

Green Software in the Large: Energy-driven Techniques, Tools and Repositories

Rui Rua

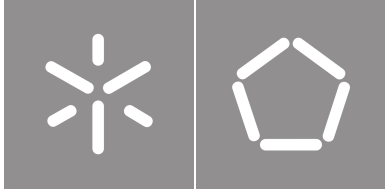
UMinho | year

Universidade do Minho
Escola de Engenharia

Rui António Ramada Rua

Green Software in the Large: Energy-driven Techniques, Tools and Repositories

Junho de 2024



Universidade do Minho
Escola de Engenharia

Rui António Ramada Rua

Green Software in the Large: Energy-driven Techniques, Tools and Repositories

Doctorate Thesis

Doctorate in MAP Doctoral Program in Computer Science (MAP-i)

Work developed under the supervision of:
João Alexandre Batista Saraiva

Cristina Videira Lopes

Junho de 2024

COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

CC BY-NC-SA 4.0

https://creativecommons.org/licenses/by-nc-sa/4.0/deed.en

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.5) (Lourenço, 2021).

Acknowledgements

I wish to express my dearest gratitude to everyone who accompanied me on this journey. A big thank you to my outstanding supervisors, Prof. João Saraiva and Prof. Cristina V. Lopes, and my friends and colleagues Francisco Ribeiro, José Macedo, Marco Couto and Rui Pereira. I also want to thank Prof. Helmut Prendinger and Prof. João P. Fernandes for their valuable help.

To my family, thank you for all the unconditional support and guidance. And a very heartfelt thank you to my amazing, supportive parents, for all the hard work you put into raising me. It ultimately leads me here, and I hope it makes you guys proud. A special thanks to my sisters, for your friendship and encouragement, and for making me proud of you every day. Last, but not least, I extend my gratitude to all of my dearest friends who have been a constant source of encouragement and joy throughout this academic journey.

Finally, this work was partially funded by FCT - Foundation for Science and Technology, the Portuguese Ministry of Science, Technology and Higher Education, through national funds, and co-financed by the European Social Fund (ESF) through the Operational Programme for Human Capital (POCH), with scholarship reference SFRH/BD/146624/2019. Additionally, funding was provided by National Funds through the Portuguese funding agency, FCT, within project UIDP/50014/2020, by COST Action 19135: "CERICIRAS - Connecting Education and Research Communities for an Innovative Resource Aware Society", and by Erasmus+ project No. 2020-1-PT01-KA203-078646: "SusTrainable - Promoting Sustainability as a Fundamental Driver in Software Development Training and Education".

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the Universidade do Minho.

Braga, 22 de Janeiro de 2024

(Place) (Date)



(Rui António Ramada Rua)

“”The object of life is not to be on the side of the majority, but to escape finding oneself in the ranks of the insane””

— Marcus Aurelius, Roman Emperor, philosopher

Resumo

Software Verde em Larga Escala : Técnicas, Ferramentas e Repositórios orientados a energia

Desenvolver código energeticamente eficiente consiste atualmente num exigente desafio devido à falta de familiaridade da comunidade de desenvolvimento com técnicas de otimização e da incerteza da sua aplicabilidade. A escassez de diretrizes de eficiência energética dificulta a tomada de decisões, cruciais para prolongar a autonomia e vida útil de dispositivos móveis.

A complexidade de desenvolver código energeticamente eficiente é destacada nesta tese. O documento com a compilação de uma visão geral de esforços prévios para estimar o consumo de energia de código no ecossistema Android. De seguida, apresentam-se 2 ferramentas que materializaram tecnicas de análise em ferramentas de código aberto que facilitam a análise de desempenho em Android: *E-MANAFA*, uma ferramenta de estimação de energia e *PyAnaDroid*, um fluxo de execução configurável para análise de desempenho de aplicações Android.

De seguida, apresentam-se estudos empíricos que avaliam aspectos relacionados com o desempenho de aplicações Android. Um estudo preliminar evidencia ganhos de energia atráves do uso de técnicas de *memoization* em aplicações, enquanto um estudo em grande escala analisa o impacto de práticas de desenvolvimento e manutenção de código em aplicações reais. Um outro estudo compara teclados virtuais Android, revelando variações significativas no seu desempenho energético.

Os nossos resultados pretendem orientar a comunidade na tomada de decisões e na adoção de estratégias eficientes para desenvolver código com alto desempenho energético. As ferramentas e metodologias apresentadas, posicionam este trabalho como um recurso valioso para alcançar avanços no desenvolvimento de software móvel energeticamente eficiente.

Palavras-chave: Análise de Software, Android, Consumo de Energia, Teste e Manutenção de Software

Abstract

Green Software in the Large: Energy-driven Techniques, Tools and Repositories

Developing energy-efficient software consists in a difficult challenge due to developers' unfamiliarity with energy saving techniques and uncertainty about their applicability. Scarcity of energy efficiency guidelines hampers decision-making, crucial for extending battery lifetime and devices' uptime.

The complexity of developing energy-efficient code is highlighted in this thesis. The exploration begins by compiling an overview of state-of-the-art methodologies for estimating software energy consumption on Android. We start by presenting two distinct toolsets that materialized energy analysis and benchmarking techniques into open-source tools that facilitate energy analysis on the Android ecosystem: *E-MANAFA*, a model-based, device-independent energy profiler for Android devices and *PyAnaDroid*, a configurable execution pipeline that facilitates performance analysis of Android applications.

Following, we present several empirical studies that shed light on various aspects of Android applications' performance. A preliminary study shows potential energy gains with memoization techniques in Android apps, while a large-scale study analyzes the impact of development practices, identifying key factors affecting energy performance. Another study compares Android virtual keyboards, revealing significant performance variations and suggesting potential energy savings.

These results produced can guide developers, researchers, and practitioners in making informed choices and adopting effective strategies for developing energy-efficient code and optimizing performance. The presented tools and methodologies, coupled with the nuanced understanding of energy performance dynamics, position this research as a valuable resource for advancing the state of energy-efficient software development in the context of mobile applications.

Keywords: Android, Energy Consumption, Software Testing and Maintenance, Software Analysis

Contents

List of Figures **xi**

List of Tables **xiv**

List of Algorithms **xvi**

Acronyms **xviii**

1 Introduction **1**

1.1 Problem Statement and Research Goals 5

1.2 Thesis Outline and Origin of Chapters 7

1.3 Contributions 8

1.3.1 Artifacts 8

1.3.2 Results 9

1.3.3 Further Contributions 10

2 State of the Art **13**

2.1 Green Computing 13

2.2 Android Platform 15

2.3 Performance Analysis on Android devices 18

2.3.1 Estimating Energy Consumption on Android devices 19

2.3.2 Automating Performance Analysis On Android 25

2.3.3 Analyzing and Optimizing Performance of Android Software 27

3 Automating Performance Analysis on Android ecosystem **32**

3.1 Introduction . . . 33

3.2 *E-MANAFA* Framework . . . 36

3.2.1 *E-MANAFA* Main Components . . . 37

3.2.2 *E-MANAFA* workflow . . . 40

3.2.3 Preliminary Results . . . 41

3.2.4 Limitations . . . 42

3.3 *PyAnaDroid* Framework . . . 43

3.3.1 *PyAnaDroid* Building Blocks . . . 45

3.3.2 Preliminary Results . . . 51

3.3.3 Data Collection . . . 53

3.3.4 Building Results . . . 55

3.3.5 Execution Results . . . 56

3.3.6 Discussion . . . 59

3.3.7 Threats to Validity . . . 62

3.4 Summary . . . 63

4 Empirical Analysis of native Android source code **65**

4.1 Introduction . . . 66

4.2 Towards Using Memoization for Saving Energy in Android . . . 70

4.2.1 What Can be Memoized? . . . 71

4.2.2 Experimental Setup . . . 72

4.2.3 Results . . . 75

4.2.4 Discussion . . . 81

4.2.5 Threats to Validity . . . 83

4.3 A large-scale empirical study on mobile performance: Energy, Run-time and Memory 84

4.3.1 Methodology . . . 85

4.3.2 Large-Scale Android Performance Analysis . . . 96

4.3.3 Contributions . . . 119

4.3.4 Research Answers and Related Findings . . . 120

4.3.5 Programming Guidelines and other Findings . . . 124

4.3.6 Dataset . . . 128

4.3.7 Threats to Validity . . . 130
4.4 Summary . . . 133

5 Greenspecting Android Virtual Keyboards 136
5.1 Introduction . . . 137
5.2 Studying the Energy Efficiency of Virtual Keyboards . . . 138
5.2.1 Keyboards Under Test . . . 139
5.2.2 Experimental Setup . . . 140
5.2.3 Experimental Procedure . . . 143
5.3 Analysing the Energy Consumption . . . 148
5.3.1 Default and Minimal mode . . . 150
5.3.2 Real User Interaction and Simulation . . . 151
5.4 Discussion . . . 153
5.5 Threats to validity . . . 157
5.6 Summary . . . 159

6 Conclusions 160
6.1 Future Work . . . 162
6.1.1 Refining this Thesis . . . 163
6.1.2 Beyond this Thesis . . . 165

Bibliography 168

Appendices

A Chapter 4 Appendix 186
A.1 Libraries . . . 186
A.1.1 Collections/Data Structures Libraries . . . 186
A.1.2 Logging Libraries . . . 188
A.1.3 Image Loading/Rendering Libraries . . . 189
A.1.4 I/O Libraries . . . 191

List of Figures

1 First Mobile Phone (Motorola DynaTAC 8000X) and his inventor, Martin Cooper 2

2 The five domains of Sustainable Software Engineering (Cruz, 2022) 3

3 Android and Linux Comparison (redrawn from Levin, 2015) 17

4 Standard process of generating a standard Android application distribution package (Android Package (APK)) 19

5 Graphical representation of the Trapezoid rule 20

6 Typical setups for estimating energy performance using energy profilers (at left) and power monitors (at right) 21

7 An illustrative example of several statically-detected issues on an Android project in Android Studio with Lint 30

8 *E-MANAFA* architecture diagram 37

9 E-MANAFA Results Inspector 38

10 Example of a log of a BatteryStats' Battery History section 39

11 Comparison between non-instrumented and instrumented (+I) app versions in terms of runtime and Central Processing Unit (CPU) energy consumption, respectively 42

12 *PyAnaDroid* default workflow 45

13 Distribution of the total energy consumed by each test and distribution of the average energy consumption per function, respectively. 58

14 Distribution of the elapsed time by test and distribution of the average method coverage per test, respectively. 59

15 Distribution of the Java Exceptions and errors occurred during the test sessions, respectively. 60

16 Methods with consumption between 0 and 175 mJ. 76
17 Box plots representing the energy spent by the original and memoized methods, considering 10 executions of the tests, where the memoized version spent less energy. 76
18 Box plots representing the energy spent by the original and memoized methods, considering 10 to 50 executions of the tests, where the original version spent less energy. 77
19 Box plots representing the energy spent by the original and memoized methods, considering 10 to 50 executions of the tests, where no consistent statistical difference among the different runs of tests was found for original and memoized methods. 77
20 Energy losses from the original method to the memoized one of the methods that have positive results. 78
21 Energy losses from the original method to the memoized one of the methods that have negative results. 78
22 Energy losses from the original method to the memoized one of the methods that have no clear positive or negative direction. 79
23 Overview of the execution workflow 89
24 Distribution of number of versions per app 92
25 Distribution of number of app/versions per release year 95
26 Average Energy consumption across apps versions' release years 97
27 Memory and run-time across apps versions' release years 97
28 Filtering process followed to answer **RQ4.3.1** 99
29 Energy consumption of tests with Red Application Programming Interface (API)s 106
30 Energy consumption of tests without Red APIs 107
31 Memory used by tests with Red APIs 109
32 Elapsed time of tests with Red APIs 110
33 Lint Issues energy consumption 111
34 Lint Issues memory consumption 111
35 Lint Issues run-time 112
36 HyperText Transfer Protocol (HTTP) Libraries energy consumption 115
37 HTTP Libraries memory consumption (left) and run-time (right) 116
38 Threading Libraries energy consumption 117
39 Threading Libraries memory and run-time 118

40 JavaScript Object Notation (JSON) Libraries energy consumption 119
41 JSON Libraries memory and run-time 119

42 Statistics of the volunteers used in our study 146
43 Energy consumption box plots of all keyboards 149
44 Elapsed time box plots of all keyboards 149
45 Correlation matrix of the sampled data during tests execution 151
46 Energy consumption box plots of the 2 keyboards tested with real users 152
47 Tests Elapsed time of both keyboards tested with real users 152
48 Energy consumption box plots of the 2 keyboards for the naive automatic procedure . . 153
49 Tests Elapsed time of both keyboards for the naive automatic procedure 154

50 Collections/Data Structures Libraries Energy Consumption 186
51 Collections/Data Structures Libraries Elapsed Time 187
52 Collections/Data Structures Libraries Memory Usage 187
53 Logging Libraries Energy Consumption 188
54 Logging Libraries Elapsed Time 188
55 Logging Libraries Memory Usage 189
56 Image Loading/Rendering Energy Consumption 189
57 Image Loading/Rendering Libraries Elapsed Time 190
58 Image Loading/Rendering Libraries Memory Usage 190
59 I/O Libraries Energy Consumption 191
60 I/O Libraries Elapsed Time 191
61 I/O Libraries Memory Usage 192

List of Tables

1 Language Rankings for Different Optimization Goals *(Pereira et al., 2017)*. 15

2 Examples of Android native frameworks 17

3 Comparison among the state-of-the-art tools for estimating energy in the Android ecosystem 25

4 Testing frameworks supported by *PyAnaDroid* 49

5 Languages found in the 248 analyzed projects 54

6 Average values for several projects' configurations 55

7 Building accuracy of different approaches 56

8 Characterization of methods used in experiments. 73

9 1st (on the left-hand-side of the column) and 3rd (on the right) quartiles of the energy losses from the use of the original method to the use of the memoized version. 80

10 For each method, and for each test setting, we present the percentage (of the 25 runs) that the memoized version consumes less energy than the original one. Underlined values have statistical significance while all the others have; bold values represent more than 50% of the 25 runs spend more energy in the memoized methods than the original ones. 80

11 Auxiliary tools used for Android projects' instrumentation and compilation 88

12 List of permissions that activate sensors 89

13 Analyzed App Categories 95

14 Criteria considered for the manual analysis 100

15 Relation between raises and drops of the 3 performance indicators 101

16 Impact of changes in energy consumption 102

17 Impact of changes in performance on versions with increased functionality 103

18 Impact of changes in performance on versions with *bugfixes* 104
19 Red APIs invoked in Monkey tests 105
20 P-values and resultant Hypothesis of assessing if there were differences in test performance using Red APIs 108
21 Correlation results between performance indicators and Lint issues priority 113
22 Evaluated Libraries 114
23 Most efficient/inefficient libraries **[G3]** 126
24 Metrics gathered during tests/apps execution 129
25 Resume of the different metrics obtained 130

26 GBoard: Languages, number of files and (source code) lines 139
27 KUT features list. E - Enabled by default, D - Disables by default. 140
28 External and internal factors that can impact keyboard performance 142
29 p-values and hypothesis test decision for each of the Keyboard Under test (KUT) with non-parametric samples 156

