therefore, it is suggested by Wilke et al. [4] that app vendors must submit energy consumption measurement tests alongside the mobile application.

Vulnerability 1: App vendors submit energy consumption measurement tests along with the app.

We detect vulnerability 1 in the energy certificate proposed by Wilke et al. [4]. This alternative suggests that app vendors should deploy the test cases with the app. It uses more precise tests because app vendors know their applications best. Besides, developers spend less time and resources on the use case design and configuration. However, app vendors could exploit the tests for better energy evaluation.
We have verified multiple times that vendors can exploit vulnerabilities in benchmarks to obtain better performance ratings. The Dieselgate case analyzed in Chapter 2 is one of many examples of app vendors gaming the benchmarks. In conclusion, to apply this approach, we need to trust app vendors, which can be unreliable.

Time-Limited Tests

Designing an app's use case can be challenging for the developer. Sometimes there is no clear information about the app's functionalities and objectives. In this situation, the developer can not create a specific use case for testing the app. For example, games can have multiple modes and functionalities that can compromise app categorization. Hence, the solution is to test the app with human intervention for a specific time. The apps could then be compared, considering the energy consumption and the time used. This approach has been used in [1] and other articles that performed energy measurement tests in apps.

This option does not guarantee the reproducibility of the results; the human interaction is not precise, and delays can arise between each interaction. Therefore, each test can have different outcomes, which originates uncertainty about the re-

sults. And, without test automation, implementing the energy certificate in an app store is demanding, so it has scalability problems.

Test Automation

The automation of tests allows reliability and reproducibility of results. There are delays between User Interface (UI) interactions in a test manually conducted [11]. Besides, the number of tests can be higher because it does not implicate human interaction with the device.

To automate the tests, developers must master the application workflow, and the app can have multiple flow alternatives to achieve the same goal. Furthermore, it is a callous process, and the testing team must know how the app operates. To automatize tests, developers create scripts that interact with the app to mimic human interaction (e.g., Click Button, Insert text). The scripts that allow the user replication interaction are manually crafted, and each script must ensure the correct operation of the app [11]. The articles employed automatized tests [3, 4].

3.2.2 Background Service

As we already know, when we use an app, it consumes energy in its operation. The Central Processing Unit (CPU) and other hardware components, such as GPS, allow the user to operate the app. Despite not using an app, the app could consume a lot of energy because of the background service. In some cases, the app uses energy 24/7 while not in use [15]. Therefore, energy certificates must consider this type of consumption in their measurements. If not, the energy certificate can mislead the user by omitting information that sometimes can be the most impactful in an app's consumption.

Most energy certifications proposed neglected background service consumption. Only one referred to and measured this consumption [15]. In that certification, we can verify that background service consumption can not be ignored and can significantly impact overall consumption. The relation between background service energy consumption between two similar apps has 75%, a meaningful difference that impacts the app consumption. If the user only has the consumption information of the app when it is in the foreground, he cannot make a pondered choice.

Vulnerability 2: The background service consumption is not regarded in the energy certification mechanism.

Some apps have high background consumption, and these benefit when this type of consumption is not taken into account in attributing energy certificates, which is unfair to the competition. Therefore, an app can consume massive energy in the background and still have a good energy efficiency rating. We better analyze this vulnerability and how it affects the energy certificate robustness in Strategy 1.

3.2.3 Helper Apps Consumption

Android applications can delegate services to other applications via so-called Intents. For example, in a mail client app, another application opens the attachment, depending on the file type (e.g., an image or PDF viewer). Consequently, the app that accesses the attachments determines how much energy this use scenario consumes [4].

We identified a lack of transparency in energy certificates on how the use case consumption should be measured when the App Under Test (AUT) needs to call a helper app to complete the use case. The energy certificate description must mention how to handle the situation where the AUT delegates services to other apps. We can distinguish two gross approaches for this situation: ignore the helper app energy consumption and include the helper app consumption in the consumption of the AUT. In this Subsection, we analyzed the two approaches.

Ignore the Helper App Consumption:

This alternative considers only the AUT energy consumption in the energy certificate. The consumption information of any app besides the AUT is discarded or not measured because that information will not be used in the energy rating attribution. This approach has been applied in [4].

However, the energy label will not reflect the actual consumption of the use cases tested because it is only considered a percentage of the total consumption of the use case. Apps that use more helper apps benefit from this consumption measurement method. Therefore, app vendors can start delegating their app's services to helper applications. These services' consumption will no longer count for AUT consumption, artificially decreasing the app's consumption. For example, developers can transfer some app services to the browser, and the app can call the browser to perform the services.

Vulnerability 3: The helper app consumption is not regarded in the energy certification mechanism.

Apps can delegate services to helper apps to complete a specific activity; these apps benefit when the energy certificate does not consider the helper app consumption. We better analyze this vulnerability and how it affects the energy certificate robustness in Strategy 2.

Use All the Consumption Information, Including the Consumption of the Helper Apps:

When an app needs to call a helper app to perform a particular service, the user can choose which app will conduct the service. This approach does not discard the consumption information from the helper apps; the test case consumption

considers all helper apps involved. Moreover, the helper app can significantly impact the total energy consumption of the test case.

It may not be possible to test with the default helper app because some devices' brands have exclusive apps. For example, Samsung Galaxy devices running Android 7.0 (Nougat) and higher use Samsung Notes[1] as the default PDF reader. However, devices that do not meet these requirements can not install this app but may also have a default PDF reader. The devices' default helper app may have different consumption. Hence, testing with the default device application may not give the user of another device brand reliable consumption information; the user cannot ensure the conditions where the app has been tested.

Another alternative is to choose a helper app that can be installed on all devices. The energy certification mechanism should define default apps to perform specific tasks; this way, users can better understand AUT consumption. Even though they may not use the helper app used in the tests, they can install and use it to obtain similar results. However, this approach can be unfair to the helper apps not chosen as default for testing by the energy certificate.

3.2.4 Refactoring Approaches

In this Subsection, we present a certification mechanism Almasri et al. [1] and a refactoring tool Gregório N. et al. [5] based on the refactoring approaches.

Almasri et al. [1] propose an EE ranking based on application refactoring. They consider an app's optimization level a good indicator of EE.

To optimize the app, are applied refactoring tools that allow us to track high-consumption anti-patterns. They used the refactoring tools: EARMO proposed by [38] and Leafactor [4] to detect high-consumption anti-patterns in the app. The anti-patterns detected must be replaced with more efficient alternatives, so the refactoring aims to improve the app's efficiency. The correction of anti-patterns detected is not automatic, so it is up to the developer to determine which refactoring possibilities can be applied; this can be a problem because some refactorings suggested by the tools do not go according to the app developer's opinion [37]. After the refactoring, in theory, the optimized app consumes less energy.

The app's energy consumption measurement tests consist of 30 minutes of continuous exhaustive usage of the app. It is not clear how the functionalities of each app are approached, which can lead to a reproducibility problem. They measure the app consumption using the energy profiler PowerTutor tool to collect all the consumption data. This tool, unfortunately, is no longer available in the recent Android versions.

The EE calculation presented in Figure 3.1 relates the actual energy consumption and the refactored one. They used a data set of 20 apps for the tests, and the results obtained after refactoring are auspicious. The energy consumption

[1] https://www.samsung.com/us/support/owners/app/samsung-notes (Last Access: Jul 8, 2023)

improved in almost all the tests, proving that the original applications were not optimized enough. The formula used to obtain the DR (Difference Rate), considered in the energy-efficiency rating, is $DR = \frac{E1-E2}{E1}$, E1 is the app's energy consumption before refactoring, and E2 is the energy consumption of the app after performing the refactoring. This formula relates the initial energy consumption with the post-refactoring, showing how much the app improved with the optimization. If the value is close to 0%, it means that the app was already remarkably optimized, and the refactoring tools did not detect many power-hungry patterns in the app. Therefore if the DR is more significant than 0%, it proves that the app has optimization problems detected by the refactoring tools. Based on the DR evaluation obtained, the app gets a Star rating associated; for example, with a DR of 5%, the app receives four stars in the ranking, but with a DR of 26%, the app only gets three stars. Hence, already optimized apps obtain the top positions in the energy-efficiency ranking.

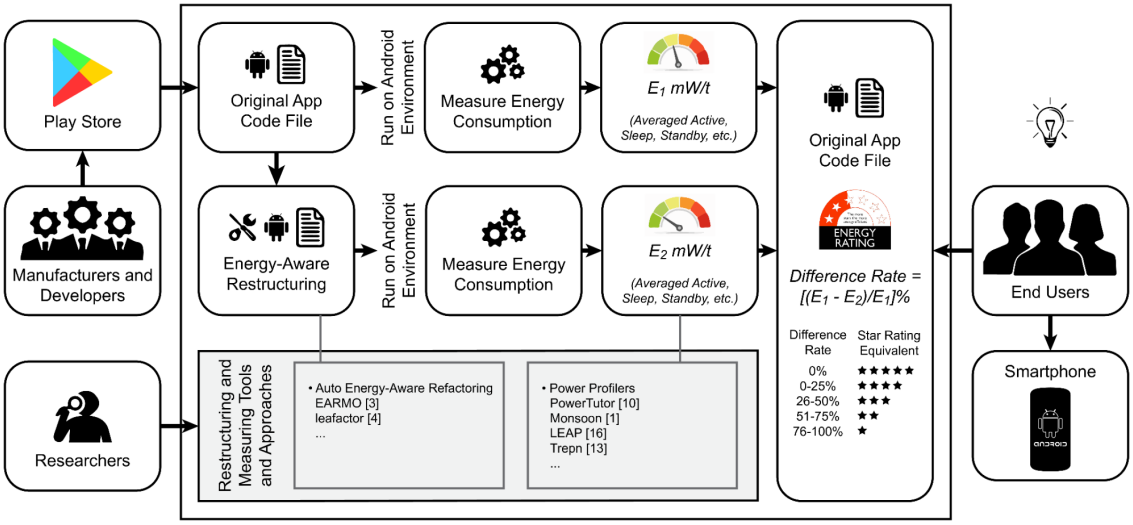


Figure 3.1: Energy certificate for apps proposed by Almasri et al. [1]

Moreover, Gregório N. et al. [5] developed a library of detectors that improves the detection process, Energy-aware Android Patterns for Kadabra (E-APK), extended Kadabra. It can detect refactoring opportunities that can improve the app's efficiency. In addition, the tool can work directly with the Android Package Kit (APK) in case we cannot access the app's source code. It is a valuable attribute because most app stores do not provide the app's source code.

App Refactoring

Refactoring software involves altering the code's structure without changing its behavior, which software maintainers can use to reorganize a program [36]. Software quality decays when developers make poor design choices or add new functionalities, which may lead to the origin of anti-patterns. Developers should avoid anti-patterns that cause battery drain, as documented [37]. Developers can manually refactor the app or use some refactoring tool to assist with the process; refactoring must be done carefully to ensure that the app continues to work as expected on different devices and operating systems.

The refactoring tools detect anti-patterns, so developers must decide which refactoring operations can be applied to the identified locations. This step is challenging because each anti-pattern can impact the software differently, changing the software design. Hence, the refactoring tools' suggestions should not be applied in some cases.

Refactoring Tools

Tools can support developers in the refactoring process. Some tools can automatically refactor applications after the developer agrees with the suggestions. Tools that can directly analyze source code, such as IDE plugins, are desirable; however, most app repositories do not provide the source code. Therefore, in these cases, we must use tools that do not rely on the source code and can detect energy-greedy patterns based on the app APK [3].

Some Android Studio plugins are EcoDroid [39], aDoctor [38], and Randroid [40] detect energy-greedy patterns and can automatically refactor them. For Eclipse IDE, we found Leafactor [41], which refactors the source code to follow a set of patterns known to be energy efficient. The EARMO tool proposed by [37] can detect energy-greedy patterns in the app code. It does not need the app's source code; it directly analyzes the APK. The tool suggests refactoring opportunities that developers should evaluate. EARMO tool found that 68% of the refactorings suggested were accepted and proved that the tool reduces the amount of energy consumption.

Vulnerabilities and Limitations

In the following, we identify limitations and vulnerabilities in the certification mechanism that uses refactoring tools. It is a heavy and time-consuming task for the developers. Developers must define which code should be refactored because some refactorings can impact the software design. In the documentation of the EARMO tool, we can verify that the app developers accepted just 68% of the refactoring recommendations. This way, apps will be tested with refactorings that should not have been applied [37].

More problems are related to exaggerated confidence in refactoring tools. They conclude that the app is fully optimized if no energy-greedy anti-patterns are detected. It is not a trustable approach because there can be an inefficient app with no anti-patterns detected by the tools. Hence, this rating system can evaluate an app with the best evaluation despite the app's high consumption. This approach can only be viable if the anti-patterns detected by the tools are the only energetic problems in an app. The refactoring is limited and must be used by the developers to improve their applications. However, in this rating, apps that do not have any anti-patterns detectable are favored despite having a high consumption.