List of Algorithms

1 Dynamic Execution workflow 90

2 Text input simulation algorithm 145

3 Real User Text input algorithm 148

4 Naive text input algorithm 148

Code Listings

3.1 Default Test execution process of PyAnaDroid 49
4.1 Example of memoization in a recursive function in Python 71

Acronyms

AAPT Android Asset Packaging Tool

ADB Android Debug Bridge

ANR App Not Responding

AOT Ahead Of Time

API Application Programming Interface

APK Android Package

ART Android RunTime

AST Abstract Syntax Tree

AUT App Under Test

CLBG Computer Language Benchmark Game

CLI Command Line Interface

CPU Central Processing Unit

CRUD Create Read Update Delete

CSV Comma-Separated Values

DB Database

DUT Device Under test

GPS Global Positioning System

GPU Graphics Processing Unit

GSMCA Global System for Mobile Communications Association

GUI Graphical User Interface

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

ICT Information and Communication Technology

IDE Integrated Development Environment

IPC Inter-Process Communication

IT Information Technology

JAR Java ARchive

JIT Just In Time

JNI Java Native Interface

JS JavaScript

JSON JavaScript Object Notation

JVM Java Virtual Machine

KUT Keyboard Under test

LCD Liquid Crystal Display

LLM Large Language Model

MB MegaBytes

NDK Native Development Kit

NFC Near Field Communication

OS Operative System

PMIC Power Management Integrated Circuit

POC Proof Of Concept

PSS Proportional Set Size

RAD Rapid Appication Development

RAM Random Access Memory

REST REpresentational State Transfer

RPC Remote Process Communication

RSSI Received Signal Strength Indicator

SDG Sustainable Development Goal

SDK Standard Development Kit

SIM Subscriber Identity Module

SoC System On Chip

SQL Structured Query Language

UI User Interface

URL Uniform Resource Locator

UX User EXperience

XML EXtensible Markup Language

1

Introduction

Within the landscape of technological progress, few innovations have become as deeply ingrained in human life as mobile devices. Among them, smartphones stand out as a definitive representation. Over a few decades, these compact gadgets have transformed not only how we communicate but also our lifestyles, work, and engagement with the world. In the evolving digital era, smartphones have played a pivotal role in driving substantial socio-cultural shifts, reshaping our daily habits, enhancing our connectivity, and broadening our potential beyond previous boundaries.

The evolution of mobile devices is a captivating narrative of innovation and accelerated progress. The story begins with the advent of the first commercialized handheld mobile phone, the Motorola DynaTAC 8000X, in 1983, often affectionately referred to as the "brick phone" (Figure 1). This monumental achievement marked the dawn of mobile telephony, although it was a far cry from the sleek, feature-rich smartphones that we carry today. The cellular phone, in its earliest incarnation, was a somewhat impractical expensive symbol of prestige.

The subsequent years witnessed a monumental avalanche of developments, gradually transforming mobile devices into multi-functional powerhouses. The once-astronomical prices of cell phones plummeted, and their size decreased significantly. Gradually, these devices morphed into portable computing hubs, incorporating a myriad of features, from texting and web browsing to Global Positioning System (GPS) navigation and multimedia consumption. The advent of the iPhone in 2007 and the proliferation of Android-based devices further accelerated the smartphone revolution, pushing the boundaries of what these devices could do.

Today, smartphones have become inseparable companions in our daily lives. They serve as our communication lifelines, personal organizers, entertainment hubs, and gateways to the digital realm. With the rise of mobile applications, these devices have transformed into Swiss Army knives of the digital



Figure 1: First Mobile Phone (Motorola DynaTAC 8000X) and his inventor, Martin Cooper

age, allowing us to access a wide array of services and functions with a few taps of our fingers. This technological symbiosis has blurred the line between the virtual and the physical, reshaping the way we navigate our world, communicate, and even conduct business.

As the importance of mobile devices in our lives has grown, so too has the challenge of managing and leveraging their energy consumption. Despite the remarkable strides in smartphone technology, its main limitation is yet to be overcome: its limited energy capacity. This challenge is not only a matter of convenience but also one of environmental and economic significance. The increasing demand for energy-efficient mobile devices is driven by a complex set of factors, from environmental concerns to user expectations and business interests.

Currently, we observe efforts emerging from all economic sectors in order to reinvent their businesses to become environmentally responsible. The same concern is growing in the Information Technology (IT) sector. Tech organizations are starting initiatives to be carbon-free in the near future. For example, big companies such as Google have been publicly expressing their intent to be carbon-free by 2030 (Google, 2021). More concretely, when talking about software, a definition for Sustainable Software Engineering was proposed by the Karlskrona Manifesto (Becker et al., 2015), which encompasses 5 distinct domains, illustrated in Figure 2.

This thesis is mostly focused on the technical and environmental domains of Sustainable Software Engineering. The technical domain approaches the longevity of information, systems, and infrastructure

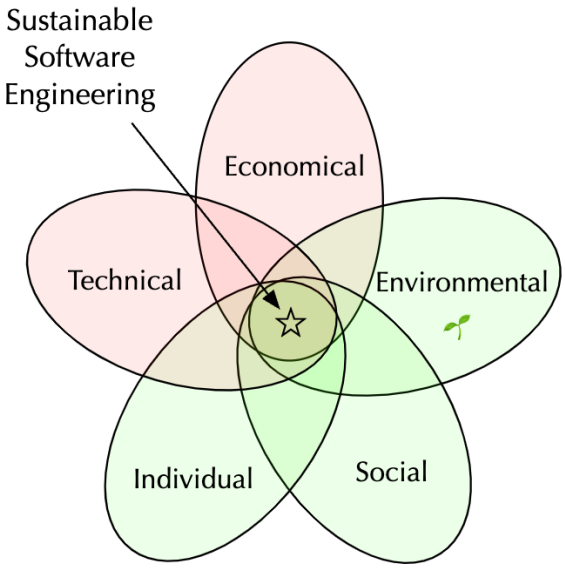


Figure 2: The five domains of Sustainable Software Engineering (Cruz, 2022)

and their adequate evolution with changing surrounding conditions. It includes maintenance, innovation, data integrity, among others. The environmental domain is concerned with the long-term effects of human activities on natural systems. This dimension includes natural resources, climate change, etc. Furthermore, this thesis is an effort to extend the Green Software field, which focuses on reducing energy consumption through software analysis and optimization. This branch of Software Engineering analyses the complete software development lifecycle, analyzing the sustainability of software on many different levels: the impact of developing, using, maintaining, scaling, etc.

The significance of Green Software analysis becomes even more apparent when one considers the ubiquity of smartphones in today's world. The Global System for Mobile Communications Association (GSMCA) estimated that by 2025, there would be 5.9 billion unique mobile subscribers worldwide, representing nearly 70% of the global population. This widespread adoption of smartphones underscores their undeniable importance in our daily lives. In the context of these numbers, it is imperative to address the issue of smartphone energy consumption for several reasons:

- **Environmental Sustainability**: The manufacturing, use, and disposal of electronic devices contribute significantly to carbon emissions and electronic waste. Energy-efficient smartphones can help mitigate this environmental impact in several ways, by diminishing energy consumption during their use and possibly extending the devices' lifetime.

- **User EXperience (UX)**: Prolonged battery life and efficient energy use enhance the user experience. Users expect their devices to last through a full day of usage without interruptions, and addressing this expectation is a key challenge.
- **Economic Viability**: The economic implications of smartphone energy consumption are profound. Energy-efficient devices can reduce electricity bills and extend the lifespan of devices, ultimately saving consumers money.
- **Technological Advancements**: By optimizing energy use, it is possible to create more advanced features and functionalities without compromising battery life, driving innovation in the smartphone industry.

This thesis is dedicated to exploring the multifaceted aspects of energy consumption in mobile devices, with a specific focus on Android smartphones, their software and their technological progress. This journey will traverse the intricate landscape of Android software development, energy-hungry software, and strategies to enhance energy efficiency. The inquiry will delve into the challenges and opportunities presented by smartphones' energy consumption, paving the way for the development of innovative solutions to minimize waste and maximize utility. To tackle these challenges and harness the opportunities that energy efficiency in smartphones presents, this thesis will investigate a range of key areas:

- **Tools and techniques**: The thesis will endeavor to propose innovative strategies and technologies for measuring and enhancing mobile software energy efficiency, considering solutions purely based on software.
- **Energy and Performance Software Patterns**: Certain coding practices are known to have certain expected performance behavior when executed in certain execution contexts. Also, many of these practices have alternative implementations that can provide the same functionality while providing different performance behavior. In this thesis, we evaluate the impact of previously cataloged performance patterns on open-source software and we aim to contribute to state-of-the-art knowledge by identifying new performance patterns.
- **Energy-Intensive Applications**: Certain applications and services, such as augmented reality, gaming, and video streaming, consume a disproportionate amount of energy. Since there is a wide range of available alternatives for users to choose from, we provide comparisons and insights about the energy footprint of some of the most widely used alternatives. The way applications are

used by their users strongly affects other energy consumed by them, examining and emulating user behavior is essential for creating energy-efficient solutions that align with user expectations. Furthermore, in the scope of this thesis, we provide several insights about the performance footprint of well-known Android applications executed in realistic execution scenarios.

1.1 Problem Statement and Research Goals

The journey to manage and optimize smartphones' energy consumption is fraught with complexities, requiring a comprehensive examination of the interplay between hardware, software, and user behavior. The Android ecosystem features a multitude of manufacturers, each offering unique hardware components and software variations, resulting in an expansive diversity of screen sizes, CPU architectures, and operating system versions. Moreover, the heterogeneous nature of Android devices, with heterogeneous hardware and software and consequently, power management strategies, introduces significant variations in energy consumption across devices. This complexity is compounded by the existence of thousands of different Android devices, each with its specific power consumption characteristics. Also, mobile software consists of multiple layers, including the operating system, system services, and third-party applications, each with its own energy profile. Consequently, the task of accurately estimating the energy consumption on the Android platform is a formidable challenge.

Energy efficiency is not solely a matter of extending battery life but has profound implications for the environment and the overall sustainability of the digital ecosystem. The concept of energy efficiency, as discussed previously, has become a vital non-functional software requirement in the era of smartphones. Instead of approaching energy efficiency solely as a performance requirement, a more sophisticated perspective is necessary. Attempting to model energy consumption based solely on CPU cycles and basic metrics is too simplistic, given the multitude of underlying subcomponents within modern smartphones, such as location sensors, haptic feedback engines, GPUs, various heterogeneous CPUs, and network connections. Existing tools often fall short of providing a holistic view of energy usage, and their effectiveness in real-world scenarios is limited. Consequently, this thesis is dedicated to addressing a main research question, which is addressed by 3 primary research goals, that define more concrete objectives to address the broader research question. The main research question is the following:

Main Research Question

What are the current limitations of the state-of-the-art approaches to improving the energy performance of the Android ecosystem, and how can these limitations be overcome by the development and scientific community?

While energy efficiency and runtime performance optimization are often considered separately, they are intricately linked. Some runtime improvements can lead to increased energy consumption, and vice-versa. If also considering other performance indications such as memory usage or bandwidth, the complexity of the task of finding an adequate solution greatly increases. Balancing several performance indicators is a delicate task, and there is a need for tools that can assist developers in making informed decisions to strike the right balance. Furthermore, motivated by the need to define tools and techniques that facilitate performance analysis,nthis thesis incorporates the following research goal:

Research Goal 1

Develop novel tools that provide and facilitate integrated energy and performance analysis for Android software (Chapter 3).

Various efforts from the scientific community to provide knowledge regarding the efficiency of mobile apps and programming practices tend to be materialized in the form of empirical studies. These studies typically use similar empirical/testing procedures (Choudhary et al., 2015; Kong et al., 2019a), sharing tools, procedures, and even knowledge. However, the entire procedures tend to be done from scratch whenever a new study is carried out, due to applicability or availability limitations of the procedure followed in previous works. Given these limitations and the absence of standard and easily replicable procedures to perform performance analysis of Android source code, this thesis also considers the following research goal:

Research Goal 2

Propose procedures and guidelines to facilitate the replication and validation of performance analysis (Chapter 3).

Performance-related code changes may not always be straightforward and may impact various aspects of the app's codebase. Furthermore, our work aims not only to provide guidelines and artifacts

to facilitate performance analysis but also to contribute to the state-of-the-art knowledge base by assessing identifying and evaluating performance patterns that are intrinsic to typical development scenarios of Android development. Furthermore, our work targets an additional research goal:

Research Goal 3

Study and document best practices and recurrent solutions that can be applied to enhance the efficiency of mobile applications and devices(Chapters 4 and 5).

1.2 Thesis Outline and Origin of Chapters

In this section, we present the outline for the rest of this thesis, presenting and motivating the genesis of each Chapter. We start by presenting, in Chapter 2 a thorough analysis and discussion of the state of the art directly related to these thesis research topics. The 3 chapters following the state-of-the-art contain a fully detailed description of this thesis's core contributions, with the contents of such chapters being published in peer-reviewed journals and conferences. Afterward, we summarize the contents of each one, while also indicating the publication(s) that originated them.

Chapter 3 describes the main artifacts developed in the scope of this thesis, *E-MANAFA* and *PyAnaDroid*, which consist of the materialization of techniques that are the foundations of the empirical studies presented in the subsequent chapters:



"E-MANAFA: Energy Monitoring and ANAlysis tool For Android"
Rui Rua, João Saraiva
In: 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)
pages: ***1 – 4*** *| DOI:* ***10.1145/3551349.3561342***



"PyAnaDroid: A fully-customizable execution pipeline for benchmarking Android Applications"
Rui Rua, João Saraiva
In: 39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023)
pages: ***586 – 591*** *| DOI:* ***10.1109/ICSME58846.2023.00077***

Chapter 4 presents two distinct empirical studies that analyzed the impact of several programming practices on the energy performance of Android devices. The former consists of a preliminary evaluation of the impact of using memorization for saving energy in Android apps. The latter consists of a large-scale study that automated the performance analysis of a large set of open-source Android apps to evaluate the

performance footprint of programming practices and patterns. These studies are based on the following publications:



"Towards Using Memoization for Saving Energy in Android"
Rui Rua, Marco Couto, Adriano Pinto, Jácome Cunha, João Saraiva
In: 22nd Ibero-American Conference on Software Engineering (CIbSE 2019)
pages: ***279 – 292*** *| DOI:* ***N/A***



"A large-scale empirical study on mobile performance: Energy, Runtime and Memory"
Rui Rua, João Saraiva
In: Empirical Software Engineering (EMSE), 2023
pages: ***67 pages*** *| DOI:* ***10.1007/s10664-023-10391-y***

Chapter 5 presents an empirical study that compared the energy performance of the most widely-used virtual keyboards in the Android ecosystem. This study is based on the following conference paper:



"Greenspecting Android Virtual Keyboards"
Rui Rua, Tiago Fraga, Marco Couto, João Saraiva
In: 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft 2020)
pages: ***98 –- 108*** *| DOI:* ***10.1145/3387905.3388600***

Finally, Chapter 6 presents a discussion of the final conclusions and future work directions.

All the Uniform Resource Locator (URL) references in this document have been saved in an online internet archive. In case a URL is not available, it can be accessed from http://web.archive.org.

1.3 Contributions

Besides the contributions materialized in the form of research papers that originated Chapters 3, 4 and 5, the work developed in the scope of this thesis produced several other relevant outputs that are worth mentioning. Either in the form of publications, artifacts, or results/knowledge, this section catalogs these contributions according to their type.

1.3.1 Artifacts

In the scope of this thesis, we conceived a set of artifacts to contribute to the current state-of-the-art performance and energy analysis to help developers and researchers analyze Android applications' performance

and respective source code:

- **E-Manafa**: a device-independent, plug-and-play software model-based tool for performing fine-grained estimates of energy consumption on Android devices. E-Manafa is able to estimate system-level and per-component energy consumption and when combined with source code instrumentation, is also able to estimate method-level energy consumption.

- **E-Manafa Results Inspector**: An open-source data visualization Graphical User Interface (GUI) to visualize and analyze results from *E-MANAFA* profiling sessions.

- **Pyanadroid**: An execution pipeline for automating the process of analyzing and benchmarking Android applications' energy consumption. PyAnaDroid can be configured to use different testing frameworks, energy profilers, and software analysis tools in its execution pipeline, being able to perform automatic instrumentation and building of Android projects and respective source code.

- **JInst**: A tool for performing source code instrumentation of both Java and Kotlin source code. It is able to perform several types of source code instrumentation, by instrumentation unit tests and/or source code using instruction injection or method-level annotations.

- **Greensource**: GreenSource is an open-access repository, created to store and manage metrics and metadata gathered through static and dynamic analysis performed over Android projects and applications. As such, the GreenSource contains a database within, providing a RESTful API, which enables a uniform form of communication that provides the ability to perform Create Read Update Delete (CRUD) operations through HTTP requests.

1.3.2 Results

In the scope of this thesis, we conducted many studies that involved applying different performance analysis methodologies over various forms of software. From the studies and respective analyses performed, we were able to extract several results that can serve as material and guidance for developers and practitioners, in order to contribute to the development and maintenance of a more sustainable mobile software. Furthermore, we present as a contribution of the developed work the following results:

- A preliminary overview of the expected impact on the energy performance of using memoization in Android source code.

- The largest empirical study of the literature involving dynamic analysis of Android applications.
- A curated dataset obtained from the analysis of 1322 versions of 215 open-source apps. This dataset contains static and dynamic metrics obtained from the analysis of these apps, that can be used for further analysis and studies. This dataset is presented and characterized in more detail in Section 4.3.6.
- A comprehensive investigation of the relationship between Android Lint Performance Issues' severity and the corresponding performance impact.
- Empirical validation of previously defined performance-based best practices to improve the energy efficiency of Android apps.
- A rank of the most efficient, widely-used open-source virtual keyboards and respective configurations. These results can be used by Android users in order to save energy and extend the device's battery lifetime and uptime. The analysis followed to draw these conclusions is documented in detail in Section 5.

1.3.3 Further Contributions

The primary focus of this thesis revolves around contributions directly aligned with its core objective: providing methodologies for conducting performance analysis and decision support for energy-saving practices in Android software development. These contributions, extensively elucidated in subsequent chapters, are dedicated to furnishing methodologies and tools that facilitate the analysis and evolution of software, particularly when the central concern is the reduction of energy consumption over time.

It's important to note that this thesis operates within the broader research domain of Energy Analysis in Software, often referred to as Green Computing, which focuses on reducing energy consumption through software analysis and optimization. During the period in which this thesis was developed, our engagement extended beyond this work as we actively contributed to various research endeavors in this field, from conceiving initial concepts to active involvement in the associated studies and article writing. Each of these individual projects had unique scopes and objectives. In the following sections, we will provide brief descriptions of these distinct projects, along with references to the publications stemming from them.

Energy Efficiency of Programming Languages

Programming languages possess distinctive characteristics, ranging from their grammatical structures to

their programming and execution paradigms, all of which can influence the performance attributes of the software developed with them, including execution time and memory management. Given this premise, we embarked on a study that compared the energy efficiency of different programming languages. We used a set of benchmark programs derived from the Computer Language Benchmark Game (CLBG) as subjects for this comparative analysis. These benchmarks, which spanned a range of general-purpose languages like C, Python, and Java, allowed us to gain insights into how different languages and programming paradigms can impact energy consumption. This research venture shed light on the energy implications associated with language choice and execution paradigms.

This work resulted in the following two publications:

"Energy Efficiency Across Programming Languages: How do Energy, Time, and Memory Relate?",
Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João P. Fernandes, João Saraiva
in *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE 2017).*
pages: ***256 – 267*** *| DOI:* ***10.1145/3136014.3136031***

"Functional Going Green: An Empirical Evaluation of Functional Languages Performance",
José Nuno Macedo, Francisco Ribeiro, Rui Rua, Marco Couto, Jácome Cunha, João Paulo Fernandes, João Saraiva, Rui Pereira
in *Lecture Notes in Computer Science*
(to be published)
pages: ***28***

Energy Efficiency of Web Browsers in the Android Ecosystem

With regard to mobile platforms, it has become evident that different software development techniques have different impacts on energy consumption. End users of these devices have been showing concerns regarding energy-related issues, rejecting applications they identify as energy-greedy, and looking for ways to increase uptime on their devices. Being a web browser one of the most used software applications, it is very relevant to understand which browser is the most energy-efficient. In order to properly compare the web browsers in terms of energy consumption, we conducted a study where we defined a set of typical usage scenarios to be replicated in the different browsers, executed in the same testing environment and conditions. The results of our study show that there are significant differences in terms of energy consumption among the considered browsers.

This work resulted in the following publication:

"Energy Efficiency of Web Browsers in the Android Ecosystem",
Nélson Gonçalves, Rui Rua, Jácome Cunha, Rui Pereira, João Saraiva
in *Workshop on Resource AWareness of Systems and Society (RAW)*
pages: ***9 pages*** *| DOI:* ***10.48550/arXiv.2205.11399***

Collecting and sharing Performance Analysis

When the objective is to identify energy-intensive patterns or code within Android applications, researchers are frequently compelled to create and execute energy tests, a process that can undoubtedly be both resource-intensive and time-consuming.

The mission of the "GreenSource" repository is to alleviate the necessity of crafting energy tests from scratch. This repository harnesses a vast compendium of open-source Android applications as its testing subjects, featuring an extensive aggregation of test metrics. These metrics have been garnered through the execution of multiple energy tests on each application, employing different Android devices, testing frameworks, or energy profilers.

The encompassed metrics encompass a wide array of parameters, spanning from energy consumption to a comprehensive log of method calls, sensor utilization, and even CPU load. This trove of data serves as a valuable resource for information retrieval endeavors, enabling the establishment of connections between energy profiles and the source code or operational behavior of applications.

The development and the preliminary populating of the infrastructure developed to store and share software and its analysis in the repository resulted in the publication of the following work:

"GreenSource: a Large-scale Collection of Android Code, Tests and Energy Metrics",
Rui Rua, Marco Couto, João Saraiva
in *IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019)*
pages: ***176 – 175*** *| DOI:* ***10.1109/MSR.2019.00035***

2

State of the Art

Over the past decade, the realm of energy analysis in software has evolved into a vast and highly productive research domain. This transformation has been driven by the widespread adoption of battery-powered devices across various sectors. As developers increasingly focus on energy-related considerations, researchers have responded with a plethora of techniques, tools, and methodologies aimed at fostering sustainable, energy-conscious software development. The central concerns in this field revolve around the accurate profiling and prediction of energy consumption, strategies for reducing energy usage through software enhancements, and the provision of concrete feedback to practitioners and developers.

This chapter will describe the current state of the art of the scientific field where this dissertation is inserted. It presents a comprehensive overview of several research endeavors that pertain to the analysis and optimization of energy in software. These research initiatives encompass a diverse array of topics, occasionally featuring overlapping concepts and methodologies. Consequently, we have structured this section by introducing the main related topics in distinct sub-sections. Furthermore, Section 2.1 introduces the Green Computing area and its fundamentals. Section 2.2 introduces the core concepts of the Android platform, providing a brief overview of the platform architecture and of the process of development of Android native applications. Section 2.3 extensively describes the current state-of-the-art of (energy) performance analysis on Android.

2.1 Green Computing

The sustainable use of energy resources is no longer a theoretical concept; it is a pressing necessity. The fossil fuels that power our world are finite, and their consumption contributes to climate change. In an ever-evolving world driven by technology, the intersection of global sustainability and cutting-edge

innovation is gaining increasing importance. The Information and Communication Technology (ICT) sector itself is responsible for up to 10% of the world's total energy consumption, and its footprint is expected to increase by at least 50% until 2030 (The Shift Project, 2019). In this context, energy-efficient technology has become a linchpin in the global effort to mitigate these challenges. The energy consumption of mobile devices, and particularly smartphones, holds a pivotal role in this narrative.

The United Nations' Sustainable Development Goals (SDGs) (United Nations & Development, 2015) stand as a universal call to action, rallying nations and organizations to address pressing global challenges, such as climate change and environmental degradation. In parallel, the concept of "Green Software" has emerged as a pivotal force in harnessing the power of software and digital technology to support these ambitious sustainable objectives. Green Software, with its focus on environmental responsibility and resource efficiency, not only aligns with the broader sustainability agenda but also acts as a catalyst, enabling organizations and individuals to contribute to the achievement of the Sustainable Development Goal (SDG)s through software-driven innovation.

In this century, the focus on computer performance is changing: a program's execution time and memory usage are no longer the sole concerns when discussing performance. The fast adoption of non-wired computer devices and the construction of big data centers in recent years is making energy consumption one of the main bottlenecks when building powerful computers (Schlachter, 2013; Theis & Wong, 2017) and software (Pinto & Castor, 2017). More concretely, mobile devices have been increasing their complexity and the number of computing resources, such as CPU, Graphics Processing Unit (GPU), Bluetooth, and camera, among others. As a result of this increase, the battery of these devices becomes a resource that, due to its limited capacity, requires more efficient management. In this setup, energy efficiency plays a crucial role not only in mitigating environmental impact but also in user satisfaction, by reducing battery consumption, extending device lifespan or enhancing user experience.

As a result of the increased concern, in recent years there has been a growing demand in software engineering for energy-aware development methodologies and approaches (Pinto & Castor, 2017; Pinto et al., 2014a). Developers are still unaware of ways to write, maintain and develop software systems with energy efficiency as a concern (Pinto & Castor, 2017). In order to respond to this problem, the scientific community has been designing techniques and tools to estimate the energy efficiency of software artifacts and components (Couto et al., 2014; Nucci et al., 2017b; Qualcomm Innovation Center, 2016). In addition to developing energy monitoring techniques, software engineers have been using software analysis and testing techniques to demonstrate different energy impacts for different software development

approaches, such as data structures (Oliveira et al., 2019; Pereira et al., 2016, 2018), libraries (Linares-Vásquez et al., 2014; Pathak et al., 2011), patterns (Cruz & Abreu, 2017; Cruz & Abreu, 2018; Li & Halfond, 2014; Li et al., 2016; Linares-Vásquez et al., 2014; Morales et al., 2018), frameworks (Corbalan et al., 2018b; Oliveira et al., 2017a) and languages (Chen & Zong, 2016; Couto et al., 2017b; Oliveira et al., 2017b; Pereira et al., 2017). All the efforts made by practitioners and researchers have contributed to understanding how to develop energy-efficient software and have produced significant results that show that it is possible to significantly improve energy consumption through design alternatives. For instance, Agosta et al. (Agosta et al., 2012) defined which Java methods can be memoized based on their functional purity. They established a theoretical model to predict the effectiveness of memoization in terms of energy consumption on a desktop setup. A recent research effort (Pereira et al., 2021a) compared the performance of a wide range of popular programming languages, according to 3 different indicators (runtime, memory usage and energy) and ranked the selected languages according to their performance (one of those rankings is presented at Table 1). The study evidenced that is possible to significantly reduce software energy consumption just by migrating to energy-efficient languages (for instance, their results demonstrated that, in their empirical evaluation Python consumed 78 times more energy than C).

Table 1: Language Rankings for Different Optimization Goals
(Pereira et al., 2017).

Time & Memory	Energy & Time	Energy & Memory	Energy & Time & Memory
C • Pascal • Go	C	C • Pascal	C • Pascal • Go
Rust • C++ • Fortran	Rust	Rust • C++ • Fortran • Go	Rust • C++ • Fortran
Ada	C++	Ada	Ada
Java • Chapel • Lisp • OCaml	Ada	Java • Chapel • Lisp	Java • Chapel • Lisp • OCaml
Haskell • C#	Java	OCaml • Swift • Haskell	Swift • Haskell • C#
Swift • PHP	Pascal • Chapel	C# • PHP	Dart • F# • Racket • Hack • PHP
F# • Racket • Hack • Python	Lisp • OCaml • Go	Dart • F# • Racket • Hack • Python	JavaScript • Ruby • Python
JavaScript • Ruby	Fortran • Haskell • C#	JavaScript • Ruby	TypeScript • Erlang
Dart • TypeScript • Erlang	Swift	TypeScript	Lua • JRuby • Perl
JRuby • Perl	Dart • F#	Erlang • Lua • Perl	
Lua	JavaScript	JRuby	
	Racket		
	TypeScript • Hack		
	PHP		
	Erlang		
	Lua • JRuby		
	Ruby		

2.2 Android Platform

The proliferation of Android-based mobile devices has marked a monumental shift in the digital landscape, with the Android operating system reigning as the most widely adopted mobile platform globally. Given

Android's dominance in the mobile market, it acts as an emblematic representation of the complexities and challenges that developers encounter when striving to create high-performing, user-friendly applications that cater to a diverse user base. As the Android ecosystem continues to expand, so too does the complexity and heterogeneity within it. This rich tapestry of diversity encompasses a multitude of device manufacturers, hardware configurations, and Android platform versions, making it a formidable challenge for app developers and a critical area for research. The need to study and analyze Android apps' behavior is underscored by this extraordinary heterogeneity, which poses a profound impact on the user experience and the efficacy of mobile applications.

In this section, we describe the pillars of the Android Infrastructure, as well as present the current state of the Android Development paradigm. Furthermore, we delve into the foundations of the Android Operating System and its applications, allowing us to understand its main components. Finally, we describe the current state of Android Development, by presenting methodologies, tools and approaches used by the community to construct on top and support the Android ecosystem.

While Android is rooted in Linux and extensively utilizes its infrastructure, particularly the kernel, it has evolved into a distinct operating system (in contrast to OS X and iOS, which have substantial code commonalities, apart from the user interface and certain frameworks). Android brings forth an extensive array of frameworks along with a supporting runtime (Dalvik, and more recently, Android RunTime (ART)) (Figure 3), delivering a multi-layered structure. Android goes beyond providing just the fundamental kernel and shell binaries, since it offers a self-contained GUI environment and a robust collection of frameworks. Complementing this a the user-friendly development language, Java (and more recently, Kotlin), which equips developers with a Rapid Appication Development (RAD). This means that developers can tap into pre-existing, thoroughly tested code within the frameworks to access advanced functionalities like cameras, motion sensors, GUI widgets, and more, all with just a few lines of code.