Vulnerability 4: A refactored app is considered energy efficient.

Refactoring tools can improve the app's performance and efficiency by detecting refactoring opportunities. Still, they cannot ensure that all detections correspond to efficiency problems. Neither that all app issues were detected. So, the tools may give an incorrect evaluation. We better analyze this vulnerability and how it affects the energy certificate robustness in Strategy 3.

3.2.5 Path Analysis Approach

It is based on Static Analysis, which allows us to retrieve valuable information by analyzing the app's source code. In this Subsection, we present a tool based on this approach, wcec-android, developed by Kelson D. et al., [6].

Wcec-android estimates an app's consumption without needing a device to run the app, which allows analyzing an app quickly and easily without having programming knowledge or any app details. The energy consumption is estimated based on a consumption model of a Device Under Test (DUT). The consumption model contains the hardware components consumption, such as Wi-Fi and GPS. Each device has its consumption model, which can be changed in the tool, allowing testing of multiple devices.

Wcec-android traces all possible paths and builds a Control Flow Graph (CFG) representing each possible path. The CFG contains all possible execution paths of the program fundamental to performing the consumption analysis. Then, is calculated the energetic cost of each path. Finally, to find the energy upper bound in the CFG are applied equations that calculate the Worst-Case Energy Consumption (WCEC).

Vulnerabilities and Limitations

As we can notice, the WCEC represents a linear flow in the graph, so two paths cannot be followed simultaneously. Apps may have threads, which allow multiple tasks to be performed concurrently. However, the tool can only follow one of the paths. Therefore, the consumption of the paths not followed is not regarded in the final evaluation.

Vulnerability 5: The evaluation mechanism cannot deal with threads.

The static strategy has a lot of potential. It can extract information from a program's source code without executing the app [34]. However, it has problems with thread usage in apps, which is a great drawback. In Strategy 4, we analyze this vulnerability and present a scenario developed to validate their existence.

3.3 Summary

In this Section, we fulfill two objectives. First, we satisfy the objective: *Analysis of energy certification for mobile applications*, we analyzed five energy certification mechanisms [1, 2, 3, 4] and three more tools that allow an energetic analysis [5, 6, 7]. We have achieved our goal of *Identify vulnerabilities in energy certification mechanisms*. During our evaluation, we discovered five vulnerabilities in the mechanisms.

- **Vulnerability 1:** *App vendors submit energy consumption measurement tests along with the app.* Suppose the app stores require app vendors to submit consumption tests. Then, app vendors can easily submit adulterated tests that permit the app to consume less energy. This vulnerability does not affect static analysis approaches.
- **Vulnerability 2:** *The background service consumption is not regarded in the energy certification mechanism.* Some apps consume energy while inactive, so if the certification process does not consider this type of consumption, these apps may benefit. This vulnerability was explored in Strategy 1, in Section 4.1.
- **Vulnerability 3:** *The helper app consumption is not regarded in the energy certification mechanism.* Explored in Strategy 2, on Section 4.2, where we develop a scenario to validate the vulnerability veracity. The certification process is unfair if it does not regard the helper app consumption. Helper apps perform tasks demanded by the app under test (AUT) and can consume substantial energy.
- **Vulnerability 4:** *A refactored app is considered energy efficient.* Refactoring tools detect optimization problems in the app. However, if the refactoring tools do not detect any problem, it does not mean the app is fully optimized. Because some problems may be undetectable, we analyzed this vulnerability and designed a simulation in Strategy 3, in Section 4.3.
- **Vulnerability 5:** *The evaluation mechanism cannot deal with threads.* The tools based on static analysis that analyze a program's paths have the limitation of cannot following two paths simultaneously, which prevents them from dealing with threads. In Strategy 4, in Section 4.4, we present a better analysis of this vulnerability.

In the approach chapter, Chapter 4, we present our strategies to validate the existence of the vulnerabilities identified in this Chapter.

Chapter 4

Approach

The energy certification mechanisms must be reliable, enabling developers, providers, and users to compare apps concerning their power consumption behavior. Otherwise, developers can artificially explore their products to rank as high as possible. In Chapter 3, we examined certifications for apps and explored their vulnerabilities. This Chapter presents approaches to demonstrate how developers could exploit those vulnerabilities.

In the following sections, we present strategies to simulate how developers could exploit the vulnerabilities identified. The strategies describe how we approach the susceptibilities, explain our decisions, and present our scenarios. We explore the identified vulnerabilities: 2, 3, 4, and 5 to validate that developers can exploit them. Vulnerability 1 is already well-defined and does not need more validation; developers can manipulate energy consumption measurement tests submitted with the application. Subsequently, in Chapter 5, we present the detailed experimental setup of the experiments defined in the strategies. Finally, we exhibit the experimental results in Chapter 6.

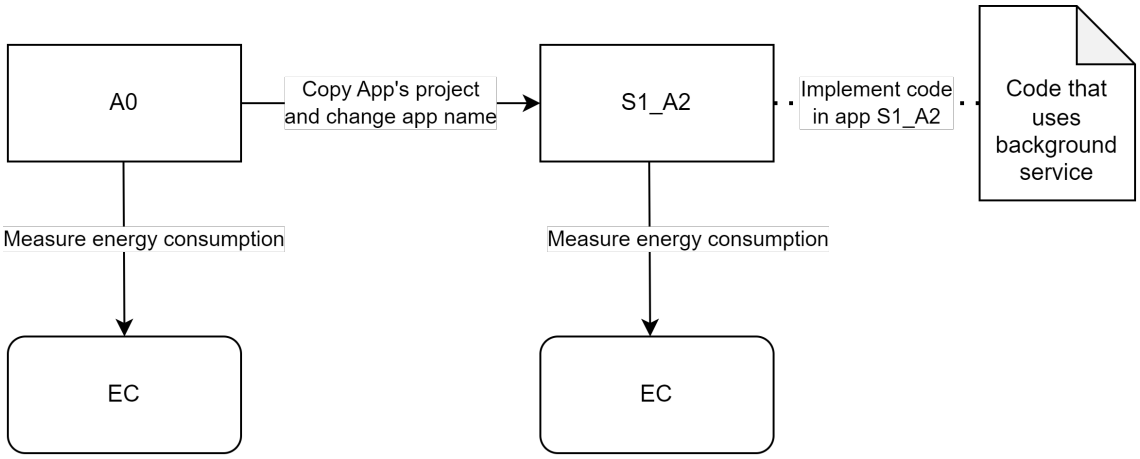
4.1 Strategy 1: The Background Service Consumption Is Not Regarded in the Energy Certification Mechanism

This strategy wants to simulate how developers could exploit the vulnerability: *Vulnerability 2: The background service consumption is not regarded in the energy certification mechanism.*

Most energy certification mechanisms for apps do not consider background service consumption; only Wilke et al. [4] have considered this type of consumption. Therefore, this strategy wants to emphasize the importance of background service by exploiting the certifications [1, 2, 3] that did not account for the background service.

4.1.1 Scenario

To create adversarial strategies, we design a scenario presented in Figure 4.1. We compare the consumption of two similar apps with similar foreground consumption. Although, one of the applications has a high background consumption compared to the other. Therefore, the applications will receive an equivalent energetic evaluation despite the different consumption for the same functionalities.



A0: Original app
S1_A2: Copy of A0 with the implementation of the code that uses background service
EC: Energy consumption of the app (including background service consumption measurements)

Figure 4.1: Scenario representing the implementation of background service, for comparison of consumption with the original app.

In this experiment, app S1_A2 is originated by the modification of S1_A1. App S1_A2 contains changes that cause the app to consume a lot of energy with the background service. Then we can measure the consumption of both apps. We estimate the app's consumption while running with the same use cases and under the same conditions to ensure reliability. Furthermore, we also tested the energy consumption of the apps while not running.

Apps can consume battery while not used; for example, email client apps such as Gmail[1] and Microsoft Outlook[2] that synchronize in background. We have discovered two methods that cause the device's battery to drain even when the app is not in use.

- We can use Services[3], an Android application component, to ensure that the app does tasks after closing it. This approach keeps the app performing tasks despite not being in the foreground. The drawback of Services is

[1] https://play.google.com/store/apps/details?id=com.google.android.gm&hl=pt_PT&gl=US (Last Access: Jul 8, 2023)
[2] https://www.microsoft.com/en-us/microsoft-365/outlook-mobile-for-android-and-ios (Last Access: Jul 8, 2023)
[3] https://developer.android.com/guide/components/services (Last Access: Jul 8, 2023)

that the app must inform the user that a service is running while they are active. Therefore, the app must notify the user that the service is running; the notification only persists until the service ends.

- Another approach is to create a WorkRequest[4]. Typically used to notify users, it allows scheduling activities to perform at a specific time. This approach will enable us to schedule work even though the app is not running.

The most suitable approach is to create a WorkRequest, which fits more with our experiment than Services. WorkRequest allows performing work without affecting the user too much or changing the application's structure. On the other hand, the persistent notification caused by Services impacts the user interaction with the app. We aim to select an approach that does not change app functionalities or add new ones. Therefore, by notifying the user, Services adds new functionality to the app.

We want the addition of background service not to affect the app's operation or functionalities. The background service must consume the device's energy without performing valuable work. It can be considered an inefficiency problem that must be penalized by energy certifications.

4.2 Strategy 2: The Helper App Consumption Is Not Regarded in the Energy Certification Mechanism

With this strategy, we aim to simulate how developers could exploit the vulnerability: *Vulnerability 3: The helper app consumption is not regarded in the energy certification mechanism.* Most energy certificates do not regard this vulnerability.

Energy certification mechanisms do not clarify how to approach cases where the AUT needs to call another app, a helper app, to complete the use case test. Only Wilke et al. [4] considered this and ignored the helper app consumption. The helper app can do most of the work to complete some functionalities of the AUT.

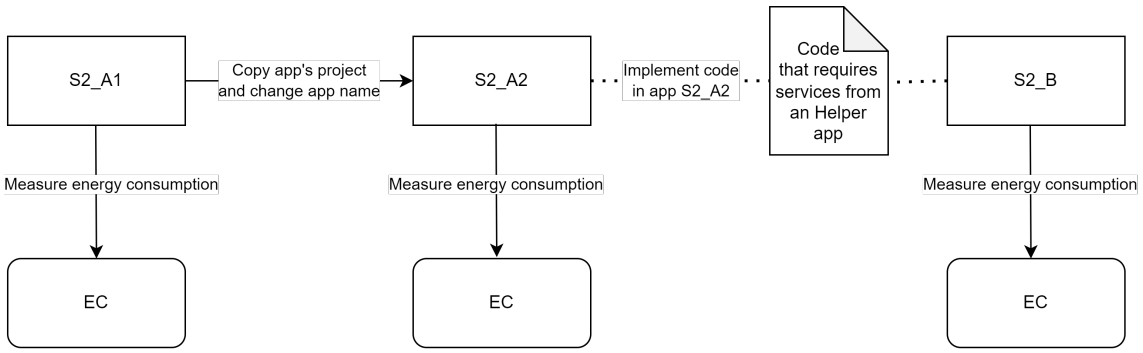
Therefore, when energy certifications do not consider the helper apps' consumption, developers can start to delegate app functionalities to these apps. Then, the measured consumption of an AUT will be a fraction of the actual consumption.

4.2.1 Scenario

As we can verify in the scenario represented in Figure 4.2, the app S2_A1 energy consumption can be compared to app S2_A2 and the helper app (B). First, we need to define app S2_A1. Then copy app S2_A1 to create app S2_A2, which must be modified to delegate services to helper apps. For example, we can delegate video broadcasting to the Browser; therefore, the Browser will perform app tasks

[4] https://developer.android.com/guide/background/persistent/getting-started/define-work (Last Access: Jul 8, 2023)

that do not count toward the AUT energy consumption. Finally, we can compare the energy consumption of S2_A1 with S2_A2.



S2_A1: Original app
S2_A2: Copy of S2_A1 with the implementation of code that requires services from a helper app
S2_B: Helper app
EC: Energy consumption of the app

Figure 4.2: Scenario representing the implementation of helper app calls, for comparison of consumption with the original app.

4.3 Strategy 3: A Refactored App Is Considered Energy Efficient

This strategy wants to simulate how developers could exploit the vulnerability: *Vulnerability 4: A refactored app is considered energy efficient.* This vulnerability is directed at using refactoring tools in the certification process. We explored a certification mechanism by Almasri et al. [1] and an evaluation tool, Kadabra, by Gregório N. et al. [5].