In this thesis, we will focus mostly on the Applications and Frameworks layers of the Android stack, since our main goal is helping developers to build energy-efficient applications. Furthermore, we will focus on components that developers and researchers can easily modify without performing foundational changes to the operative system itself.

Android frameworks are divided into separate namespaces, according to their functionality. Packages in the android.* namespace are available for use by developers. Packages in com.android.* are internal and reserved for operating system and Android-specific libraries. Android also supports most of the standard Java runtime packages in the java.* namespace. The entire set of frameworks is bundled

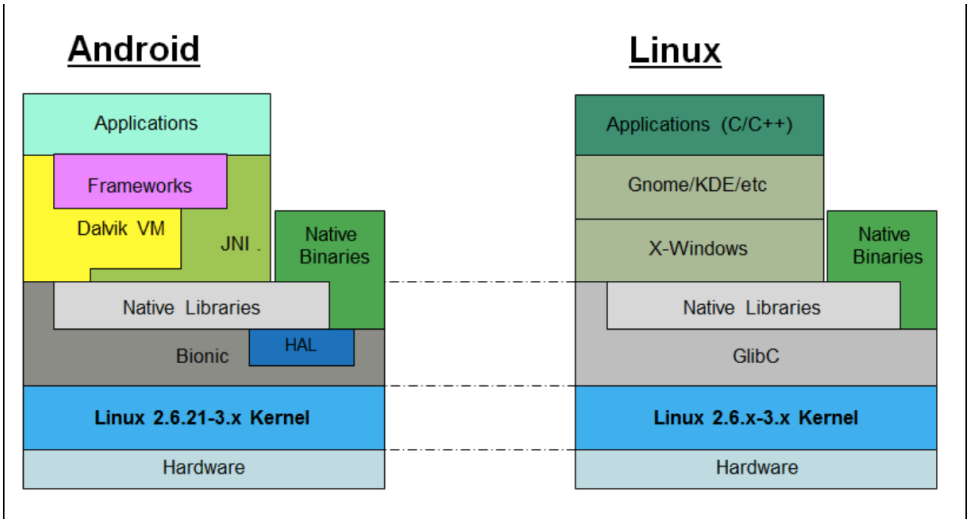


Figure 3: Android and Linux Comparison (redrawn from Levin, 2015)

Table 2: Examples of Android native frameworks

Package	Description
android.hardware	Camera, input and sensor support
android.location	Location support
android.net	Network support
android.database	Database support (SQLite)
android.view	UI Components

into several Java ARchive (JAR) files on the device, in the /system/framework directory. Other popular frameworks such as Android Jetpack (androidx.* namespace), present in over 90% of the apps present on Play Store (AppBrain, 2020a), are shipped separately from the Android platform.

Native Android apps can be written with Kotlin or Java, while support for C/C++ and Rust is also provided via NDK. The Android SDK tools compile the code along with any data and resource files into a container: APK or Android App Bundle. Android applications are constructed around 4 primary components, each serving a distinct purpose:

- **Activities**: Visible elements that users interact with. They handle tasks like displaying content, or receiving user input. Apps can have multiple activities, but one of them usually serves as the main entry point.

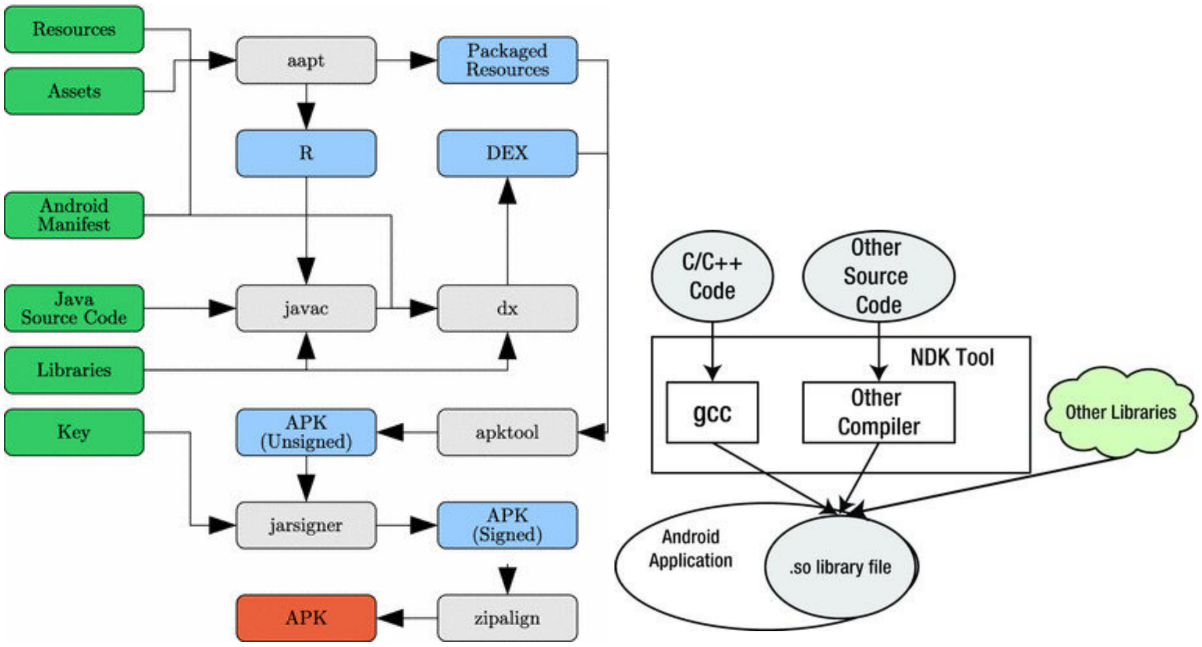
- **Services**: Services are the behind-the-scenes workers of Android apps. Common use cases for services include handling network requests, playing music, or periodically updating data.

- **Broadcast Receivers**: Event listeners for Android apps. They enable apps to respond to system-wide events or broadcasts, such as a low battery notification or incoming SMS message.
- **Content providers:** facilitate data management and sharing between apps. They serve as an interface for accessing and managing structured data, such as calendar events or other app-specific data.

To convert Android app source code into the standard distribution bundle, an APK, several essential tools and processes are involved, which are illustrated in Figure 4. The Android Asset Packaging Tool (AAPT) compiles and processes resources such as EXtensible Markup Language (XML) files, images, and layouts into a binary format. The Java Compiler (javac) is used to compile Java/Kotlin source code into Java bytecode. The Dalvik Executable (dex) tool converts the compiled bytecode into Android-VM-specific-bytecode, Dalvik bytecode. If the project contains Native Development Kit (NDK) code, this code is compiled with the respective compilers (e.g. a C compiler such as GCC) which produces a .so file (Figure 4). These resources and compiled code are combined and processed by the Android Package Manager tool (apktool), creating an Android Application Package (APK) file. The APK is then optimized and signed using tools like ProGuard for code obfuscation and jarsigner for signing with a digital certificate. At each one of these stages, several optimization procedures are carried and many other opportunities for optimization arise. For instance, AAPT can benefit from optimizing resource files to reduce APK size and loading times. Javac and dex can be fine-tuned to generate more efficient bytecode or minimize the number of classes and enhance app startup times. Additionally, the APK can be further optimized by reducing redundant resources and libraries.

2.3 Performance Analysis on Android devices

In this section, we present the state-of-the-art (energy) performance analysis on Android. We start by presenting techniques and tools to estimate energy consumption on Android devices (Section 2.3.1), comparing them in several aspects and introducing their main advantages and limitations. Afterward, in Section 2.3.2, we present procedures and techniques that allow the use of the previously presented artifacts to automate performance analysis of the Android platform and Android applications. Finally, in Section 2.3.3, we present several relevant efforts from the state-of-the-art that analyzed the performance of Android software in order to provide relevant knowledge regarding the impact of widely used coding patterns and practices in the consumption of the corresponding applications.



Process of APK generation

Process of integrating NDK code within an APK

Figure 4: Standard process of generating a standard Android application distribution package (APK)

2.3.1 Estimating Energy Consumption on Android devices

Energy measurements face numerous external influences that introduce complexities to their accuracy. Despite concerted efforts to mitigate potential confounding factors, there exist persistent challenges in achieving precise measurements aligned with the actual energy consumption attributed to an application. For instance, considering Android devices as an illustrative example, while it is feasible to regulate the temperature of a device to a predetermined level, maintaining an identical temperature across two distinct executions proves to be non-trivial. The inherent variability in environmental conditions and hardware states contributes to a degree of non-determinism in energy measurements. This variability poses a significant hurdle when attempting to conduct rigorous testing for the energy efficiency of mobile applications. Consequently, achieving consistent and reliable energy efficiency assessments remains a complex task in the realm of Android mobile development.

Furthermore, energy analysis in software tends to rely not necessarily in exact measurements, but on approximations, since the available tools to measure energy consumption face physical and technical limitations that affect the precision of the measurements. Consequently, the data collected from energy/power measurement tools is presented in the form of timestamp (t) and power level (P) pairs: (t_i, P_i). Furthermore, determining the total energy consumption, or work performed, within the time frame from timestamp t_0 to t_n involves integrating power over time:

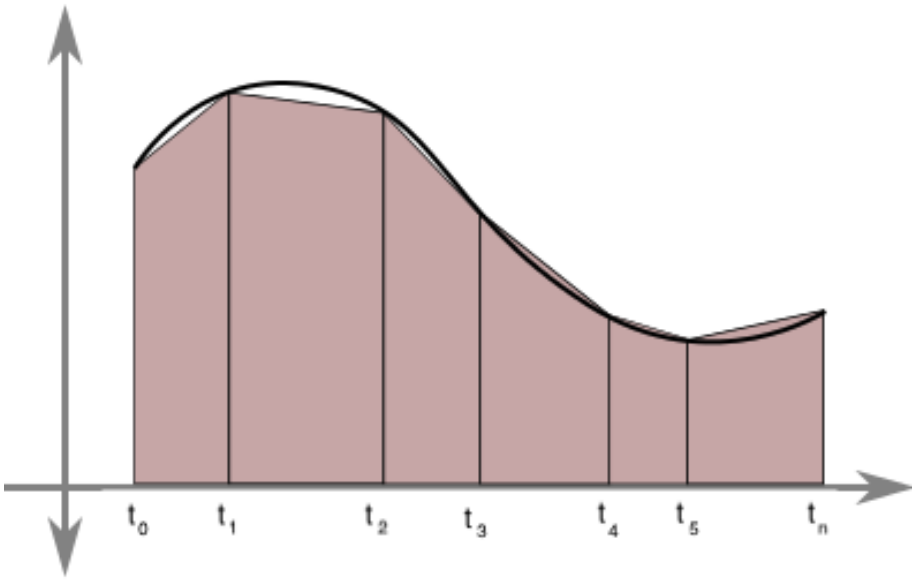


Figure 5: Graphical representation of the Trapezoid rule

$$W = \int_{t_0}^{t_n} P(t)dt \tag{2.1}$$

However, given that P is a continuous variable while the available data consists of sampled values, the process of mathematical integration necessitates approximation. This is achieved through a numerical approach, by employing the Trapezoid Rule:

$$\int_{t_0}^{t_n} P(t)dt \approx \Delta t \left[\frac{P(t_0)}{2} + P(t_1) + P(t_2) + \ldots + P(t_{n-1}) + \frac{P(t_n)}{2} \right] \tag{2.2}$$

Figure 5 visually represents this calculation. The energy spent during an experiment corresponds to the area under the Power function curve between the timestamp when the interaction commenced (t_0) and when it concluded (t_n). As per the International System of Units, time is measured in seconds (s), power in watts (W), and energy consumption in joules (J). However, it is always imperative to recognize the significance of accurate approximations in translating discrete data into meaningful insights regarding energy consumption.

Over the years, researchers been have proposing different approaches to reliably and easily measure the energy consumed by software systems. The initial lack of appropriate tools and knowledge, and the fact that operating systems were not yet tailored to provide energy-related information like they did for execution time or memory usage, were challenges to overcome (Pinto & Castor, 2017). There are two main approaches to collecting power data from the execution traces of mobile applications: **a) Energy**

profilers, purely software-based tools that model energy consumption as a function of features that quantify impacting factors of energy consumption and provide an estimation of the energy consumption; **b) Power monitors**, hardware-based tools that measure the instantaneous power delivered to the smartphone. Power monitors are usually more reliable since they rely on physical measures, as opposed to the energy estimates collected with Energy profilers. The typical laboratory setups for using both approaches for empirically estimating the energy consumed by Android devices are illustrated in Figure 6.

Both academia and industry have been developing tools aiming to allow measuring or estimating energy consumption in the Android ecosystem. The most accurate tools tend to be Power Monitors, such as Monsoon (Monsoon, 2021) and ODroid (HARDKERNEL CO., 2022). However, these are not easily available for developers and researchers, incurring a considerable monetary and setup-time cost. They also may imply disassembling the device to bypass the battery, incurring the risk of damaging the device or voiding the manufacturer's warranty. Furthermore, they also limit the reproducibility and parallelization of studies that consider execution on several devices.

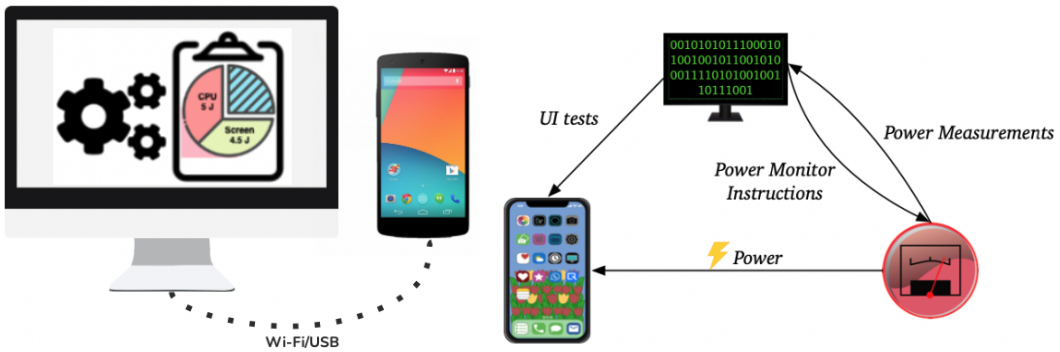


Figure 6: Typical setups for estimating energy performance using energy profilers (at left) and power monitors (at right)

Motivated by the various drawbacks associated with the use of hardware-based tools, a diverse array of software-based energy profilers has emerged. The approach based on energy models was initiated by several works, which focused on building profilers targeted for Android devices. **PowerTutor** (Zhang et al., 2010), **DevScope** (Jung et al., 2012), or **AppScope** (Yoon et al., 2012) are some examples of such works, each one with different goals. All three report the amount of energy consumed by each hardware component. Yet, PowerTutor is able to divide that consumption per active application, while DevScope's main feature is a technique for dynamically and independently updating the model during

profiling, making it more accurate. The strategy employed by DevScope was also followed by AppScope, which ended up being the same type of profiler as PowerTutor (i.e. application-level profiler), but with a substantially increased accuracy. The **PowerProf** (Kjærgaard & Blunck, 2012) and **eProf** (Pathak et al., 2012) profilers are also similar examples, although the context for which they were initially developed is more specific than the previously referred profilers. Using these energy models allowed the aforementioned profilers to account for hardware state changes, which could have a significant impact on energy consumption. Yet, monitoring these variations requires the continuous fetching and processing of state data, which implies that their sampling rate (i.e., the number of estimations per time unit) was quite low (for instance, *PowerTutor* reports has a sampling rate of 1 estimation/reading per second). Hence, when the goal is to analyze consumption at a more fine-grained granularity level (such as methods or lines of code), these tools simply were not adequate.

A different approach for energy models was deemed necessary. As such, researchers explored alternatives to trade-off a more detailed information for an analysis with a finer granularity level, while keeping accurate estimations. In order to do this, two strategies were employed. The first one consists of creating an *instruction-level energy model*. This model consists of assigning to each possible instruction, in a given abstraction level (i.e. assembly, bytecode, language, etc.), a consumption value that best summarizes how much energy is that particular instruction expected to consume. The consumption values were obtained through external power monitors. This was the strategy followed, for instance, by **vLens** (Li et al., 2013), **eLens** (Hao et al., 2013), and **eCalc** (Hao et al., 2012). These works proposed an approach where energy models for Java instructions (vLens and eLens) or bytecode instructions (eCalc) are combined with program analysis techniques, in order to estimate the energy consumption of Android applications, in a particular and controlled execution scenario.

Researchers also explored a data mining approach to create energy profilers, specifically for the Android ecosystem. The idea was to create a technique capable of using test results of several applications and respective execution traces, to predict the energy consumed by repeated code fragments. This concept was explored with **GreenScaler** (Chowdhury et al., 2019). This profiler benefits from a large data collection of energy test results, obtained from a previous work (Hindle, 2013), and uses data mining to create a model which accounts for energy estimates for the usage of several system calls. Upon inspecting which system calls are used when testing an application, GreenScaler can accurately report the expected energy consumed.

These energy models can be useful both for dynamic and static analysis approaches. Nevertheless,

they need to be created for every testing architecture (i.e., each smartphone or computer), which can be a costly and undesired task. The Android ecosystem is characterized by a multitude of manufacturers, each producing smartphones and tablets with their unique hardware components and software variations. This extensive fragmentation extends to the diversity in screen sizes, resolutions, CPU architectures, and operating system versions. The existence of thousands of different Android devices with their specific power consumption characteristics and variations in battery capacities makes developing a universal energy estimation model a daunting task. Consequently, a single energy estimation model often fails to capture the intricacies of the entire Android landscape. This raised a demand for generic energy profilers, which could provide accurate estimations for several testing architectures, without the need to create models beforehand. In order to do this, a deep understanding of the hardware's energy behavior is necessary, hence hardware manufacturers were the ones capable of developing these types of profilers.

Consequently, several attempts to produce more user-friendly and accurate tools that can be integrated with the typical development environment have emerged in the literature. Google itself developed one of such tools and integrated it into Android Studio, the main developer environment for developing Android apps. This energy profiler, **Android Profiler** (Google, 2021b), allows developers to analyze their app's energy consumption by providing a GUI that allows visualizing the variation of the power values being consumed by the CPU, GPS and Networking during app execution. This tool was specially conceived for app developers, being suitable for performing single-app analysis. However, it is Integrated Development Environment (IDE)-dependent, which limits its capability to be ported to automatic processes aiming at analyzing a large set of executions or corpus of applications, since it is built into Android Studio and relies on its GUI and graphical representations to present the power behavior of an app during its execution.

One of the most used profilers to conduct energy-related studies in the literature is Qualcomm's **Trepn Profiler** (Qualcomm Innovation Center, 2016), which allows measuring power consumption in devices with Snapdragon chipsets. However, Trepn has not been updated since 2015 and was even removed from the Google Play Store recently, while there are no reports of the tool providing accurate results in recent devices and platform versions. Besides Trepn, there are other profilers such as GreenScaler (Chowdhury et al., 2019) or **PETrA** (Nucci et al., 2017a) that can be used to estimate energy consumption on the Android platform. However, These tools do not provide warranties that they can work with different devices or Android platforms. GreenScaler is a model-based approach for Samsung Galaxy devices that can estimate energy consumption based on system calls and CPU measurements. PETrA is a monitoring tool that collects dynamic data from Android services such as *batterystats* (Google, 2021a). More recently,

a new tool named **Powdroid** (Bouaffar et al., 2021) emerged, being designed for performing black-box energy analysis, limiting its portability for white-box testing scenarios. While energy profilers exhibit the potential for high accuracy, their precision is heavily contingent on the specific usage context. For instance, up until 2016, the Trepn Profiler stood as the sole profiler that accounted for GPU energy consumption as a contributing factor. Additionally, despite many profilers boasting high accuracy figures, the absence of standardized benchmarks for evaluating these tools makes it challenging to draw meaningful comparisons.

In the realm of energy profiling tools, a more comprehensive analysis can be found in prior research conducted by Hoque et al. (Hoque et al., 2015). In this comprehensive study a taxonomy of these profilers was established, categorizing them based on the following four key aspects:

- **Granularity:** This aspect pertains to the specificity of the profiler when examining energy consumption, considering four distinct possibilities: system-level, sub-component-level, application-level, and API-level granularity.

- **Model Type:** It involves the heuristics employed by the power model to estimate energy consumption on mobile devices. Examples include linear regression, utilization-based models, and genetic models, each offering unique approaches to predicting energy usage.

- **Model Construction:** Where the power model is constructed. It can either be "On device" or "Off device".

- **Deployment Type:** Where the profiler is executed. It can also be "On device" or "Off device", implying that the profiler requires a workstation, a server or a device connected to the device to monitor the profiling process.

This taxonomy provides a structured framework for understanding the diverse landscape of software-based energy profilers, allowing researchers and developers to make informed choices based on the specific requirements of their applications. Inspired by this contribution, we present Table 3, which synthesizes a comparison of the most commonly used state-of-the-art tools for estimating energy consumption on Android. These tools are compared according to not only to the 4 aspects defined by Hoque et al, but by many other criteria, including whether their code is openly available, requires calibration, works only for a limited set of devices, sample rate, etc. Moreover, it also classifies these tools in terms of portability and applicability, by showing in which Android versions they can be used, if they require device disassembly, or if they can be used as a service or third-party library. In this comparison table, we also include

Table 3: Comparison among the state-of-the-art tools for estimating energy in the Android ecosystem

Tool	Open source	Plug And Play	Calibration	Device dependent	API version	Device disassembly	Sample rate	Per-comp. energy estimates	Library
Monsoon (Monsoon, 2021)	✗	✗	✗	✗	All	✓	1 ms	✗	✓
ODroid (HARDKERNEL CO., 2022)	✗	✗	✗	✗	All	✓	1 ms	✗	✓
Android Profiler (Google, 2021b)	✓	✓	✗	✗	≥ 18	✗	1 ms	✗	✗
PETrA (Nucci et al., 2017a)	✓	✗	✓*	✓*	≥ 18	✗	1 ms	✗	✗
Trepn Profiler (Qualcomm Innovation Center, 2016)	✗	✓	✗	✗	$\leq 26*$	✗	100 ms	✗	✓
PowDroid (Bouaffar et al., 2021)	✓	✓	✗	✗	≥ 21	✗	1 ms	✗	✗
Greenscaler (Chowdhury et al., 2019)	✓	✗	✓	✗	All	✗	1 ms	✗	✓
ORKA (Westfield & Gopalan, 2016)	✗	✗	✓*	✗	All	✗	1 ms	✗	✓
Power Tutor (Zhang et al., 2010)	✓	✓	✗	✓	≤ 11	✗	1 s	✗	✓
E-MANAFA (Rua & Saraiva, 2022)	✓	✓	✗	✗	≥ 28	✗	1 ms	✓	✓

E-MANAFA, an energy profiler developed in the scope of this thesis and later presented in Chapter 3. As can be observed in Table 3, *E-MANAFA* presents the best possible values in practically all criteria, except for *API version*. However, according to data reported in 2023, *E-MANAFA* can be used in more than 83,8% of the devices used globally (Composables, 2023), and this figure will tend to increase over time.

Software-based solutions tailored for iOS devices are relatively scarce, with two notable options being the Energy Diagnostics Instrument 4 and the Location Energy Instrument, primarily designed for location-based applications. These tools are accompanied by iOS Software Development Kits (Apple, 2023).

2.3.2 Automating Performance Analysis On Android

The execution of one app depends mainly on several aspects (Pinto & Castor, 2017). In a mobile setup, the number of possible execution scenarios is practically uncountable. According to Android, there are 24000 different types of devices from over 1300 different brands running Android, and this number continues to grow. Also, Android offers over 30 different platform versions (API levels) for its devices and suffers from fragmentation (Park et al., 2013), which is the biggest challenge in testing Android apps according to a recent study (Lin et al., 2020). Context also plays a key role, since the way software is built and used has a critical influence on energy consumption (Ortiz et al., 2019; Pinto & Castor, 2017), which also depends on the static and dynamic features of not only the user (personal features), but also the device, and environment (network conditions, temperature, among others) (Pereira et al., 2021b). This tremendous number of possible execution scenarios makes it almost impossible to achieve meaningful and universal conclusions, as well as gather representative data regarding the apps and platform development paradigm. A large-scale empirical study performed in multiple devices, platform versions, apps and contexts can be a key factor to better understand the software performance behavior (Pereira et al., 2021b).

In order to produce results with relevance and statistical significance about the energy consumption of software, the scientific community tends to use diversified sets of pieces of software that allow obtaining a minimally representative sample of the development paradigm of the platform under study. Regarding mobile platforms, researchers either develop Proof Of Concept (POC) applications or use open software repositories to collect the corpus of applications to be analyzed in their studies. When the objective is to analyze application code in order to representative results, typically the community resorts to open-source repositories (Couto et al., 2015a; Cruz et al., 2017; Das et al., 2020; Ribeiro et al., 2021; Rua et al., 2019; Rua et al., 2020). With regard to studies that involve dynamic analysis of automatically executed applications, the sets of applications used tend not to be large, with a median size of 8 apps and no more than 100 (Kong et al., 2019a).

Using testing frameworks to exercise applications in order to evaluate source code performance is a common approach in the literature. When it comes to running a large number of tests on a set of variable-size apps, collecting or replicating app-specific real-user inputs is impractical. Consequently, developers and researchers use testing frameworks to emulate user interaction. One of the most commonly used frameworks is UI/Application Exerciser Monkey (Monkey for short) (Google, 2021b), which has been used in many previous works (Chowdhury et al., 2019; Hu et al., 2017; Li et al., 2020; Linares-Vásquez et al., 2014; Nucci et al., 2017a) to help detect energy/performance-related issues. Unit testing frameworks such as JUnit or Robotium are used in scenarios where the source code is available (Couto et al., 2015a; Li et al., 2020) or for developing app-specific scenarios when the set of

The Android ecosystem offers a wide range of alternatives for testing applications. Automating application execution using automated execution procedures, such as the ones provided by testing frameworks, is the most common method followed by the scientific community to evaluate the dynamic execution of Android (Kong et al., 2019a; Lin et al., 2020) apps: Correctness (Li et al., 2020), security (Egele et al., 2013), or performance (Rua et al., 2020; Wilke et al., 2013) are among the main requirements evaluated by the community in their studies.

A recent study (Li et al., 2023) proposed a new testing tool to detect energy issues in real-world applications. This study dynamically analyzed the consumption of 36 widely-used applications, using Monsoon (Monsoon, 2021) for power monitoring and the Dynodroid testing tool to exercise applications. This study concluded that 62% of the energy issues detected need to be executed with specific inputs or in special contexts to be detected and only less than 20% can be manifested with simple inputs. Finally, they also concluded that almost 90% of the detected issues in their experiment were previously unknown

to developers and that energy issues are generally harder to fix than non-energy related issues. 2 different studies also tried to assess the impact of logging on Android applications (Chowdhury et al., 2018; Zeng et al., 2019), having concluded that performing intensive logging has a significant impact on application performance. Chowdhury et al (Chowdhury et al., 2018) also conclude that logging at a limited rate (less than 1 message per second) has no significant impact on performance and that factors such as logging rate, disk flush and message size have a significant impact on energy consumption.

In the literature, there are a few attempts to standardize procedures for benchmarking the performance (more concretely, energy performance) of Android apps. GreenDroid (Couto et al., 2015a) was one of the first tools to automate the process of building and running Android projects to obtain estimates of their energy consumption. PETrA (Nucci et al., 2017a) was also included in a pipeline that performed source code instrumentation and was used alongside Monkey to exercise applications while monitoring energy consumption. Malavolta et al. (Malavolta et al., 2020) also presented a profiler-independent tool for the automation of measurement-based experiments on Android devices. This tool was conceived to be profiler-independent and used to replicate the performance-based study of Android apps using a black-box approach, being able to process native and web-based apps. However, all these tools have limited applicability. None of these tools is able to automatically instrument and build modern applications with state-of-the-art tools. GreenDroid is only able to exercise Java apps using unit tests and profile applications with Power Tutor (Zhang et al., 2010), a deprecated profiler. PETrA was only able to exercise apps with Monkey and is not suitable for conducting large-scale empirical studies, since it relies on a GUI to conduct the testing and profiling process. The tool presented in Malavolta et al. is not able to automatically build Android projects or execute apps with different testing frameworks.

2.3.3 Analyzing and Optimizing Performance of Android Software

Developing an application can be an intricate endeavor, demanding developers navigate through interconnected decisions and trade-offs when addressing intricate challenges. To simplify this process, developers often resort to various software solutions such as libraries, APIs, and frameworks to solve common programming problems or to community-driven platforms such as StackOverflow for answers. Within this context, the task of designing and implementing software has evolved into the art of selecting the most suitable solutions from an array of options (Baldwin & Clark, 2000) and integrating them to construct

functional systems. When related to energy consumption, these various solutions, encompassing programming techniques, idioms, libraries, components, and tools for which there are multiple, interchangeable, readily available solutions that potentially have different energy footprints were previously labeled as energy variation hotspots (Oliveira et al., 2020).

Regarding performance evaluation on mobile devices, several studies and efforts have appeared with a view of assessing the impact of widely used, widely-known programming practices on both app and operating system performance. Many recent studies have assessed the performance impact of software practices at different levels of granularity. From languages (Couto et al., 2017b; Lima et al., 2016; Pereira et al., 2017, 2021a), development tools or software standards (Couto et al., 2017a, 2015b; Oliveira et al., 2023; Pereira et al., 2018), to methods and libraries (Linares-Vásquez et al., 2014; Pereira et al., 2020, 2016), several community efforts have proven that it is possible to optimize software at several granularity levels.