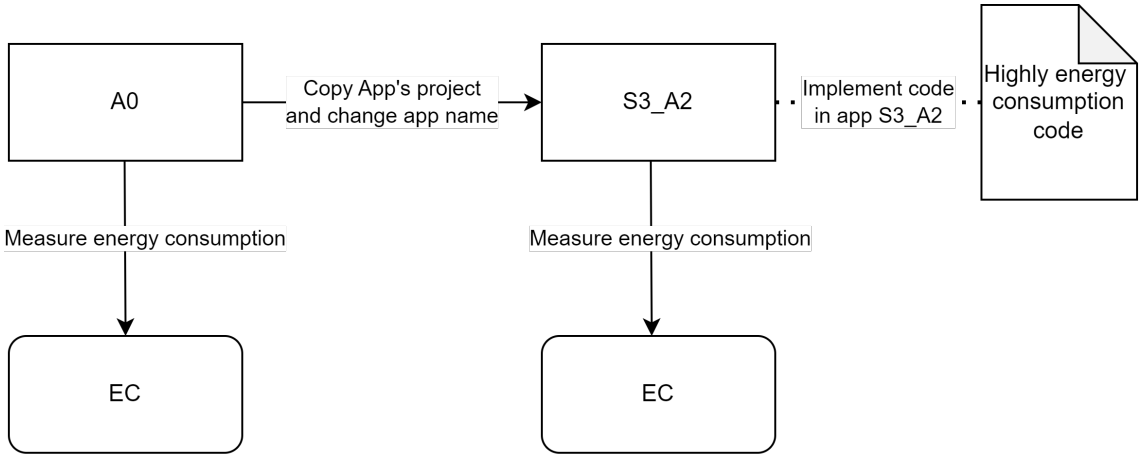
Refactoring tools detect anti-patterns and suggest optimizations to the developer, and they can be handy in enhancing app efficiency. These tools typically output the type and location of the problems detected. Therefore, they may be able to perform an energetic evaluation by returning the number of issues, and then, the apps are evaluated based on that output.

4.3.1 Scenario

We designed a scenario represented in Figure 4.3 that demonstrates that the refactoring tools can attribute a good EE evaluation to an app that is not optimized. First, we select the code that produces a high energy consumption, and the refactoring tools cannot detect it. This code is responsible for the app's consumption increase. Then, we created S3_A2, a copy of app A0, where we implemented the code selected. Finally, we can compare the energy consumption of A0 and S3_A2.

The consumption of S3_A2 is expected to be greater than A0 despite having the

same detectable issues. Therefore, with entirely different consumption, both apps get the same evaluation.



S3_A1: Original app
S3_A2: Copy of A0 with the implementation of the high energy consumption code
EC: Energy consumption of the app

Figure 4.3: Scenario representing the implementation of high consumption code in S3_A2 for comparing with the original app A0 consumption.

The following presents how we approach Almasri et al., [1] Approach.

Almasri Approach: A0 and S3_A2 apps have the same number of refactoring opportunities detectable by refactoring tools. Hence, both apps receive the same EE according to the energy certification shown in Figure 3.1 despite the different energy consumption of the apps. They performed the apps' refactoring using the EARMO. The EARMO tool suggests refactoring opportunities that the developer can manually apply [37]. To ensure the reliability of this experimental scenario, we must perform the refactoring using the tool.

Concluding, we perform the refactorings presented in the scenario with the tools referred to in this Section. EARMO, used by Almasri and Kadabra, used by Nelson Gregório. In Chapter 5, we present the setup of this experimental environment.

4.4 Strategy 4: The Evaluation Mechanism Cannot Deal With Threads

This strategy wants to simulate how developers could exploit the vulnerability: *Vulnerability 5: The evaluation mechanism cannot deal with threads.* This problem was detected in the wcec-android tool developed by Kelson D. et al. [6].

Wcec-android is based on a static approach, which allows estimating the consumption without running the app. We already explained how the tool works in

Subsection 3.2.5, where we understood that this approach has great potential.

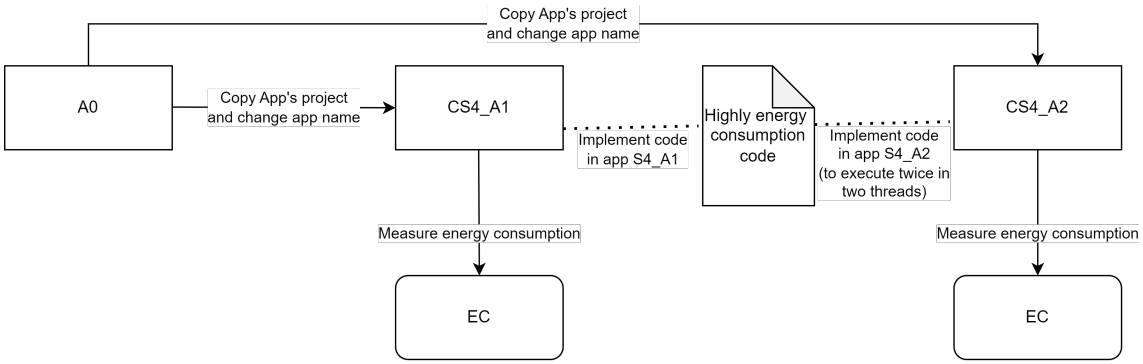
We noticed that the tool does not account for simultaneous paths when building the CFG that threads can originate. However, threads are familiar in apps, so it is a limiting factor that the static strategy has not yet overcome. Therefore, apps with multiple threads may have a different consumption estimation because the tool will only consider the consumption of one of the running threads. The tool may only be used in apps without threads, but it is a substantial limiting factor.

4.4.1 Scenario

Understanding that the tool cannot deal with multiple threads running simultaneously, we can exploit that limitation. Hence, according to the tool, an app with two or more threads will consume the same energy. In this strategy, we developed a scenario that helps us simulate how threads can impact the energy consumption of an app. To make a fair comparison, we create two versions of the same app: one with one thread (S4_A1) and another with two threads (S4_A2) performing the same work.

Figure 4.4 displays the test scenario of this strategy. We modify the original version of the app (A0), creating two modified versions of A0: the S4_A1 and S4_A2 versions. Then we measure the consumption of both versions to analyze further. We also collect the output from wcec-android and verify if our supposition is correct.

To verify the existence of the limitation, the consumption of S4_A2 must be higher than S4_A1, and the output from the wcec-android for both tools must be similar.



A0: Original app
S4_A1: Copy of A0 with the implementation of the high energy consumption code
S4_A2: Copy of A0 with the implementation of the high energy consumption code (implemented in two threads to execute twice simultaneously)
EC: Energy consumption of the app

Figure 4.4: Scenario representing the implementation of high consumption code for comparison of consumption with the original app.

4.5 Summary

In this Chapter, we designed four strategies to simulate how developers could exploit the vulnerabilities identified in Chapter 3, which allowed us to complete one more objective: *Design scenarios that simulate how app developers can exploit the vulnerabilities to obtain a better evaluation than they should in the energy certification mechanisms*. Each strategy contains a scenario that helped us defeat the evaluation tools. The strategies explored are:

- ***Strategy 1:*** *The Background Service Consumption Is Not Regarded in the Energy Certification Mechanism* explores ***Vulnerability*** 2. In Section 4.1, we demonstrated that using the PeriodicTaskWorker class, the app can consume the device's battery when it is not in use. Then, we can collect the energy consumption used when the app is in the background.
- ***Strategy 2:*** *The Helper App Consumption Is Not Regarded in the Energy Certification Mechanism* explores ***Vulnerability*** 3. Some energy certification mechanisms do not consider the consumption of helper apps. Our Strategy, presented in 4.2, permits us to compare the energy consumption of an app that reproduces a video embedded with an app that redirects the user to a YouTube video, reproducing the same video. YouTube acts as a helper app. Then, we collect the consumption of both apps and the helper app.
- ***Strategy 3:*** *A Refactored App Is Considered Energy Efficient.* explores ***Vulnerability*** 4. Refactoring tools may not detect all energetic problems in an app. Therefore, we aim to compare the consumption of two apps with the same refactoring problems detected despite one having undetectable problems. In Section 4.3, we present our approach, which allows us to compare the consumption of the two apps and collect the evaluation data using refactoring tools.
- ***Strategy 4:*** *The Evaluation Mechanism Cannot Deal With Threads.* explores ***Vulnerability*** 5. Tools that find the WCEC using static analysis interpret the app's source code and trace all possible execution paths. These paths can only flow in one direction; therefore, the tool may not give the correct evaluation if some app has threads. In Section 4.4, we present a scenario that allows the validation of our statement by comparing two similar apps, one with one additional thread and the other with two additional threads. Then, we collect the consumption of both apps using an energy measurement method and perform the evaluation using a WCEC tool.

We remember ***Vulnerability*** 1***:*** *App vendors submit energy consumption measurement tests along with the app*. App vendors can easily submit adulterated tests that permit the app to consume less energy. This vulnerability does not need further validation; therefore, we did not build a test environment.

In the next chapter, Chapter 5, we present the experimental setup necessary to reproduce our experiments defined in the strategies. Chapter 6 presents the results and discussion of our experiences.

Chapter 5

Experimental Setup

This Chapter explains how readers can replicate the experiences. We collected a set of apps of different categories from F-Droid[1]. This well-known app store provides the source code of the apps published. We choose one app from six meaningful categories. The apps chosen are displayed in Table 5.1, with all their essential information. The complete apps' information is presented in Appendices on Table A.1. We developed the app Challenge 1, where we made the first modifications and tests. All apps were modified accordingly with each strategy and used in the experiments. However, in Strategy 2, we use an app specially created for the experiments.

App Number	App Name	Source	Category
1	Challenge1	Own App	No category
2	Fast N Fitness	F-Droid	Sports & Health
3	BTC Map	F-Droid	Money
4	Vector Pinball	F-Droid	Games
5	ArityCalc	F-Droid	Science & Education
6	omWeather	F-Droid	Internet
7	Qwotable	F-Droid	Reading

Table 5.1: Information about the apps used in the experiences.

For each app, we developed a specific test case that represents the typical use of the app. The typical use of the app depends on the app category and functionalities. The tests were done with the saving mode off, and the battery capacity above 80% in an Android device used in the experiments was a Samsung Galaxy A22 5G with 4GB of RAM and a battery capacity of 5000 mAh, running on Android 13. Table 5.2 presents other important information about the device. The experimental results obtained in Chapter 6 are relative to this device. Therefore, we cannot guarantee that similar results will be achieved with other devices.

To handle and build the apps, we used Android Studio IDE, the Electric Eel | 2022.1.1 version, where we could change the app IDs to allow multiple versions of the same app on the device. Each version is modified to reach the goals of

[1] https://f-droid.org/en/packages/ (Last Access: Jul 8, 2023)

Component	Specification
Name	Samsung Galaxy A22 5G
Launch	June 2021
Screen	TFT LCD, 90Hz
	6.6 inches
	1080 x 2400 pixels
Storage / Memory	64Gb / 4Gb RAM
OS	Android 13, One UI core 5
CPU	Octa-core (2x2.2 GHz Cortex-A76
	& 6x2.0 GHz Cortex-A55)
GPU	Mali-G57 MC2
Chipset	Mediatek MT6833 Dimensity 700 (7 nm)
Battery Capacity	5000 mAh

Table 5.2: Specifications of the smartphone used in the experiments.

a specific strategy. We choose Android Studio IDE because it is amply used for Android development.

We compare two app versions in the strategies to acknowledge the consumption differences in the exact scenarios. We used BatteryStats[2] to measure energy consumption, providing critical information about the app's operation. It was already presented in Section 2.1. This tool is the most appropriate for our situation. We collected multiple pieces of information in each run, such as foreground and background time, CPU time, consumption in the foreground and background, consumption of the cache, consumption of the screen, and total consumption. We exclude screen consumption in our experiments because it cannot be fully controlled despite representing much of the app consumption.

To increase energy consumption, we focus on increasing the CPU usage of the app. Other methods were considered, like using high-consumption hardware components of the device such as WIFI and GPS; however, with these approaches, we need to add app' permissions that are not in our interest. Therefore, to increase CPU usage, we implemented a thread that constantly performs work to increase CPU usage. However, the work to be done by the thread is indifferent because it will be running indefinitely, causing high consumption regardless. Therefore, we use an array sorting algorithm to increase the CPU usage in Strategies 1, 3, and 4, the Bubble Sort [3]. BubbleSort is known to be one of the most time-consuming sorting algorithms, to sort a vector.

Each app has been tested a minimum of 15 times on the same device and under the same circumstances to reduce data uncertainty. All other apps were closed, and all unnecessary hardware components were disabled, such as GPS and WIFI.

We created a GitLab repository [4] that contains all the code necessary for the ex-

[2] https://developer.android.com/topic/performance/power/setup-battery-historian (Last Access: Jul 8, 2023)

[3] https://www.geeksforgeeks.org/bubble-sort/ (Last Access: Jul 8, 2023)

[4] https://gitlab.com/Ramalho2000/daniel-ramalho-dissertation (Last Access: Jul 8,

periments; it also includes a ready-to-use example app from each Strategy.

5.1 Strategy 1

As previously discussed, we aim to prove that energy certification mechanisms must consider background energy consumption. Therefore, we developed a way of allowing the app to consume energy while not in use. However, the app must not consume more in the foreground. We already decided that WorkRequest[5] is the best method where we can schedule the work to only begin when the app is closed and stop when the app is open again. We used the class PeriodicWorkRequest extended from WorkRequest, which allows the scheduling of periodic tasks. The work selected that consumes much energy is the BubbleSort.

To implement our approach, we start by placing the demanded dependencies, presented in Listing 5.1. Without this dependency, we cannot create a WorkRequest.

```
dependencies {
  //https://developer.android.com/jetpack/androidx/releases/work#groovy
  def work_version = "2.7.0"
  implementation "androidx.work:work-runtime:$work_version"
}
```

Listing 5.1: Strategy 1 dependencies

In our GitLab repository, we present the code required for creating the class `MyPeriodicTaskWorker`, which extends the `Worker` class. This class performs work synchronously on a background thread [6]. In the overridden method `doWork()` of the `Worker` class has placed the code that we want to be executed synchronously in the background.

We verify if the maximum time of background time has been exceeded and if the app is not running. If these conditions are met, the device starts to perform the work continuously. On the other hand, if the background time has been exceeded, all Workers are canceled, and the background tasks end.

Finally, in the app's `MainActivity`, we need to schedule tasks for our `MyPeriodicTaskWorker` class. But first, we validate if the app is not running. The method `InitBackgroundTasks()`, called by the `MainActivity`, starts the background activity. For the sake of the experiment, we defined a maximum background time; otherwise, the apps will drain the battery indefinitely. Then, we create a PeriodicWorkRequest and enqueue it in the WorkManager.

Testing Method As to the consumption collection method of the applications, the apps were tested for 30 minutes. We measured the app consumption all at

2023)

[5] https://developer.android.com/guide/background/persistent/getting-started/define-work (Last Access: Jul 8, 2023)

[6] https://developer.android.com/reference/androidx/work/Worker (Last Access: Jul 8, 2023)

once since there were many tests to be done, and each one took a lot of time. Therefore, the consumption collected would not be the actual consumption of the app if it were running without the interference of the other apps. However, the experimental setup defined is enough to validate our strategy. We tested app number 1 of Table 5.1, Challenge1, isolated and collected the consumption data acting as a control version. Based on the consumption of the control version, we can deduce that the other apps must have a similar consumption when running isolated.

5.2 Strategy 2

In strategy 2, we aim to prove that energy certification mechanisms must regard the consumption of helper apps. Therefore, we developed an app that displays a video and another similar to the first one that redirects the user to a video player app. The video the apps show is the same to ensure a fair comparison.

To test the app's consumption, we open the app and wait for 10 seconds before pressing the button to start the video. Then, we wait until the video ends, which lasts exactly 1 minute. Therefore, the total test time is 70 seconds, of which 60 seconds is video reproducing. Finally, we collect the consumption information of both apps and the consumption of YouTube used by the second app.

Figures 5.1 and 5.2 present the flow of the app used in the experiment. In the first Figure, the app shown is app S2_A1, which reproduces an embedded video. Users, after pressing the button *WATCH VIDEO*, a YouTube player appears and displays a video. In the second Figure, the app shown is app S2_A2, which redirects the user to a video on YouTube after pressing the same button as the first app.

In the following subsections, we present an overview of the code used, and our GitLab repository presents the complete code.

5.2.1 App S2_A1 Setup

To display a video in the app, we used android-youtube-player[7], a stable YouTube player for Android that can be customized to our needs. This library could incorporate a YouTube video player in our app. The library simplifies the Google recommendation of inserting the IFrame Player API inside a WebView. It is reliable and is used by more than 5 thousand apps.

To implement the library, first, we need to add the dependency presented in Listing 5.2. The dependency is added in *build.gradle* file on the project level. Without adding this dependency, we cannot use the library methods.

```
1 dependencies {
2   //dependency for youtube-player
```

[7] https://github.com/PierfrancescoSoffritti/android-youtube-player (Last Access: Jul 8, 2023)

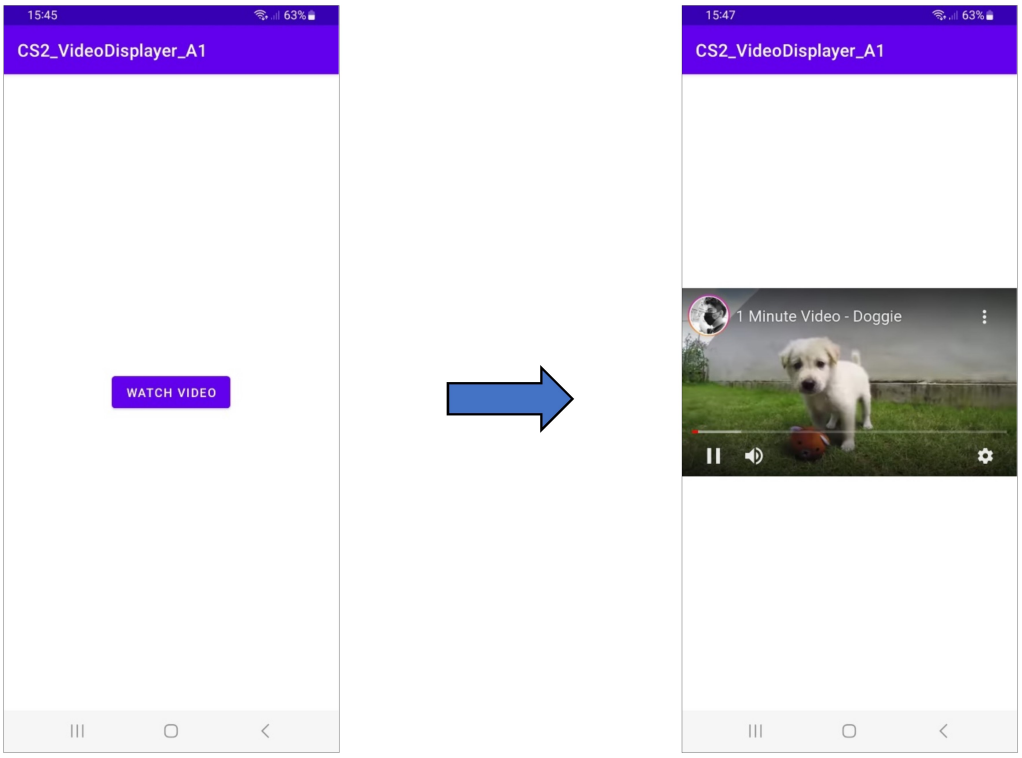


Figure 5.1: Example of the app that reproduces a YouTube video (S2_A1)

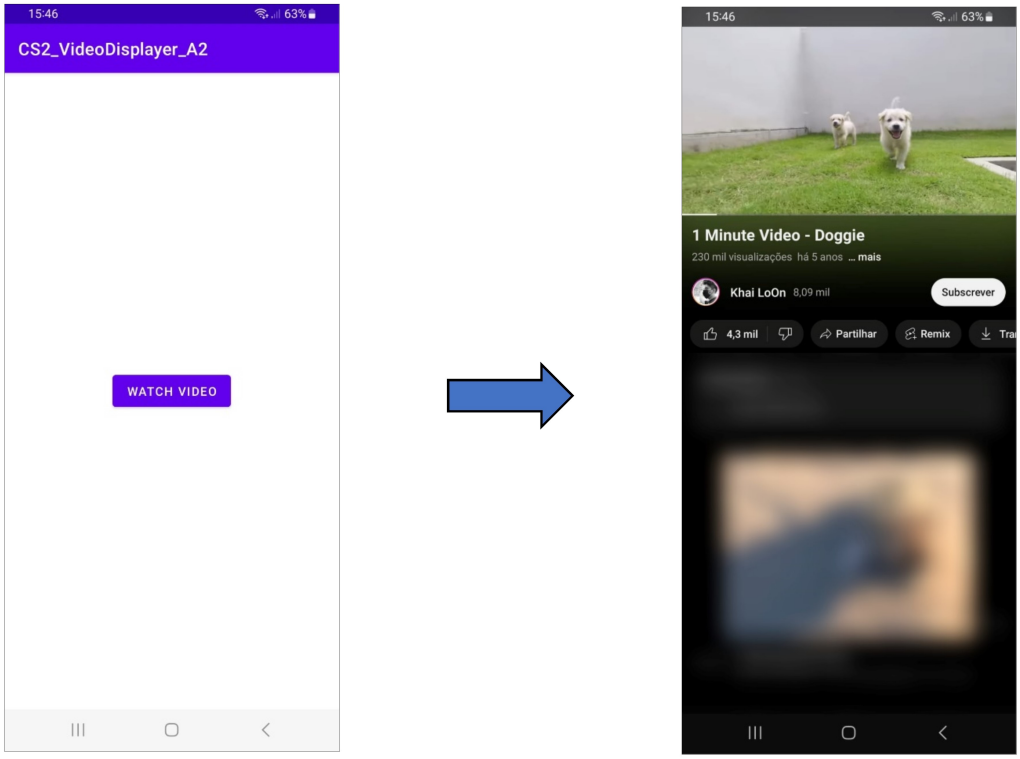


Figure 5.2: Example of the app that redirects to a YouTube video (S2_A2)

```
3     //URL: https://github.com/PierfrancescoSoffritti/android-youtube-player
4     implementation 'com.pierfrancescosoffritti.androidyoutubeplayer:core:12.0.0'
5 }
```

Listing 5.2: android-youtube-player dependency in S2_A1 app of Strategy 2.

Secondly, we create a new Fragment that contains all the necessary code to reproduce the video in the app. Moreover, we must define the video player layout `fragment_video_player` in the layouts directory. This layout creates an `YouTube PlayerView` centred on the screen.

Then, in the Fragment, we can change the `videoId` as we like, it can be obtained from the YouTube video URL. The Id is shown in the URL after the *watch?v=* keyword; for example: in *https://www.youtube.com/watch?v=YLslsZuEaNE* URL, the `videoId` is *YLslsZuEaNE*. Eventually, the Fragment loads the YouTube video chosen into the `YouTubePlayerView` defined in the layout.

Finally, we must create an instance of the Fragment `videoPlayerFragment` and display it to the user as presented in the `MainActivity`. We implemented a `onClick Listener`, which requires a button `watchButton` to function. The listener handles the click, replacing the present content with the Fragment content.

5.2.2 App S2_A2 Setup

In the second app, to redirect the user to the video player, we used Intent[8] with the action: `ACTION\_VIEW`, which allows the user to be redirected to another page with a click of a button. We redirect the user to YouTube, where the video is reproduced.

In Listing 5.3, we present the code necessary to replicate our experiment. The `String url` can be changed as we want; it identifies the URL to where the user will be redirected. To implement this, we must create a button named `watchButton`, and the listener presented handles the click. Then, when the user clicks the button, he is redirected to the URL defined. And the user can return to the app by pressing the back button on the device.

```
1 watchButton.setOnClickListener(v -> {
2   //Place the URL of the YouTube video
3   String url = "https://www.youtube.com/watch?v=YLslsZuEaNE";
4   Intent i = new Intent(Intent.ACTION_VIEW);
5   i.setData(Uri.parse(url));
6   startActivity(i);
7 });
```

Listing 5.3: Code necessary for the app S2_A2 of Strategy 2.

5.3 Strategy 3 and 4

Strategies 3 and 4 have similar setups; therefore, they can be explained in the same section. We used threads implementing the class Thread in Java in both strategies. The only setup difference is that in Strategy 4, two threads are created to perform the task, and in Strategy 3, only one instead. We form arrays with

[8] https://developer.android.com/guide/components/intents-common(Last Access: Jul 8, 2023)

random numbers, and then the threads order them indefinitely using the BubbleSort, consuming much energy. The complete code to which we refer below is presented in our GitLab repository.

In the `MainActivity`, to allow continuous CPU usage, we need to apply some modifications. First, we need to add the method `bubbleSort()`, already explained. Secondly, we add our own created class `OrderArrayThread`, which extends the Thread class. Inside the class is placed an infinite loop continuously calling the `bubbleSort()` method. Finally, in the `onCreate()` method, where usually is placed the initialization code, we created an instance of `OrderArrayThread` named t1. Then, we can begin the execution of the thread using the `start()` method, which allows two threads to run concurrently.

As we already discussed, strategies 3 and 4 setup is very similar. The difference is in the code placed on `onCreate()` method. In this method, two threads must be created instead of one. To reach that, we must create a thread t2 instance of `OrderArrayThread` and call the `start()` method as we did previously.

All apps here tested for a timestamp of 1 minute to allow a meaningful comparison between them, despite not being our primary purpose because the apps are from different categories. We consider this timestamp suitable for bringing meaningful data.

5.3.1 EARMO and Kadabra Setup

In Strategy 3, we must set up EARMO and Kadabra tools. Both tools are capable of analyzing apps and outputting their evaluations. The evaluation is the number of energy greedy patterns identified.