Most studies from the state-of-the-art that analyze native Android apps' coding practices only cover Java Virtual Machine (JVM) source code or bytecode code (Couto et al., 2015a; Couto et al., 2020; Cruz et al., 2017; Linares-Vásquez et al., 2014; Maia et al., 2020; Ribeiro et al., 2021), since Java was for many years the reference approach for Android development. With the growing popularity of other programming languages, frameworks and libraries were created for development that allowed not only to develop code using other programming languages, but also to develop cross-platform codebases. Since recent studies show that different languages have different energy footprints (Couto et al., 2017b; Oliveira et al., 2017b; Pereira et al., 2017, 2021a), other studies have emerged in the last years aiming to analyze the energy consumption of applications built with different languages or development frameworks (Biørn-Hansen et al., 2020; Corbalan et al., 2018a; Oliveira et al., 2023). These studies showed that although the use of these approaches may lead to decreased performance compared to the native development approach, there are several cases where their use even substantially benefits energy consumption. Malavalta et al. (Peters et al., 2021) also evaluated the run-time performance impact of migrating from Java to Kotlin, by dynamically analyzing 10 open-source apps that conducted such migration. The authors concluded that migrating to Kotlin has a statistically significant impact on CPU and memory usage, while not significantly impacting energy consumption. This study also confirmed that most open-source Android apps migrated to Kotlin.

Mazuera-Rozo et al. (Mazuera-Rozo et al., 2020) conducted a large-scale study that aimed to define a taxonomy of the types of performance bugs, as well as to study their survivability through mobile apps'

lifetime. The authors designed 2 distinct classification systems for Android and IOS, since there are several types of performance bugs that are platform-specific. For Android, the authors classified performance bugs into 6 major groups:

- **Resource Leak:** Bugs related to improper handling of device resources. These types of bugs were the most frequently found on the analyzed corpus of Android apps. The authors divided these bugs into two subcategories: memory leak and sub-optimal CPU usage.
- **Performance Bad Practices:** The authors define these bugs as "non-application of well-known performance best practices", i.e., performing the opposite of widely studied performance recommendations. These include many sub-groups, such as "Missing cache Opportunity", "Compiler Settings", etc.
- **GUI-related bugs**: Issues related to User Interface (UI) logic/rendering. For example, performance bugs causing animation lagging belong to this category.
- **Connection-related**: Issues related to network, Inter-Process Communication (IPC)/Remote Process Communication (RPC) communication. These include Broadcast Receiver and Service leaks and slow network issues.
- **Database operations:** Bugs related to either Database access or data manipulation. Examples of these bugs found by the authors are **Inefficient usages of Structured Query Language (SQL)** or **Eager loading of data from Database (DB)**.
- **Unclear:** Bugs that do not fall under any of the considered categories and classifications.

There are also notable efforts from the platform itself to help developers document and detect performance issues on their projects. For instance, Android provides a list of statically detectable source-code performance issues that can be automatically detected in an app's source code using Android Lint, a tool that is integrated with the Android Studio IDE and the Gradle build system (Figure 7) in order to detect this type of issues during the development stage. Android Lint allows the detection of various possible issues of different categories, namely performance issues. This Lint tool and statically detectable performance issues have been widely studied by the research community (Couto et al., 2020; Cruz et al., 2017; Das et al., 2020; Goaër, 2020a, 2020b), by using or extending the tool to detect performance issues, evaluate their impact and refactor them. A recent research effort conducted by Malavolta et al. (Das et al., 2020)

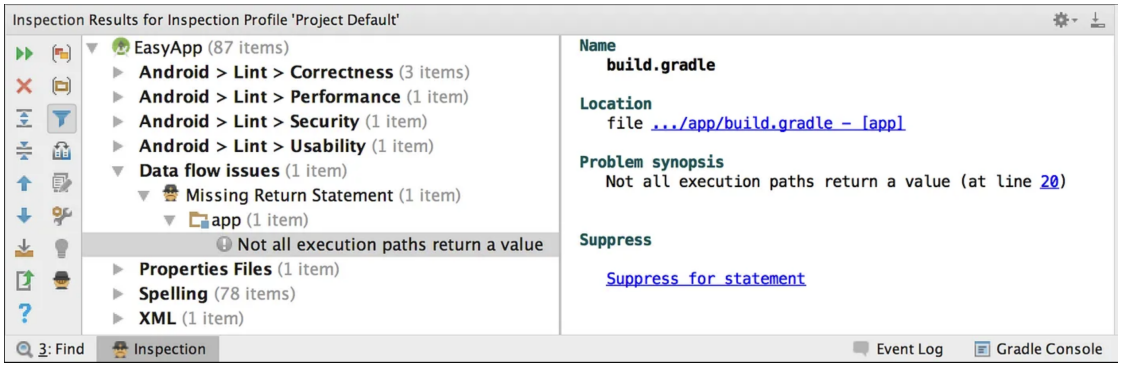


Figure 7: An illustrative example of several statically-detected issues on an Android project in Android Studio with Lint

performed a large-scale study of a large set of Android applications, aiming at analyzing the evolution of statically detectable performance issues. This study analyzed 724 open-source repositories written solely in Java (projects containing NDK or Kotlin code were excluded) of Android apps in order to detect and study the evolution of 9 performance patterns. The goal of the study was to evaluate how much time the energy patterns remained unsolved on these apps, as well as how these kinds of patterns tend to appear in the development lifecycle of Android apps. From the analyzed set of apps, 316 had performance issues, while 45% of these didn't solve such issues.

Source code metrics and other static metrics have also been used to estimate code performance (Hindle, 2012; Kin Keong et al., 2015; McCabe, 1976; Rua et al., 2019). In regards to energy performance, several indicators may be extracted from application source code that helps predict their impact on the consumption of resources of the host machine. Focusing specifically on mobile platforms, Hindle et al. (Hindle, 2013) evaluated the impact of object-oriented metrics on energy consumption, concluding that there is some promise in looking at these metrics. Nevertheless, they also concluded that more data and analysis are needed. More recently, a study (Kin Keong et al., 2015) showed that static metrics such as McCabe Cyclomatic Complexity (McCabe, 1976), number of parameters, nested block depth, weighted methods per class, number of overridden methods, number of methods, total lines of code and method lines all have a significant correlation with the power consumption of mobile applications.

Linares-Vásquez et al. (Linares-Vásquez et al., 2014) empirically measured the energy consumption of method calls in a small set of 55 Android apps to identify energy-greedy APIs (or Red APIs) and usage patterns. This study identified a total of 131 energy-greedy APIs and several guidelines targeting app developers aiming to save energy on their apps, such as avoiding certain software design principles such as Information Hiding. More recently, Li et al. (Li et al., 2020) used 2 different testing frameworks to

execute apps in different execution contexts and with different inputs to detect energy issues in 27 real-world apps. Both Lint issues and these Red APIs will be under analysis, in terms of their performance impact, with their respective results and conclusions presented in the following sections.

Efforts by the research community have also shown that it is possible to improve the energy performance of applications without making changes to their original source code. Bangash et al. (Bangash et al., 2022a) proposed a technique based on the Redex tool [1], which aims to change the bytecode of applications in order to improve run-time performance and reduce application size. A preliminary evaluation of this tool carried out over 22 applications allowed to improve the performance of 12 applications. However, this experiment allowed the authors to conclude that the proposed optimizations do not present universal results, since they did not always lead to consumption improvements in some of the analyzed applications. A recent study (Rua et al., 2020) evaluated the performance of several of the most widely used keyboards on the Android platform and concluded that it is possible to save a significant amount of energy just by changing to a more energy-saving keyboard or keyboard configuration. Also, another study (Gonçalves et al., 2022) focused on the most widely used Android browsers also concluded that there are significant differences in terms of energy consumption among these browsers. This study evaluated the efficiency of each browser for different use cases and concluded that Chrome was the most energy-friendly browser for tasks such as playing videos on Vimeo and browsing Facebook.

There are also several efforts in the literature that aim to reuse knowledge previously obtained in other studies of the literature to help and encourage developers to correct energy issues in their applications. Luis et al (Cruz et al., 2017) proposed a tool for the automatic detection and repair of 5 energy issues cataloged in a previous study (Cruz & Abreu, 2017). Another study conducted by Ribeiro et al (Ribeiro et al., 2021) proposes an Android studio plugin to help develop more efficient Java code, having the ability to do automatic detection and refactoring of 5 well-known energy smells. In order to evaluate the tool's effectiveness, it was used to analyze 100 different applications, having been able to perform 42 refactorings in 35 of the analyzed applications.

[1] Redex: https://github.com/facebook/redex

3

Automating Performance Analysis on Android ecosystem

Abstract: As we have seen in Chapter 2, estimating and comparing software energy consumption is far from trivial. Such limitation becomes disconcerting when simulating typical software usage scenarios and comparing similar applications or coding practices that are executed in diverse and non-isolated execution scenarios. In this chapter, we present two different research efforts presented in two distinct publications. These efforts materialized energy analysis and benchmarking techniques into open-source tools that facilitate energy analysis on the Android ecosystem. First, in Section 3.2, we present *E-MANAFA*, a model-based, device-independent energy profiler for Android devices. Lastly, in Section 3.3 we present *PyAnaDroid*, a configurable execution pipeline that facilitates performance benchmarking and analysis of Android applications. In both sections, we delve into the architecture of the tools and into the underlying concepts, elaborating as well on how it can be compared to similar tools and how can be used by researchers and practitioners to analyze energy consumption on Android.



"E-MANAFA: Energy Monitoring and ANAlysis tool For Android"
Rui Rua, João Saraiva
In: 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)
pages: ***1 – 4*** *| DOI:* ***10.1145/3551349.3561342***



"PyAnaDroid: A fully-customizable execution pipeline for benchmarking Android Applications"
Rui Rua, João Saraiva
In: 39th IEEE International Conference on Software Maintenance and Evolution (ICSME 2023)
pages: ***586 – 591*** *| DOI:* ***10.1109/ICSME58846.2023.00077***

3.1 Introduction

Improving the efficiency of mobile devices and mobile apps has become one of the most important concerns for software developers (Pinto & Castor, 2017). As a consequence, a lot of research work in green software is being done in the context of the Android ecosystem. Such research aims at analyzing the energy consumption of mobile apps in multiple ways, such as by measuring/estimating the energy consumed by an application or block of code (Chowdhury et al., 2019; Couto et al., 2015b; Nucci et al., 2017a), by detecting energy-expensive coding patterns (Cruz & Abreu, 2017; Pereira et al., 2016; Rua et al., 2019), or by defining green/red APIs (Linares-Vásquez et al., 2014).

Despite many recent efforts, there is still a lot of difficulty in identifying and repairing energy issues in Android applications. Although there are several techniques for energy analysis in the Android ecosystem, all of them have several limitations that make their use suitable only in certain execution contexts. Energy issues are more difficult to repair than non-energy issues (Li et al., 2022) and typically only reveal themselves in specific contexts (Li et al., 2020). As a result, finding a reliable tool to estimate energy consumption in a given execution context can be extremely complicated.

Although the scientific community has made recent significant efforts to provide insights into mobile app efficiency and programming practices, these studies typically rely on independent and non-reusable procedures, tools, and knowledge (Choudhary et al., 2015; Kong et al., 2019a). The various efforts from the scientific community to provide knowledge regarding the efficiency of mobile apps and programming practices tend to be materialized in the form of empirical studies. These studies typically use similar empirical/testing procedures (Choudhary et al., 2015; Kong et al., 2019a), sharing tools, procedures, and even knowledge. However, the entire procedures tend to be done from scratch whenever a new study is carried out, due to applicability or availability limitations of the procedure followed in previous works. In addition, most of these studies involve a time-consuming process of setup, verification, and validation of data and results, which are rarely reused in later studies. All these non-reusable efforts are part of the process of empirical analysis of applications and their elaboration and execution consume a significant

part of the time required for the design and execution of the study.

To address these issues, in this Chapter we introduce the following frameworks:

- ***E-MANAFA* energy profiler**: a plug-and-play, device-independent, model-based profiler capable of obtaining fine-grained energy measurements on Android devices. This profiler can be used in state-of-the-art devices to obtain reliable estimates of fine granularity. *E-MANAFA* is able to quantify energy consumption estimates of each device component and record events that occurred during time intervals of the monitoring process, such as component state changes, execution of background jobs, etc.

- ***PyAnaDroid***: an open-source framework designed to optimize, enhance transparency, and improve replicability in empirical studies evaluating Android software artifacts' performance. *PyAnaDroid* offers a configurable workflow that analyzes source code, extracts metrics, and provides instrumentation capabilities. It also supports various testing frameworks, instrumentation techniques, profilers (including *E-MANAFA*), and other application analysis tools, facilitating empirical studies by developers and researchers. The framework validates test results and conditions, ensuring result consistency and adherence to specified test conditions.

In order to fill the major flaws of the existing tools, in Section 3.2 we present *E-MANAFA* (Rua & Saraiva, 2022), the first tool able to quantify energy consumption estimates of each device component and record events that occurred during time intervals of the monitoring process, such as component state changes, execution of background jobs, etc. Our tool can be installed on any development machine as a Python package and be used as a 3rd party library or command line tool. It can also be combined with instrumentation techniques to estimate source code consumption. *E-MANAFA* is open-source and open to community contributions. Its operation relies on the Android platform's open-source tools and services. Furthermore, in Section 3.2, we start by presenting the architectures and main features of the tool, comparing it with other profilers from the state-of-the-art. We then introduce the main components on which *E-MANAFA* relies to obtain its estimations (Section 3.2.1) and its workflow (Section 3.2.2). Section 3.2.3 presents some preliminary results that intend to demonstrate examples of results that this profiler can provide. Section 3.2.4 presents some of the limitations of the tool and its respective components.

Regarding *PyAnaDroid*, this framework solves the optimization and replicability problems found in the state-of-the-art by offering an open-source workflow open to extension and contribution. In fact, this

framework has been designed so it can be easily extended to support more tools in its workflow, such as testing frameworks, instrumentation techniques, profilers or other static and dynamic application analysis tools. To demonstrate the functionalities of *PyAnaDroid*, we present set of preliminary results obtained through the execution of the framework. In order to illustrate its relevance to analyzing open-source artifacts, *PyAnaDroid* was used to process an initial set of 245 Android projects extracted from open-source repositories. All data and metrics collected from this analysis are openly available, as well as the execution procedure. With this analysis, we intend to evaluate the effectiveness of *PyAnaDroid* in several tasks, such as building and analyzing third-party Android projects. In summary, with the preliminary study presented in Section 3.3, we aim to answer the following questions:

Research Question 3.3.1

Is it possible to develop a reusable and expandable open-source workflow for analyzing open-source Android applications?

Research Question 3.3.2

What percentage of open-source Android applications have their code ready to be built and executed outside the development environment with minimal effort?

Research Question 3.3.3

Are there differences between black-box testing frameworks in detecting runtime errors in automatic executions of Android apps on state-of-the-art devices?

The results of this preliminary study allow us to conclude that *PyAnaDroid* is able to successfully automate the execution process of open-source apps, being able to build more than 52% of the analyzed projects without any human intervention. Our results also showed that only 46.12% apps have their code ready to be executed on a machine other than the development one. Moreover, only during the execution of 5 apps, no Java Exceptions nor errors were detected by *PyAnaDroid*. Finally, we also observed noticeable differences between the two considered black-box testing frameworks in terms of suitability for detecting runtime errors in Android applications.