EARMO Setup To use EARMO, we need to execute a jar file that analyzes the app Android Package Kit (APK) or source files. Before that, files unrelated to the app code, such as Google dependencies, must be deleted otherwise, the program will fail. We used the default properties of the tool.

Kadabra Setup In the Kadabra case, the tool automatically removes these files, therefore, we can provide the APK to the tool without any changes. Again, is executed a jar file that examines the app. The tool uses the app's package name in the analysis process. This gave us some problems because our test apps do not have their actual package name, we had to change it to allow the existence of multiple app versions. This is easily corrected by saving the APK with its original package name.

5.3.2 wcec-android Setup

In Strategy 4, we used wcec-android [9] to collect the consumption estimation of the tool. The tool has been developed to be used in a Linux environment, therefore, we used Ubuntu terminal [10] for Windows. We installed all the dependencies needed in the Linux subsystem to execute the tool. Finally, wcec-android outputs the consumption estimation value.

5.4 Summary

We presented the experimental setup necessary to reproduce our experiments, including the eight apps and the device used. We deliver all the required code to be implemented in the apps to reach the objectives of the strategies. The experimental results may vary based on the device used; for example, they can change depending on the OS and CPU versions. Therefore, we cannot ensure that our results relate to all devices. We offer a GitLab repository [11] where all the code is delivered in conjunction with app Challenge1, app number 1 of Table 5.1, which have the code implemented.

In this Chapter, we reached one of our objectives: *Setup of the experimental environment*, presented in Section 1.1. In the next Chapter, we give the results obtained from the strategies and complete our last objective to support our RQ.

[9] https://github.com/DelcioKelson/wcec-android (Last Access: Jul 8, 2023)
[10] https://ubuntu.com/wsl (Last Access: Jul 8, 2023)
[11] https://gitlab.com/Ramalho2000/daniel-ramalho-dissertation (Last Access: Jul 8, 2023)

Chapter 6

Experimental Results and Discussion

We present and analyze the results obtained in the experimental scenarios. The setup of each experiment was defined in Chapter 5. The following four sections contain the strategies results. The power consumption data was collected using Batterystats, and each app has tested at least 15 times. With the data collected, we aim to validate the existence of the vulnerabilities explored in the strategies.

In Figure 6.1, we present the consumption data of the original apps used in the experiments. The figure allows us to observe the data distribution. The maximum standard deviation is 0.07932 clearly achieved by the BTC MAP app, which indicates that the data is not too scattered. The app BTC Map permits users to navigate the map and discover interest points in every corner of the world. Moreover, the consumption data of each app follows a normal distribution.

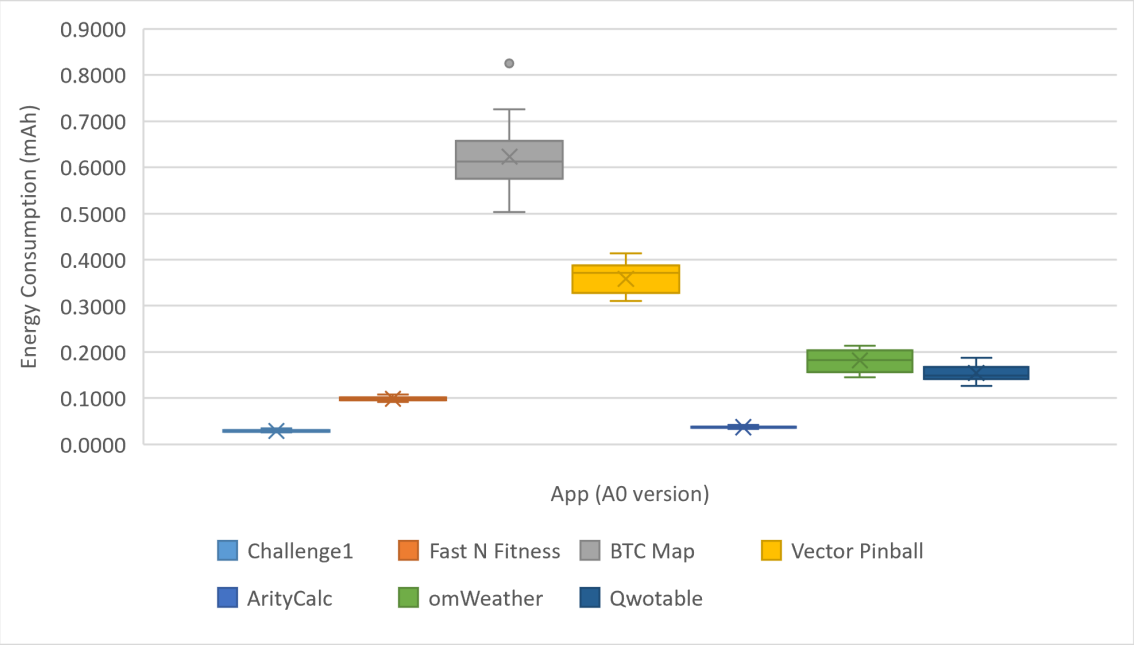


Figure 6.1: Box plot of the original apps energy consumption

The apps have different CPU usage because they belong to distinct categories allowing multiple functionalities. For example, app number three, BTC Map, per-

mits users to navigate the map and discover interest points in every corner of the world. Which consumes many device resources causing a significant consumption in the original version. On the other hand, for example, app two, named Fast N Fitness, is a straightforward app with low functionalities hence its low consumption. In the same way, it affects the consumption distribution. The more complex apps have a more significant standard deviation.

6.1 Strategy 1

In this experiment, we compared the background consumption of the original app (A0) and a modified version that consumes energy in the background (S1_A2). A0 consumption in the background was null. Therefore, Figure 6.2 shows only the S1_A2 consumption. The original app versions did not consume any energy in the background. However, version S1_A2 of each app consumed a considerable amount of energy. As discussed in Section 5.1, the app consumption presented in the table is obtained with the apps running simultaneously, impacting the actual apps' consumption.

The Control Version has been created to deduct the consumption in the background of the other apps. Therefore, we deduct that the other apps running isolated must have a similar consumption to the Control Version, approximately 32.8926 mAh.

As expected, we got some outliers provoked thanks to the lack of control of the background usage in a device. The execution of background tasks might be delayed because WorkManager is affected by the operative system [1]. The Control version's mean is 32.8926 mAh, and the standard deviation is 2.9490. The maximum standard deviation among the other apps' is 1.8414.

The static analysis tools, Kadabra and EARMO, used in [5, 1], and wcec-android used in [6] gave the same output to A0 and S1_A2. Therefore, the static analysis did not detect the modifications inserted that allowed the app to consume in the background.

[1] https://developer.android.com/reference/androidx/work/PeriodicWorkRequest (Last Access: Jul 8, 2023)

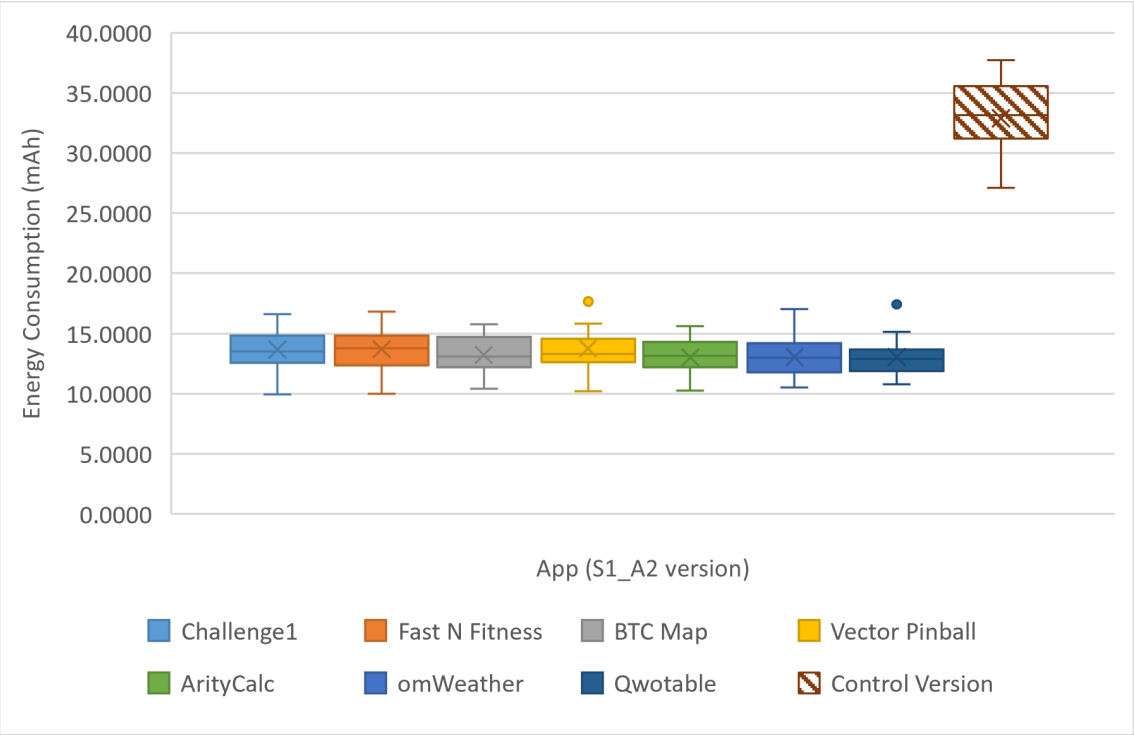


Figure 6.2: Box plot of the background service energy consumption obtained in Strategy 1

The energy certification mechanisms must consider this type of consumption because it can relate to a high part of an app's consumption; an app can drain the battery 24/7 despite not running. For example: considering the consumption of our Control Version, 32.8926 mAh for 30 minutes in the background, and the battery capacity of the device used, which is 5000 mAh, we can expect to be consumed 1.32% of the battery each hour.

Suppose the energy certificate mechanism does not consider the background energy. In that case, it will give the same evaluation to both app versions, A0 and S1_A2, persuading the consumer to believe that the apps have similar consumption. However, as we verified, A0 consumes no energy in the background, and S1_A2 consumes much. From an app vendor's point of view, he knows that the app can consume as much as we want in the background without affecting the energetic evaluation, which may harm the app market. Unfortunately, only one certification mechanism analyzed considered this type of consumption Wilke et al. [4]. The other evaluation technique's [1, 2, 3, 5, 6] gave the same evaluation to apps with and without background consumption.

6.2 Strategy 2

As presented in Section 5.2, we aim to compare the consumption of an app that reproduces a YouTube video embedded (S2_A1) with an app that redirects the user to YouTube (S2_A2); we consider YouTube (S2_B) as a helper app. In this Section, we present the strategy's experimental results.

As we can verify in Figure 6.3, we collected the energy consumption of the original app (S2_A1), the modified version (S2_A2), and the helper app (S2_B). We stacked consumption values of S2_A2 plus B to agile the comparison with the original app.

S2_A1 and S2_A2 are similar and allow equivalent functionality. Therefore, it is expected that both apps have identical consumption. However, the consumption of S2_A2, the app that redirected the user to YouTube, was shallow compared to S2_A1. We registered a mean consumption of 1.0780 mAh and a standard deviation of 0.05635 to S2_A1. To S2_A2, the mean consumption is 0.0261 mAh, and the standard deviation is 0.0031. It is a considerable difference. The consumption of the second app can be virtually discarded because the helper app is responsible for most of it in the test case, with a consumption of 1.1547 mAh and a standard deviation of 0.0424.

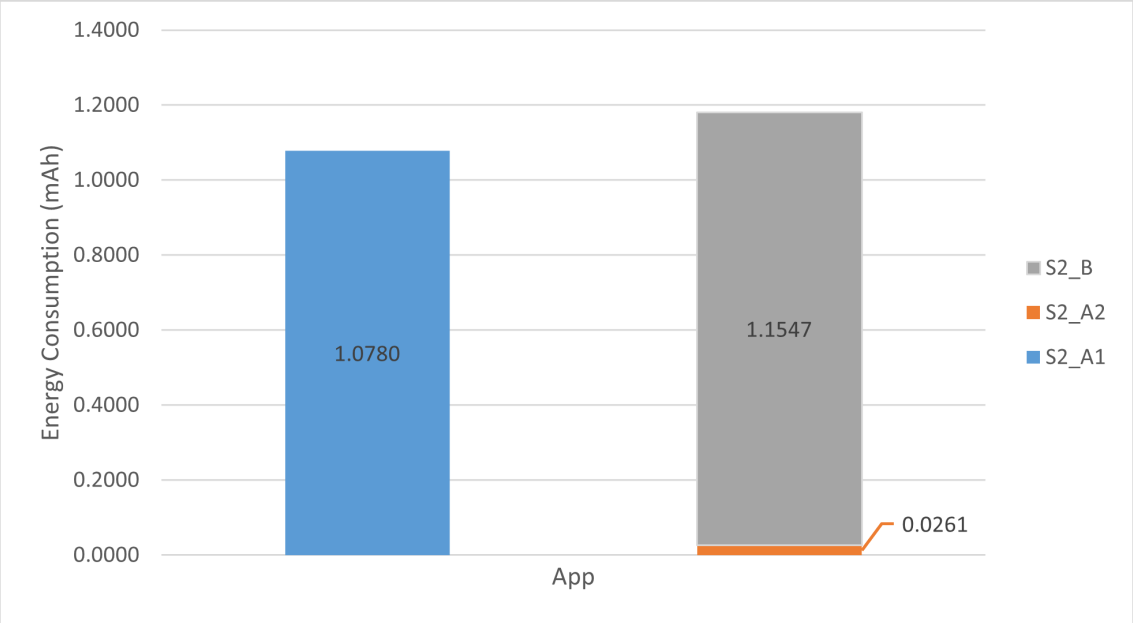


Figure 6.3: S2_A1, S2_A2 and S2_B energy consumption

In Figure 6.4, we can analyze the data distribution. Additionally, the consumption data collected from each app version follows a normal distribution and has low variance.

The agglomerated consumption of S2_A2 and S2_B is similar to that of S2_A1. Hence, it is appropriate to add the consumption of the helper app to achieve the actual energy used to complete the task. If we leave aside the S2_B consumption, the app S2_A2 benefits from the evaluation method.

f the apps were tested using a method that discards the helper apps' consumption, the app vendors might start developing their apps to delegate more services to other apps. Given the strategy presented, it may represent a real case. The developers modify one app that allows video playback originating app S_A2 that redirects the user to the video playback. This way, the app would receive an inflated rating because, as we have seen, the S_A2 consumption was superficial. As discussed in Section 4.2, only Wilke et al. [4] addressed this situation. The

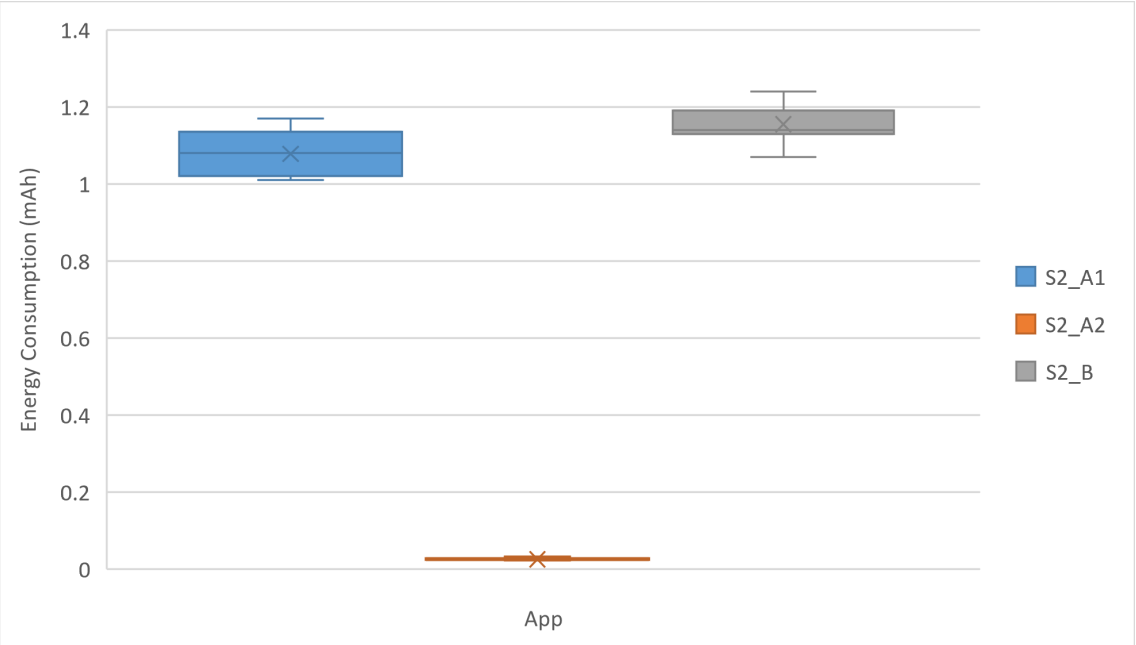


Figure 6.4: Box plot of S2_A1, S2_A2 and S2_B energy consumption

certification mechanisms [1, 2, 3] did not consider these circumstances.

6.3 Strategy 3

Figure 6.5 presents the Batterystats results, where we can interpret that the app's modifications increased their consumption. In this Strategy, we compare the consumption of the original app (A0) with a modified version (S3_A2). The consumption growth was provoked by the increased CPU usage thanks to the thread inserted that continually performs a specific activity. We observed a consumption gain similar in all app modifications; an average increase of 1.9532 mAh and a standard deviation of 0.0860.

The apps have different consumption because of their complexity and functionalities. Figure 6.6 presents the distribution of the app version consumption. The consumption data collected from each app version follow a normal distribution, with low dispersion being 0.3016 the maximum standard deviation. This deviation corresponds to the BTC MAP app. One possible cause is the app's different hardware consumption to render other map parts.

Table 6.1 shows each app version's Kadabra and EARMO tools results. As we can verify, the number of refactoring opportunities detected by each tool did not change when we modified the apps. Hence, the tools did not catch the modifications inserted in the original app leading to the same evaluation obtained by the apps A0 and S3_A2. Therefore, if refactoring tools are used in an energy certification mechanism, apps with different performances could receive the same energetic grade.

With this experiment, we understand that the refactoring tools may not detect

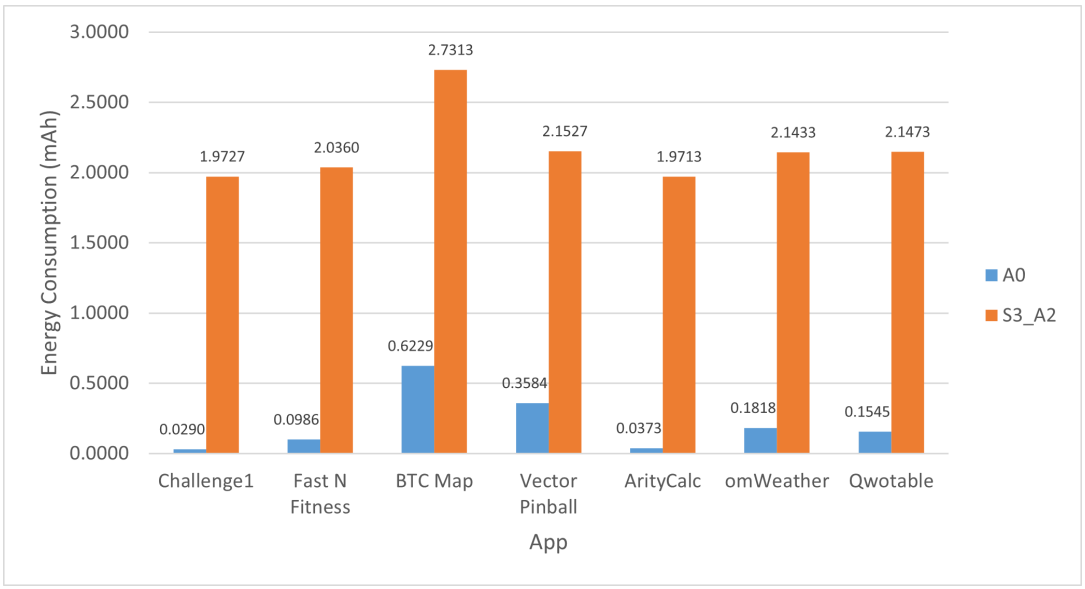


Figure 6.5: Energy consumption comparison between app's version A0 and S3_A2

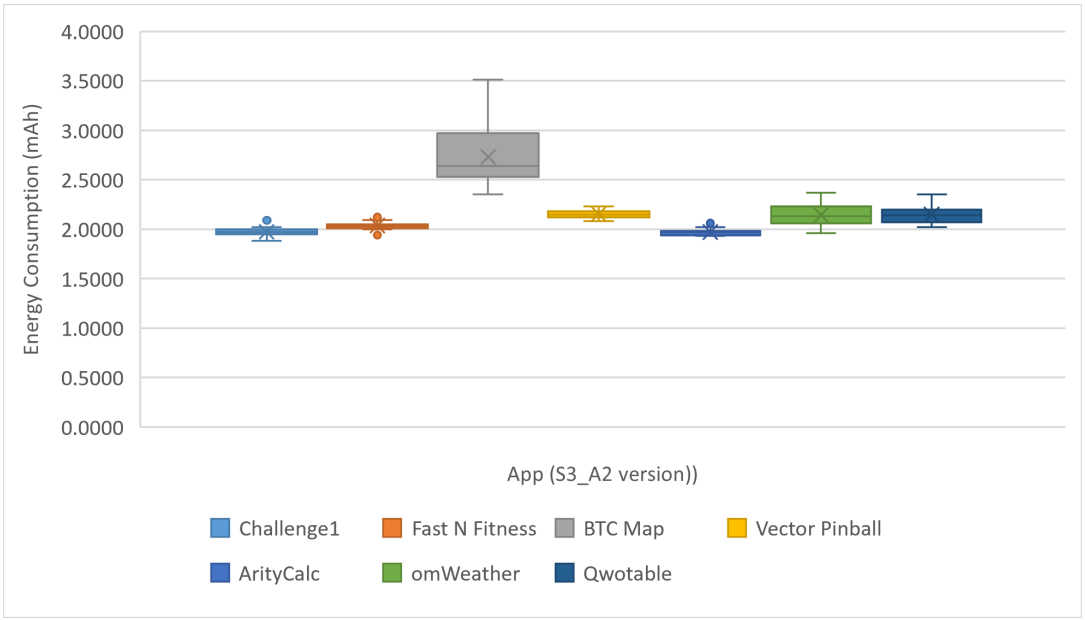


Figure 6.6: Box plot of the S3_A2 energy consumption

App Name	Kadabra		EARMO	
	S3_A1	S3_A2	S3_A1	S3_A2
Challenge1	0	0	0	0
Fast N Fitness	51	51	72	72
BTC Map	0	0	390	390
Vector Pinball	5	5	367	367
ArityCalc	0	0	220	220
omWeather	10	10	647	647
Qwotable	0	0	656	656

Table 6.1: Kadabra and EARMO results

some energetic problems, which can deceive the customer into thinking that the app is fully optimized. Because the tools gave the same evaluation to apps with much different consumption, Some of the undetectable energetic problems may drain the battery severely. Therefore, the refactoring approach may not be adequate for app energy certifications.

However, refactoring tools can quickly analyze an app, which is useful when dealing with many apps. They can examine the app's code without running it. The testing team does not need programming knowledge since the tools handle all the work. There are a lot of advantages, but if the refactoring tools cannot detect some energetic problems, they may give dishonest evaluations.

Despite the apps developed not being optimized and still getting a good grade in the evaluation mechanisms that use refactoring tools. Developers will recognize that refactoring tools do not detect all energetic problems. So, they may develop their apps to exploit those gaps, using inefficient code alternatives that are not detectable to the tools. The evaluation mechanisms [5, 1] used refactoring tools. So, they attributed the same evaluation to apps with much different consumption.

6.4 Strategy 4

In Figure 6.7, we can verify that the app consumption, measured with the Batterystats tools, increased when we inserted the new thread. The blue column represents the app version that contains one thread (S4_A1), and the orange column the two-thread version (S4_A2). Adding a new thread had a similar impact in all apps, with an average increase of 1.8210 mAh and a standard deviation of 0.0593.

To complement our data analysis, Figure 6.8 presents the data distribution of S4_A1 versions energy consumption. The data collected from each app version follows a normal distribution, and the maximum standard deviation is 0.2325, meaning the dispersion is low.

In Table 6.2, we can see the results obtained with the wcec-android tool. The figure shows that the wcec-android outputs a similar consumption estimation for each app version, S4_A1, and S4_A2. The percentage change has been calculated using the formula: $C = \frac{S4\_A1 - S4\_A2}{S4\_A1}$, with which we obtain values close to 0.0%. Hence, the tool gave the same evaluation to app versions with much different consumption, as we verified in Figure 6.7.

We tested the apps using other static analysis methods, the refactoring tools Kadabra and EARMO, used in [5, 1]. Neither of the methods attributed a different evaluation to S4_A1 and S4_A2. We can conclude that the refactoring tools did not report the new thread presence. However, in this Strategy, we focused on the wcec-android tool.