Furthermore, Section 3.3 presents the architecture, workflow, and main features of *PyAnaDroid*. Section 3.3.1 introduces the main building blocks that compose this execution pipeline. Afterward, Section 3.3.2 presents the preliminary results of an experiment designed to evaluate *PyAnaDroid*, demonstrating its features and some use cases for this tool. Section 3.3.6 presents a more detailed of the

collected preliminary results, followed by Section 3.3.7 that presents the main threats to the validity of the results demonstrated in our preliminary evaluation. Finally. Section 3.4 presents the final conclusion of the 2 research efforts of this chapter.

3.2 *E-MANAFA* Framework

This section presents the *E-MANAFA* tool, describing the foundations on which it was implemented, as well as how it works as a third-party library or Command Line Interface (CLI) tool. *E-MANAFA* was developed to be a plug-and-play energy monitoring tool, aiming to provide an intuitive and pleasant installation and usage experience for any type of user. Thus, we strive for the *E-MANAFA* to be adopted by researchers and practitioners to perform energy consumption analysis of applications. Especially analysis that covers a large corpus of applications or a large set of executions.

E-MANAFA is a software and model-based profiler, which allows estimating the energy consumption of Android devices at a low level of granularity. Like other state-of-the-art profilers (Bouaffar et al., 2021; Chowdhury et al., 2019; Nucci et al., 2017a; Qualcomm Innovation Center, 2016), despite providing system-level measurements, *E-MANAFA* can be used to estimate energy consumed by applications. Thus, energy measurements must be carried out in controlled environments, where precautions have been taken to minimize interference from system processes and other applications (Gonçalves et al., 2022; Rua et al., 2020). *E-MANAFA* is the only profiler known to the authors capable of providing quantitative estimates of the energy consumed by each device component. *E-MANAFA* can be used alongside instrumentation techniques to estimate source code consumption. The tool has already been integrated into the *PyAnaDroid* execution pipeline, which precisely uses *E-MANAFA* together with code instrumentation to benchmark Android applications.

E-MANAFA is a device-independent tool, since its operation is based on resources present in the Android Standard Development Kit (SDK) and platform. The tool can be used in any operating system where the Android SDK can be installed. Furthermore, it requires no calibration, since it relies on data provided by the manufacturer and system services to estimate energy consumption and this data is extracted from the device when the profiler is executed. To be executed in a device, our tool relies on 4 Android artifacts, namely: the *power_profile.xml* file, the *batterystats* service, the *perfetto* (Google, 2022d) service, and (eventually, the utility *logcat*. We describe each of them in the following sections. Figure 8 presents a high-level overview of the profiler architecture.

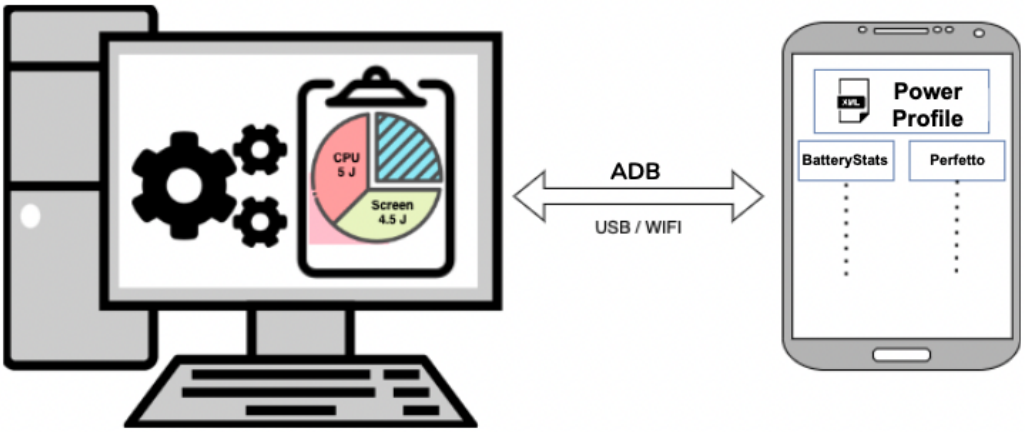


Figure 8: *E-MANAFA* architecture diagram

E-MANAFA tool is already being used to monitor, analyze and optimize the energy consumption of Android applications, namely in identifying energy hotspots in Android applications and energy-greedy libraries. The tool produces results in the form of JSON files, that can also be interpreted in an interactive and user-friendly graphical form developed for such purpose. Figure 9 shows the result of such *E-MANAFA* results inspector, where the pie chart shows the overall energy consumption of device components and in the (bottom) methods section, the energy consumption for each app method invoked is presented.

3.2.1 *E-MANAFA* Main Components

This section presents the main components and dependencies on which *E-MANAFA* relies to provide its functionalities. As mentioned in the previous section, the framework relies mainly on 4 different components: on the power profile file and the *batterystats*, *perfetto* (Google, 2022d) and *logcat* services. The following sub-sections present each one of the components and elaborate on how *E-MANAFA* uses these components to obtain energy estimations on Android.

3.2.1.1 Power Profile

In order to compute the energy consumption of an application during execution time, *E-MANAFA* uses the power profile (Google, 2022c) XML file, which is present on all devices running an Android platform. This file is provided by the manufacturers and contains an estimate of the energy consumption of each hardware component of the device, in all its possible execution states. The values present in this file are dependent on the platform and can vary significantly. This file is located inside the APK located on */system/framework/framework-res.apk*, being extracted by *E-MANAFA* before each execution with the

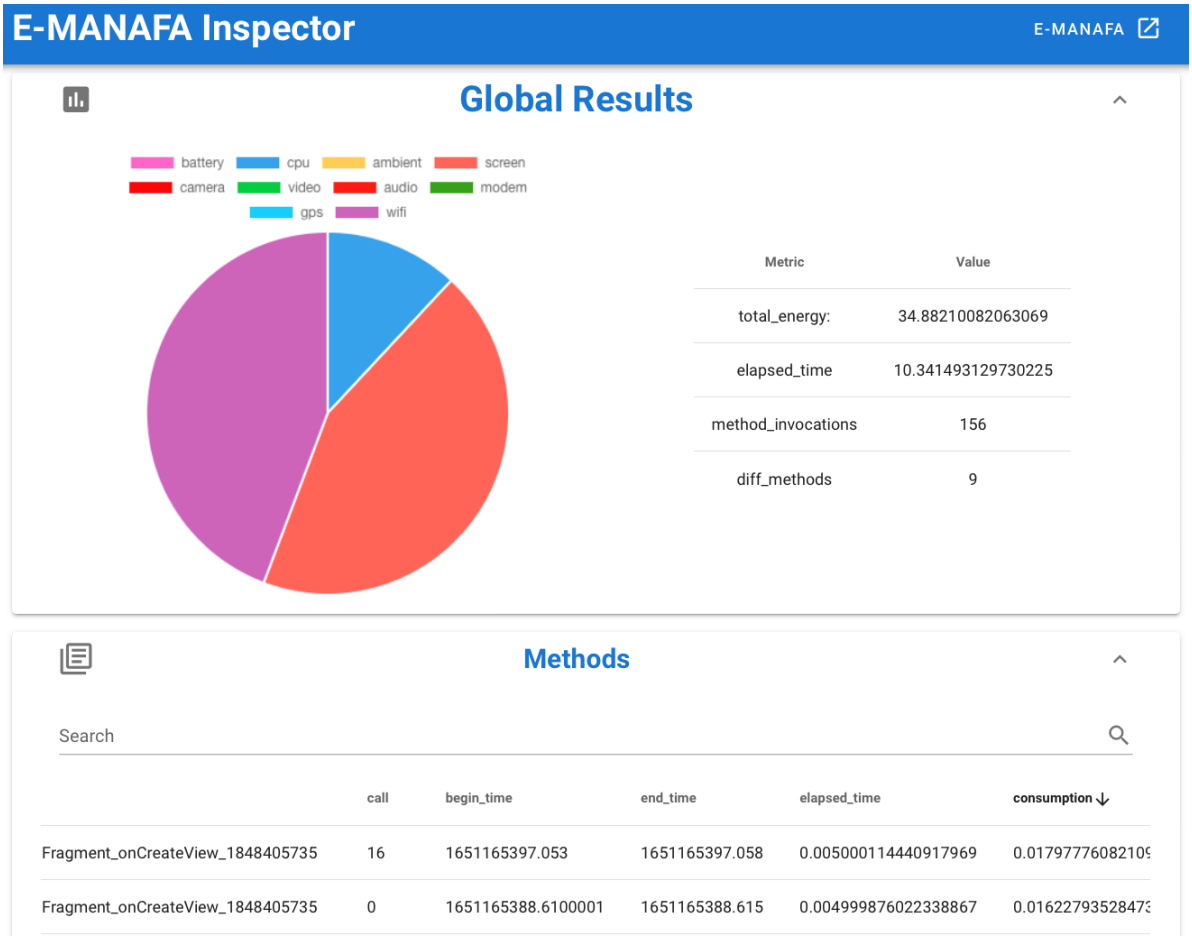


Figure 9: E-MANAFA Results Inspector

tool *apktool* (APKtool, 2022). Although most manufacturers provide accurate values for all possible states of the hardware components, Google also provides a guide to derive these values.

3.2.1.2 Batterystats

Android's *BatteryStats* service is a component that collects and maintains statistics related to battery usage on Android devices. This service is running on every device after Android 5.0 and is automatically flushed after every device's full charge. BatteryStats organizes the collected information in the form of logs and divides it into sections, the main one being the Battery History section.

BatteryStats' Battery History section shows a timeline of every battery-related event with a millisecond granularity. It records the occurrence of all types of events relevant to the analysis of energy consumption, such as changes in components' state (for example, screen or wi-fi), applications executing in the foreground and background, etc. *E-MANAFA* uses this tool to determine the state of each hardware component at a given instant of time during the monitoring session.

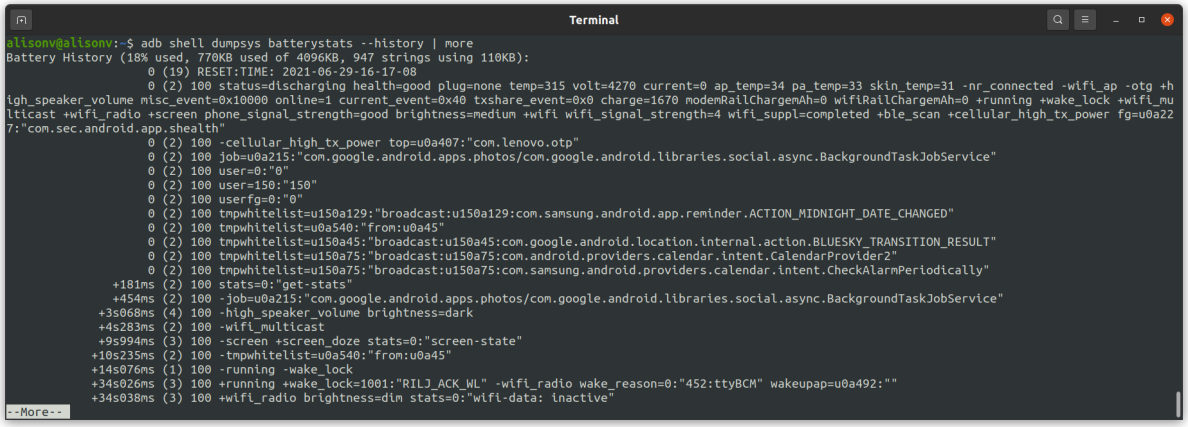


Figure 10: Example of a log of a BatteryStats' Battery History section

3.2.1.3 Perfetto

Perfetto is an open-source stack for performance analysis of different platforms developed by Google. It can be used to profile events from several data sources, such as CPU or memory-related events. *Perfetto* on Android is based on platform services available since Android 9 (P). *E-MANAFA* uses *Perfetto* to trace CPU frequency scaling events, since it can record an event every time the in-kernel *CPUfreq* scaling driver changes the frequency. On most Android devices the frequency scaling is per cluster (group of cores), where the CPUs of the same cluster change frequency at the same time. Also, many devices possess heterogeneous CPUs with clusters that work at different frequencies. *Perfetto* is able to detect individual CPU frequency change events per core with a microsecond granularity.

3.2.1.4 Logcat and app instrumentation

The Android *Logcat* service is a powerful tool that allows developers and system administrators to view and analyze log messages generated by the Android operating system, as well as applications running on the device. "Logcat" is short for "log cat," where "cat" stands for concatenate, referring to the command used to display the logs. The Android Logcat service does not store its information in a persistent storage system for long-term retention. Instead, it outputs log messages to the system log buffers. Since the primary purpose of Logcat is to provide real-time or near-real-time access to the latest log messages for debugging and monitoring purposes, its buffers have limited capacity. Logcat divides log messages into different buffers based on their priority or importance. The main log buffers include *main*, *system*, *radio*, *events*, and *crash*. Each log buffer operates as a ring buffer, meaning that once the buffer reaches its capacity, older log entries are overwritten by newer ones. This allows Logcat to maintain a rolling record

of recent log messages without consuming excessive storage space.

E-MANAFA uses Logcat to inspect the logs registered during a profiling session in order to detect method traces of executed apps.*E-MANAFA* inspects the device logs, searching for messages with specific patterns to identify app traces. In order to record app traces, we developed a library that, at compile-time, injects trace calls on methods of the app's source code, using an Aspect-Oriented (Kiczales et al., 1997) approach applied on Android native projects. We will later see that *PyAnaDroid* (Rua & Saraiva, 2023b) already automated the process of injecting the library in Android Projects in order to obtain instrumented APKs of apps that can be analyzed with *E-MANAFA*. Log inspection can be enabled/disabled in *E-MANAFA*, according to the type of the profiling session (app-level or system-level profiling).

3.2.2 *E-MANAFA* workflow

E-MANAFA integrates three main services that we described previously: *batterystats*, *perfetto* and *logcat*. The energy monitoring process can be summarized as the process of managing these 3 components and extracting and consulting the power profile file. The total consumption of the session and the consumption per component are calculated based on the information collected from these artifacts and are computed according to the following equations 3.1 and 3.2.

$$P(c,t) = I(S(c,t)) * V(t) \tag{3.1}$$

$$J(\Delta t) = \sum_{t=0}^{t_n} \sum_{c\ \epsilon\ C} P(c,t_n) * (t_{n+1} - t_n),\ t_n\ \epsilon\ \Delta t \tag{3.2}$$

Equation 3.1 demonstrates the formula used by *E-MANAFA* to compute the power consumed by a device component *c* in a given timestamp *t*. This equation is nothing more than the estimation of P(t) at instant t in Equation 2.1 presented in Chapter 2. In this equation, the instantaneous voltage *V(t)* is calculated at the system level and its value at a given time instant can be given by the data obtained with *batterystats*. *S(c,t)* consists in determining the state of the component *c* at the instant *t*, which can be inferred by analyzing the *batterystats* logs or, in the case of the CPU, by inspecting the *perfetto* logs to infer the CPU frequency at instant *t*. Finally, the electrical current values I, in Amperes, are estimated from the data in *power_profile.xml* since this file contains the current consumed by each component at each state. The exception is the CPU component, in which case the file contains current values that are drawn at different frequencies. When the file does not contain the CPU frequency at instant *t*, we estimate

its value via linear interpolation.