The tool tested, wcec-android, cannot handle thread usage, a significant limitation because many apps use threads nowadays. Similar static analysis approaches may not yet be prepared for app energy certification mechanisms, but

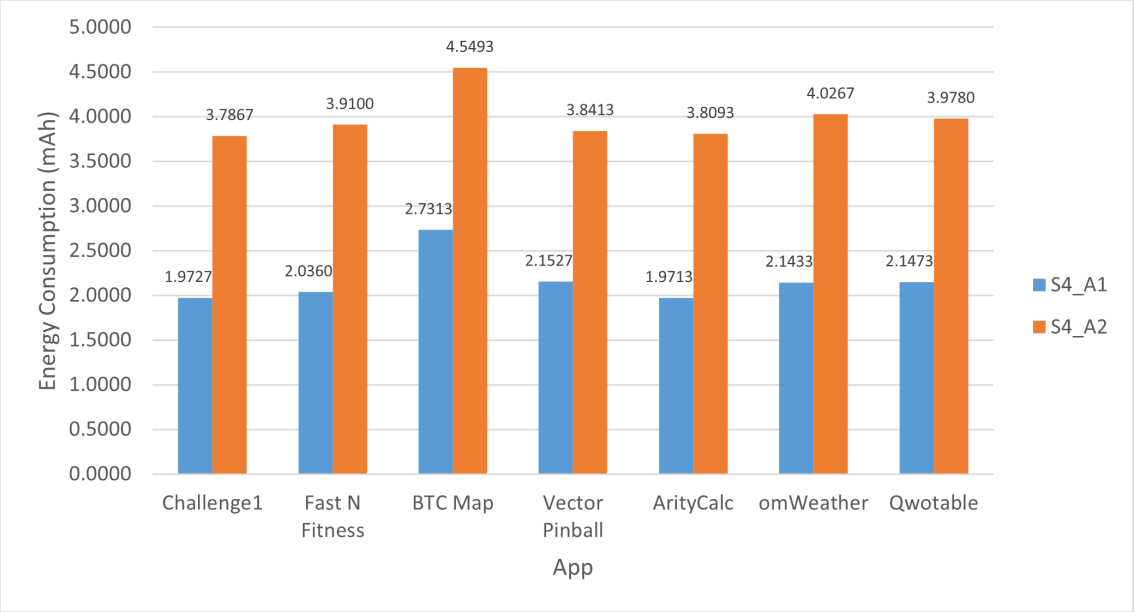


Figure 6.7: Energy consumption comparison between app's version S4_A1 and S4_A2

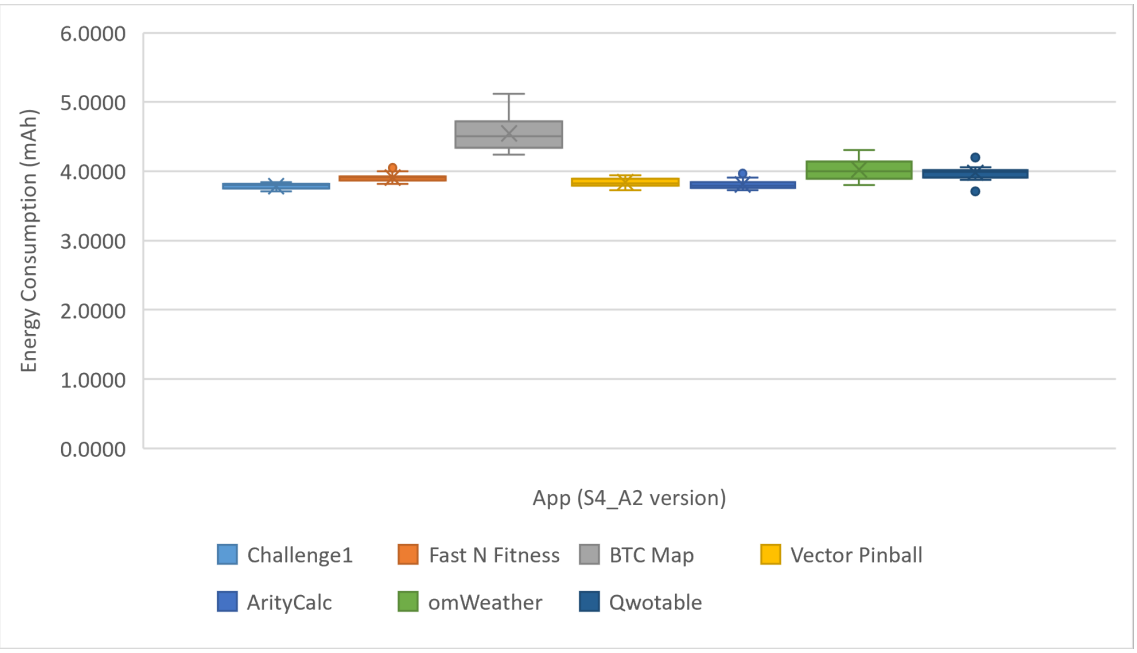


Figure 6.8: Box plot of the S4_A2 energy consumption

we need more analysis to validate this hypothesis. However, if the Worst-Case Energy Consumption (WCEC) approach could overcome the thread limitations, they may be implemented in app stores. Some of the tools' advantages are the testing speed and the testing team not needing app' or programming knowledge. Moreover, the tool can be applied easily in a large set of applications, which is crucial when dealing with millions of apps.

Knowing that the tool does not handle threads well, developers may start implementing more threads in their apps, which can increase the performance and

App Name	S4_A1	S4_A2	Percentage Change
Challenge1	2378.8181	2383.2508	0.2%
Fast N Fitness	2197200.5469	2197200.5469	0.0%
BTC Map	22.7803	22.7803	0.0%
Vector Pinball	1042350.0097	1042350.0097	0.0%
ArityCalc	40667.4708	40667.4708	0.0%
omWeather	3.0092	3.0092	0.0%
Qwotable	7566298.5922	7566298.5922	0.0%

Table 6.2: wcec-android results

functionalities of the app without downgrading in the energetic evaluation. The evaluation mechanism Kelson D. et al. [6] is susceptible to this practice. Other evaluation methods based on static analysis are also vulnerable, such as [1, 5].

6.5 Summary

We validate that energy certification mechanisms for apps have susceptibilities that developers can exploit. The problems were detected in the evaluation techniques [1, 2, 4, 3, 5, 6, 7]. We acknowledge five vulnerabilities raised in Section 3.2:

- **Vulnerability 1:** App vendors submit energy consumption measurement tests along with the app.
- **Vulnerability 2:** The background service consumption is not regarded in the energy certification mechanism.
- **Vulnerability 3:** The helper app consumption is not regarded in the energy certification mechanism.
- **Vulnerability 4:** A refactored app is considered energy efficient.
- **Vulnerability 5:** The evaluation mechanism cannot deal with threads.

Vulnerability 1 was already validated without the need to implement a strategy. Developers can submit manipulated tests that make the app consume less energy than it actually consumes. This vulnerability is presented in Wilke et al. [4].

The remaining vulnerabilities were analyzed in the following strategies:

- **Strategy 1:** In Section 6.1, we validated the existence of **Vulnerability** 2, where we acknowledged that apps could consume much energy in the background. Our apps consumed 1.32% each hour without being used. Therefore, the certification mechanism must address this type of consumption. If

not, developers can develop their apps to consume as much as they like in the background without being penalized. Only Wilke et al. [4] addressed this situation.

- **Strategy 2:** In Section 6.2, we explored **Vulnerability 3**. We obtained conclusive data that allows the validation of the vulnerability. The consumption of the app that used the helper app to complete the functionality has superficial compared to the app that performed the task completely. The app reproducing the embedded video consumed 1.0780 mAh, and the other app consumed 0.0261 mAh. So, developers can redesign their apps to delegate services to others, improving their energetic evaluation. The helper apps usage only was addressed by Wilke et al. [4].

- **Strategy 3:** The experimental results of this strategy, presented in Section 6.3, help validate the existence of **Vulnerability 4**. We noticed that refactoring tools did not detect the code inserted. Therefore, the tools gave the same evaluation to apps with much different consumption. The apps have a mean consumption difference of 1.9532 mAh, having the same energetic evaluation. App vendors can discover other gaps in the evaluation given by the refactoring tools, as we did, and exploit them. This problem was detected in the approaches that used refactoring tools: Almasri et al. [1] and Gregório N. et al. [5].

- **Strategy 4:** Section 6.4 presents the experimental results obtained exploring **Vulnerability 5**. Tools that examine the app's source code and trace all possible paths have the limitation of not being able to follow two paths simultaneously. Therefore, apps with threads may have the same evaluation. In our experiments, apps with a consumption difference of 1.8210 mAh may have the same evaluation. Therefore, developers can add more threads to their apps without affecting the apps' energetic evaluation. This problem is common to the static analysis used by [1, 5, 6]. But more prevalent in the WCEC approach developed by Kelson D. et al. [6].

We completed our last objective: *Implement and evaluate the proposed adversarial strategies designed in the scenarios*, presented in Section 1.1. We also answered our RQ, suggesting multiple ways developers could exploit the certification mechanisms.

Chapter 7

Work Plan

This Chapter reviews our work in the first and second semesters. Section 7.1 presents the summary of the work done in the first semester, and Section 7.2 the tasks done in the second.

7.1 First Semester

We started the first semester by defining the project scope and the research question RQ. Secondly, we researched and analyzed all the valuable information for state-of-the-art. Then, we explore energy certificates proposed for apps and all the necessary information to interpret them, such as the energy measurement methods used to measure an app's energy consumption.

After the research, we explored the energy certificates deeper to identify limitations and vulnerabilities. Then through our strategies, we establish how developers could exploit the certifications to make an app rank higher than it should.

The Gantt chart illustrated in Figure 7.1 exhibits the work done in the first semester; the work was divided into five main steps:

1. **Definition of the research question(s) and project scope:** In the first step, we will define the main objectives of the work.
2. **State-of-the-Art research:** We review the literature on certificate vulnerabilities explored by manufacturers and write the State-of-the-Art.
3. **Definition of the methodology and approach:** We analyzed energy certificates for apps and identified their vulnerabilities and limitations.
4. **Implementation of a proof-of-concept:** We explored the vulnerabilities identified and illustrated how developers could develop their apps to rank higher than they should.
5. **Writing the intermediate report:** Finally, after gathering all the information from the previous steps, we write the intermediate report.

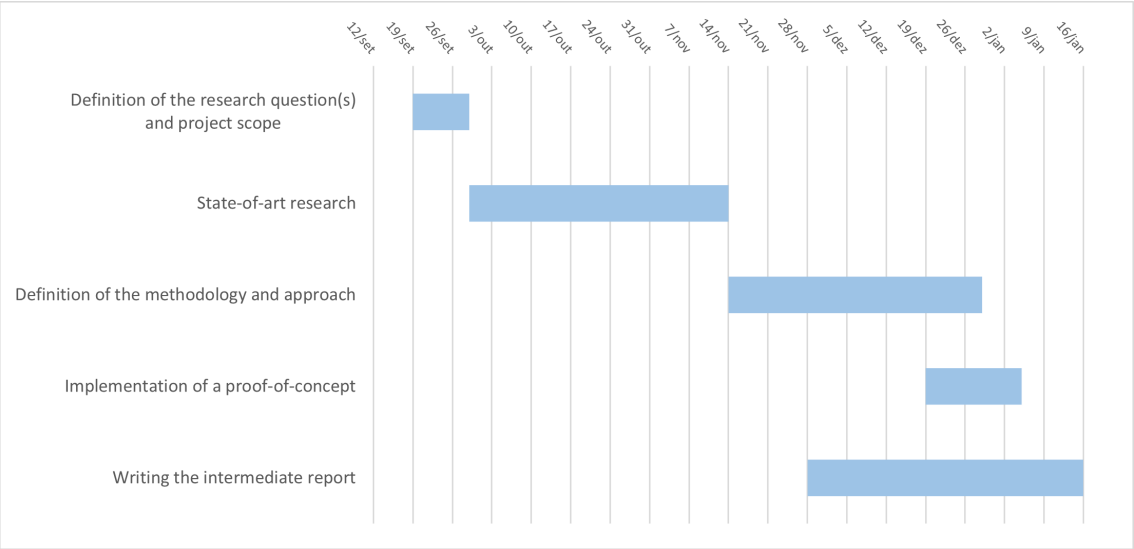


Figure 7.1: Work done in the first semester by weeks

7.2 Second Semester

In the second semester, we started by analyzing the project Greenstamp proposals. Greenstamp aim is to develop a promising approach for classifying mobile apps accordingly to their energy consumption profile. Then we redefined and improved the first-semester strategies to include the Greenstamp methods.

We built the experimental setup and conducted the experiments. The approaches are well-structured and can be reproduced. We presented essential details such as the mobile device and the apps used.

Finally, we wrote and submitted an article to a Journal.

In Figure 7.2, we show the work performed in the second semester through a Gantt chart; we divided the work into four main steps:

1. **Analyze more energy certification mechanisms:** We start the semester by analyzing the Greenstamp methods and improving our approaches.
2. **Setup of the experimental environment:** We defined the experimental environment and created a repository that contains all the code necessary to reproduce our experiments.
3. **Implementation and evaluation of the proposed adversarial strategies:** We need to validate the existence of adversarial strategies. To that end, we implemented the experimental setup and tested the apps according to our definitions.
4. **Writing and publishing an article:** We wrote and submit an article.
5. **Writing the final report:** To conclude the work plan for the second semester, we wrote the final report.

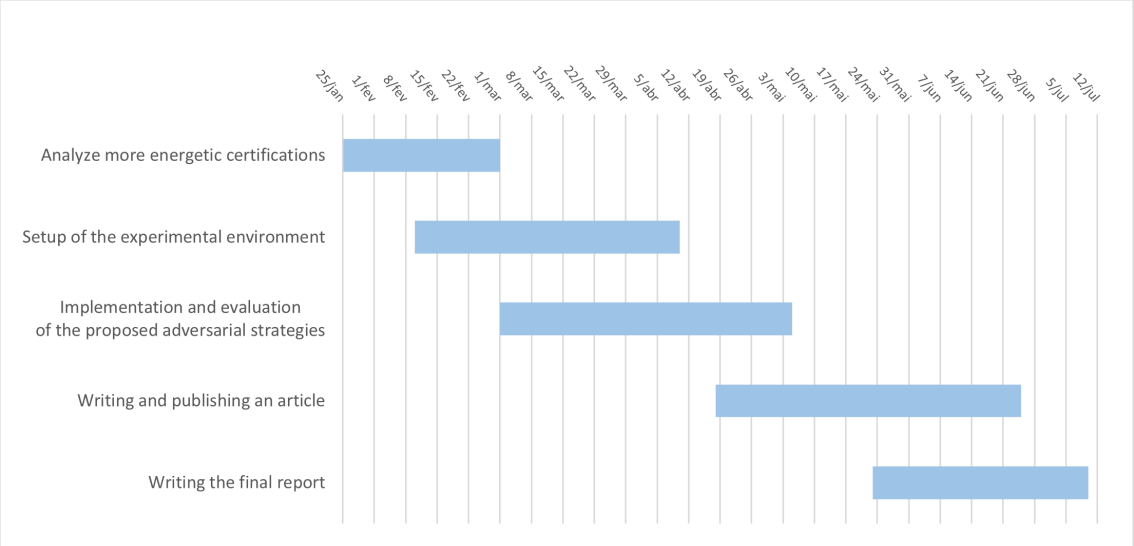


Figure 7.2: Work done in the second semester by weeks

Chapter 8

Conclusion

Energy efficiency is an important consideration when developing mobile applications. However, app stores do not present any energy information about the apps. Energy certification mechanisms for apps will allow users to make more energy-friendly choices. They must be robust to guarantee each app has a fair evaluation because developers may begin to design and develop their products to rank as high as possible artificially. Determining and reporting the mechanisms' susceptibilities, we help the following certifications to be more robust.

With our research, we conclude that vendors develop their products to rank as high as possible. We analysed four cases of vulnerability exploitation: the Diselgate case, where Volkswagen modified its vehicles to pass the emissions test despite emitting an excessive amount of pollutants; the hiring of collusive groups to positively rate a product in online marketplaces; the exploit of a journal importance indicator, the Impact Factor, by publishing review articles that tendentially have more citations, among other techniques; the gap between the EU and Chinese energy certifications for household appliances which completely different evaluation techniques.

First, we established our work objectives and research question. We researched real examples of techniques that influenced evaluations in a rating system. This analysis provided us with the knowledge to identify vulnerabilities in certifications. We then examined proposed energy certification mechanisms for apps, understood their functioning, and identified potential weaknesses developers could exploit.

We designed and implemented the experimental environments of our strategies. Subsequently, we tested the apps modified according to each environment and discussed the results obtained. We analysed the work developed by the project Greenstamp, where we could detect issues. Then, construct our strategies containing experimental scenarios to cover the new susceptibilities. Finally, we wrote and submitted an article.

With the strategies, we validated the existence of the vulnerabilities: The background service consumption is not regarded in the energy certification mechanism, The helper app consumption is not regarded in the energy certification

mechanism, A refactored app is considered energy efficient and The evaluation mechanism cannot deal with threads. The experiences performed in each scenario were conclusive. For example, the refactoring tools gave the same evaluation to two apps with a consumption difference of 1.9532 mAh in a 1-minute test. We have observed that there are ways in which developers manipulate certifications. And if the certifications overcome the issues detected, the developers will discover new ways of defeating the evaluation mechanisms. This implies a constant evolution of the evaluation method to minimize the exploit possibilities, keeping it as reliable as possible,

With our work, we conclude that app developers may: submit adulterated consumption tests, exploit the app's background usage, delegate services to other apps, exploit the energy problems undetected by the refactoring tools, and exploit the thread's usage. Discovering opportunities for developers to enhance their evaluations, we responded to our RQ: *How can developers create adversarial strategies against energy certification mechanisms that would make an app rank higher than it should?*

References

[1] Abdullah Mahmoud Almasri. Google play apps erm: (energy rating model) multi-criteria evaluation model to generate tentative energy ratings for google play store apps, Jan 2021.

[2] Johannes Meier, Marie-Christin Harre, Jan Jelschen, and Andreas Winter. Certifying energy efficiency of android applications. 09 2014.

[3] Reyhaneh Jabbarvand Behrouz, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann. Ecodroid: An approach for energy-based ranking of android apps. In *2015 IEEE/ACM 4th International Workshop on Green and Sustainable Software*, pages 8–14, 2015.

[4] Claas Wilke. *Energy-aware development and labeling for mobile applications*. PhD thesis, Dresden University of Technology, 2014.

[5] Nelson Gregório, João Paulo Fernandes, João Bispo, and Sérgio Medeiros. E-APK: Energy pattern detection in decompiled android applications. In *XXVI Brazilian Symposium on Programming Languages*, New York, NY, USA, October 2022. ACM.

[6] Delcio Kelson. wcec-android. https://github.com/DelcioKelson/wcec-android, 2023.

[7] Wellington Oliveira, Bernardo Moraes, Fernando Castor, and João Paulo Fernandes. Ebserver: Automating resource-usage data collection of android applications. In *MobileSoft*. IEEE/ACM 10th International Conference on Mobile Software Engineering and Systems (MOBILESoft) - Tools and Datasets, 2023.

[8] Quirin Schiermeier. The science behind the volkswagen emissions scandal. *Nature*, September 2015.

[9] Christopher Castille and Andrew Fultz. How does collaborative cheating emerge? a case study of the volkswagen emissions scandal. In *HICSS*, 01 2018.

[10] Luís Cruz. Tools to measure software energy consumption from your computer. http://luiscruz.github.io/2021/07/20/measuring-energy.html, 2021. Blog post.

[11] Luís Miranda da Cruz. Tools and techniques for energy-efficient mobile application development. https://handle/10216/126530, 7 2019.

[12] Lide Zhang, Birjodh Tiwana, Robert P. Dick, Zhiyun Qian, Z. Morley Mao, Zhaoguang Wang, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 105–114, 2010.

[13] Luis Cruz and Rui Abreu. Performance-based guidelines for energy efficient mobile applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 46–57, 2017.

[14] Niels Brouwers, Marco Zuniga, and Koen Langendoen. Neat: A novel energy analysis toolkit for free-roaming smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems*, SenSys '14, page 16–30, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Claas Wilke, Sebastian Richly, Georg Püschel, Christian Piechnick, Sebastian Götz, and Uwe Assmann. Energy labels for mobile applications. 01 2012.

[16] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 103–114, 2017.

[17] Ivano Malavolta, Giuseppe Procaccianti, Paul Noorland, and Petar Vukmirovic. Assessing the impact of service workers on the energy efficiency of progressive web apps. pages 35–45, 05 2017.

[18] Shaiful Chowdhury, Silvia Nardo, Abram Hindle, and Zhen Jiang. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Software Engineering*, 23, 06 2018.

[19] Bush Michel and Hu Bo. Finding the most energy efficient tv in china and in europe: not such an easy job.... eceee, 2013.

[20] Wouter Jong and Vivian van der Linde. Clean diesel and dirty scandal: The echo of volkswagen's dieselgate in an intra-industry setting. *Public Relations Review*, 48(1):102146, 2022.

[21] Moritz Contag, Guo Li, Andre Pawlowski, Felix Domke, Kirill Levchenko, Thorsten Holz, and Stefan Savage. How they did it: An analysis of emission defeat devices in modern automobiles. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, May 2017.

[22] Hao Chen, Daojing He, Sencun Zhu, and Jingshun Yang. Toward detecting collusive ranking manipulation attackers in mobile app markets. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 58–70, New York, NY, USA, 2017. Association for Computing Machinery.

[23] Hengshu Zhu, Hui Xiong, Yong Ge, and Enhong Chen. Ranking fraud detection for mobile apps. In *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management - CIKM '13*, New York, New York, USA, 2013. ACM Press.

[24] Hyun-Kyo Oh, Sang-Wook Kim, Sunju Park, and Ming Zhou. Can you trust online ratings? a mutual reinforcement model for trustworthy online rating systems. *IEEE Trans. Syst. Man Cybern. Syst.*, 45(12):1564–1576, December 2015.

[25] Kai Simons. The misused impact factor. *Science*, 322(5899):165, October 2008.

[26] Mickaël Bourgeois, Holisoa Rajerison, François Guerard, Marie Mougin-Degraef, Jacques Barbet, Nathalie Michel, Michel Cherel, and Alain Faivre-Chauvet. Contribution of [64Cu]-ATSM PET in molecular imaging of tumour hypoxia compared to classical [18F]-MISO–a selected review. *Nucl. Med. Rev. Cent. East. Eur.*, 14(2):90–95, 2011.

[27] Amaia de Ayala and María del Mar Solà. Assessing the EU energy efficiency label for appliances: Issues, potential improvements and challenges. *Energies*, 15(12):4272, June 2022.

[28] Signe Waechter, Bernadette Sütterlin, and Michael Siegrist. The misleading effect of energy efficiency information on perceived energy friendliness of electric goods. *J. Clean. Prod.*, 93:193–202, April 2015.

[29] Katarzyna Stasiuk and Dominika Maison. The influence of new and old energy labels on consumer judgements and decisions about household appliances. *Energies*, 15(4):1260, Feb 2022.

[30] Signe Waechter, Bernadette Sütterlin, and Michael Siegrist. Desired and undesired effects of energy labels–an eye-tracking study. *PLoS One*, 10(7):e0134132, July 2015.

[31] Jim Bowyer, McFarland A., Ed Pepke, Harry Groot, Jacobs M., Erickson G., and Carli Henderson. Your tv and energy consumption. 10 2019.

[32] Jóakim von Kistowski, Klaus-Dieter Lange, Jeremy A. Arnold, Sanjay Sharma, Johann Pais, and Hansfried Block. Measuring and benchmarking power consumption and energy efficiency. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 57–65, New York, NY, USA, 2018. Association for Computing Machinery.

[33] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of intel's xeon phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 389–394, 2013.

[34] Patrick Thomson. Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue*, 19(4):29–41, sep 2021.

[35] Jóakim von Kistowski, Jeremy Arnold, Karl Huppler, Klaus-Dieter Lange, John Henning, and Paul Cao. How to build a benchmark. 02 2015.

[36] Pierre Bourque and R.E. Fairley. *Guide to the Software Engineering Body of Knowledge - SWEBOK V3.0*. 01 2014.

[37] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, 44(12):1176–1206, 2018.

[38] Amina Kammoun, Rim Slama, Hedi Tabia, Tarek Ouni, and Mohmed Abid. Generative adversarial networks for face generation: A survey. *ACM Comput. Surv.*, mar 2022. Just Accepted.

[39] Ana Ribeiro, João F. Ferreira, and Alexandra Mendes. Ecoandroid: An android studio plugin for developing energy-efficient java mobile applications. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 62–69, 2021.

[40] Abderraouf Gattal, Abir Hammache, Nabila Bousbia, and Adel Nassim Henniche. Exploiting the progress of oo refactoring tools with android code smells: Randroid, a plugin for android studio. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1580–1583, New York, NY, USA, 2021. Association for Computing Machinery.

[41] Luis Cruz, Rui Abreu, and Jean-Noel Rouvignac. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, May 2017.

Appendices

Appendix A

Applications Used in the Experiments

App Number	App Name	Source	Category	App ID	Version
1	Challenge1	Own App	No category	com.challenge1	-
2	Fast N Fitness	F-Droid	Sports & Health	com.easyfitness	0.20.4
3	BTC Map	F-Droid	Money	org.btcmap	0.6.1
4	Vector Pinball	F-Droid	Games	org.woheller69.arity	1.34
5	ArityCalc	F-Droid	Science & Education	com.dozingcatsoftware.bouncy	1.12.1
6	omWeather	F-Droid	Internet	org.woheller69.omweather	1.3
7	Qwotable	F-Droid	Reading	com.lijukay.quotesAltDesign	0.2

Table A.1: More information about the apps used in the experiences.