Equation 3.2 estimates the energy consumed by a device, in Joules (J), in a given time interval Δt. This value is obtained by adding the consumption obtained for each time interval belonging to Δt between which the device components maintained their constant state. For each of these intervals $[t_n, t_n + 1]$, we calculate the product of the power $P(c, t)$ by the duration $t_n + 1 - t_n$, which corresponds to the calculation of the energy consumed by each component at a given time frame in Δt. This equation is a different and equivalent expression of the Trapezoid Rule Equation 2.1 presented in Chapter 2 which considers the inclusion of a new independent variable c.

3.2.3 Preliminary Results

In order to demonstrate the granularity level of energy consumption measurements that *E-MANAFA* obtains, we present results regarding the execution of two versions of the same app: the original application and its instrumented version using the aspect-oriented, compile-time approach. Such approach instruments app methods with calls to *Log* class methods containing the method id to trace method calls. This analysis allows us to present the capability of *E-MANAFA* to detect the energetic impact of minor changes performed in the functional behavior of Android applications, more specifically in detecting differences in terms of CPU energy consumption between 2 similar versions. For this purpose, we used a sample app containing only 50 methods in its source code.

In order to ensure that the results are consistent and independent of the tests performed, each version was executed 25 times with 2 different testing frameworks: Monkey (Google, 2021b) and Droidbot (Li et al., 2017). The test parameters were manually calibrated in order to generate tests with similar duration (approx. 75 seconds). Figure 11 contains the distribution of the values obtained for runtime and CPU energy for each of the versions of the application and execution of the framework. The runtime values are presented in seconds (left diagram), while the energy values are presented in Joules (right diagram).

As can be seen in Figure 11, the instrumented version consistently produced higher energy values than the original version, when comparing the median values (the horizontal lines inside the boxes), top whiskers, and first percentile. Comparing the medians, the instrumented version consumed 34.3% more CPU energy when executed with Monkey and 1% with Droidbot, although the runtime values of the tests are very approximate and consistent and the top whiskers and first percentile of the energy plot are similar. Furthermore, we can conclude that *E-MANAFA* is able to detect the instrumentation overhead since the instrumented tests consumed more energy than the non-instrumented versions. It is noticeable that in

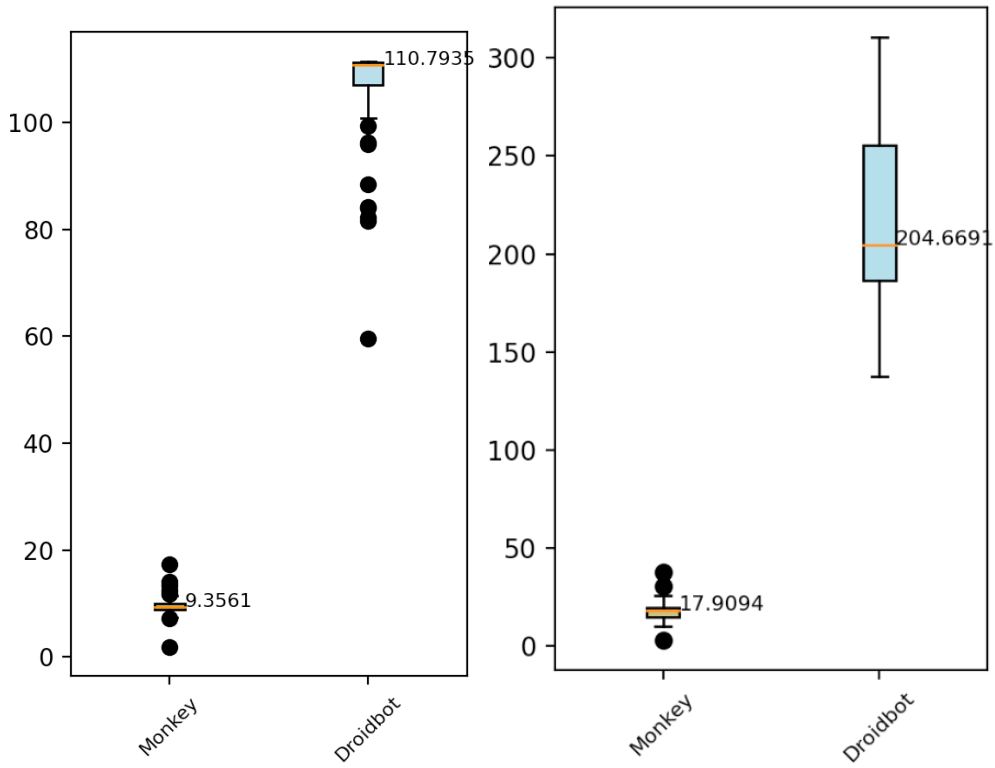


Figure 11: Comparison between non-instrumented and instrumented (+I) app versions in terms of runtime and CPU energy consumption, respectively

terms of median the values with Monkey are much higher, although the runtime values of the tests are very approximate and consistent and the top whiskers and first percentile of the energy graph are very similar. By analyzing these plots, we can conclude that *E-MANAFA* is able to detect the instrumentation overhead since the instrumented tests consumed more energy than the non-instrumented versions.

The results also allow concluding that energy is not directly proportional to runtime, since for instance, the tests executed with Monkey in the instrumented version consumed $34,3\%$ more energy than the non-instrumented version, despite in terms of runtime the difference between the versions being only of 0.1%.

3.2.4 Limitations

E-MANAFA presents the accuracy offered by well-known energy profilers that are based on Android services such as *batterystats* (Nucci et al., 2017a). Moreover, the sample rate of *E-MANAFA* is equal to or greater than one millisecond: the maximum granularity level provided by *batterystats* to record the state of the device's components. In order to confirm the accuracy of our approach, we are now conducting a study to compare the *E-MANAFA* energy estimations with real energy measurements provided by external tools.

The tool can be applied to any device that has a version higher or higher than version 9 of the platform,

which is the version from which the *perfetto* service was introduced. Thus, it is expected that the tool fills one of the major flaws found in many recent empirical studies in the literature, which is the execution of energy analysis studies on deprecated versions or devices or unrepresentative of the Android platform usage paradigm (Bangash et al., 2022b; Chowdhury et al., 2019; Escobar-Velásquez et al., 2022; Li et al., 2020).

The tool can be applied to any device that has a version higher or higher than version 9 of the platform. Thus, it is expected that the tool fills one of the major flaws found in many recent empirical studies in the literature, which is the execution of energy analysis studies on deprecated versions or devices or unrepresentative of the Android platform usage paradigm (Chowdhury et al., 2019; Escobar-Velásquez et al., 2022; Li et al., 2020).

The results reported by the tool have not yet been compared with results obtained with well-known accurate tools using external monitoring devices. Thus, it is not yet possible to comment on the accuracy of the values returned by the tool. However, it is assumed that your results will be accurate, as they are based on services (such as *batterystats*) that are used by the Android platform and other profilers (Bouaffar et al., 2021; Google, 2021b; Nucci et al., 2017a) to estimate device power consumption. Its sample rate is equal to or greater than one millisecond, as the maximum granularity level provided by *batterystats* to record the state of device components.

3.3 *PyAnaDroid* Framework

This section provides a detailed overview of the *PyAnaDroid* framework, presenting its main components and features and demonstrating how it can assist practitioners in analyzing and benchmarking Android applications. Motivated by the open-source movement, this tool is openly available for the community to use and contribute, aiming to standardize and make transparent the empirical experimental process of analyzing applications' energy consumption.

PyAnaDroid was developed to be easily integrated with the typical Android development environment and can be used to interact with any Android device or emulator. Furthermore, it was designed to be easily installable and configurable, resulting in a plug-and-play experience on any machine with Android SDK configured, relying solely on Android SDK tools and open-source libraries. As the name suggests, this framework was developed purely in Python, a multi-paradigm, cross-platform and widely used language, being the third most used language in 2021 (Overflow, 2021). The fact that it is widely used and can run

on nearly all platforms encourages its use and puts forward community collaboration for developing and improving its functioning.

PyAnaDroid is a fully configurable pipeline composed of several blocks/modules that represent processing that typically occurs in dynamic application performance analysis studies. Each implementation of its modules represents the integration of a state-of-the-art tool used by the scientific community to carry out analytical studies. *PyAnaDroid* allows replicating a workload over a set of applications, being able to monitor the execution of this workload and produce reports regarding the application's performance. In addition to providing several implementations of each of its blocks, the framework can easily support other relevant blocks for app analysis, such as refactoring or code repairing blocks. Nevertheless, *PyAnaDroid* also presents other essential features to carry out dynamic analysis studies of applications:

- ***Test condition enforcement***: Enforce restrictions on the state of the device to guarantee that the tests performed on the applications occurred under certain conditions (e.g. Wi-Fi, Bluetooth, screen, GPS) or fatal errors detected in the system logs). Violations of these restrictions may cause the testing process to be interrupted or the test to be repeated.
- ***Test validation***: Enforce a set of validations to be checked after each execution to evaluate to detect anomalies. A violation of this verification can cause the testing process to be interrupted or the test to be repeated, according to user preferences. Each supported validator provides a wide range set of binary operators to perform value assertion to perform test validation.
- ***Device abstraction***: *PyAnaDroid* provides an API to automate device interaction. Examples of supported features are establishing the type of Android Debug Bridge (ADB) connection, screen locking/unlocking, enabling/disabling device sensors, touches, gestures, text input, etc.

The framework's architecture follows recent state-of-the-art guidelines(Schuler & Anderst-Kotsis, 2018), being able to perform both white-box and black-box testing, depending on the selected testing procedure. It is also able to instrument the source code of Android projects in order to include calls to tracing or energy profiling tools. It can also automatically build Android projects and install the resultant APK(s) on a device. The installed apps can be automatically executed by the supported testing frameworks while their execution is being monitored by an Energy Profiler. Finally, it can analyze and validate the execution of each test, presenting the final results and metrics to the user.

The framework offers several implementations of each block in the execution pipeline, allowing users to replace software components with minimal effort or develop new parts to integrate into the pipeline.

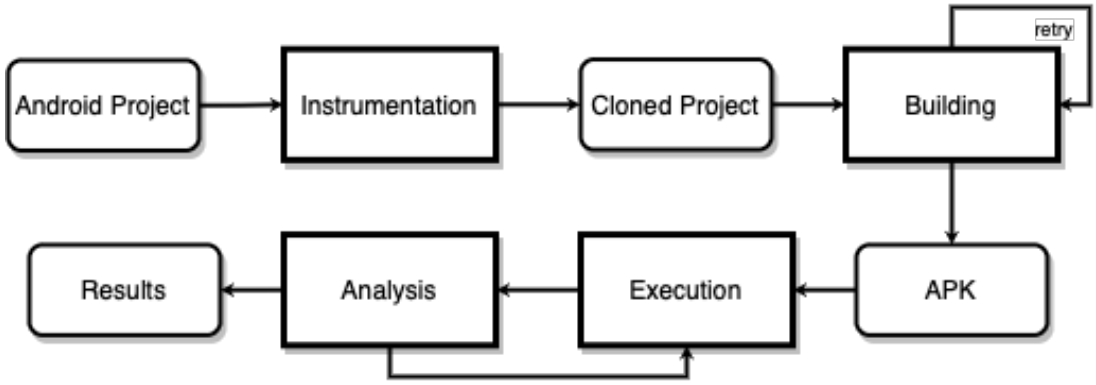


Figure 12: *PyAnaDroid* default workflow

Each of the blocks represents a task typically performed when executing and dynamically monitoring applications, being its execution optional. In this section, we introduce the main stages that compose the framework pipeline. Fig. 12 contains the default workflow of the pipeline when an Android project is provided as input. Each stage of *PyAnaDroid* is represented by a square, while the inputs/outputs of each block are represented by the rounded blocks. These blocks can be interchanged to elaborate different execution pipelines. The framework supports different types of input, which are presented in Section 3.3.1.1 and originate different execution pipelines.

In the default workflow, when an Android project is provided, the project passes through the Instrumentation stage, which clones the project and transforms the project's sources (Section 3.3.1.2). Afterward, the cloned project is built using the building block presented in Section 3.3.1.3. The expected output of such block is one or more APKs, consumed by a process illustrated in Figure 12 as *Execution*, which is detailed in Section 3.3.1.4. This process can have several iterations, which can be validated by the *Analysis* block detailed in Section 3.3.1.7.

3.3.1 *PyAnaDroid* Building Blocks

In this section, we present the building blocks that compose the *PyAnaDroid* workflow. As mentioned in the previous section, each one of these blocks consists of a step usually performed when analyzing and benchmarking software performance. The pipelines contain several implementations of each one of the blocks and these blocks can be interchanged and/or omitted, according to the user preference, which helps to provide the framework with a high level of customization.

3.3.1.1 Input

This section describes the first block of the *PyAnaDroid* framework execution pipeline. This framework supports 3 different types of inputs, whose type is inferred by the arguments passed to the CLI interface or the class constructor:

- ***package names***: Names of the packages of the applications to be tested. This input option assumes that the application is already installed on the connected device.
- ***Application Packages Kit (APKs)***: Path to APKs to test. This option allows the framework to go through all phases of the pipeline if implementations for bytecode instrumentation and APK assemble/disassemble are available.
- ***projects' directory***: Directory where Android projects are ready to be processed.

In addition to these three types of inputs that are processed by the execution pipeline, the framework also supports several input options that allow configuring aspects related to pipeline blocks, test framework configuration parameters, type of building to be performed, etc. The set of supported options is available via the usual *–help* switch.

3.3.1.2 Instrumentation

Instrumentation is a technique commonly used by in software engineering to track and analyze software executions (Couto et al., 2015a; Machiry et al., 2013; Nucci et al., 2017a). Whether performed at compile or run-time, this technique allows injecting custom logic into application code. Currently, *PyAnaDroid* only supports source code instrumentation, performed at compile-time. Furthermore, this step is being performed before the building stage, when processing an Android project as input. However, pyanadroid can be extended to perform APK instrumentation (Hu et al., 2017) or even to support run-time-instrumentation approaches, such as instrumenting the Android framework (Machiry et al., 2013).

To perform code instrumentation, we extended the GreenDroid (Couto et al., 2015a) instrumentation tool, jInst. This tool allows to instrument application methods and test cases of Android projects, to help log and delimit the execution of these blocks. JInst automatically instruments the Java source code of the project, creating a parallel project to the original so that it does not run the risk of corrupting its files and negatively interfering with the development process, a common approach in regression testing approaches. The new version follows the same approach, but now it can also collect static metrics and

metadata, from both Java and Kotlin source code, while performing source code instrumentation, using different approaches.

PyAnaDroid is able to perform two different types of instrumentation, available through the configuration of the instrumentation block provided. In both implementations, the aim is to trace the application's methods. In this case, in addition to tracing methods, it also integrates the application with monitoring with Trepn Profiler. The first implementation consists of using the Android Library named TrepnLib (Rua et al., 2019), an Android library designed to work with Trepn, providing an API that allows isolating and profiling portions/code blocks (like methods, loops or Activities' lifecycles) of apps. By instrumenting the source code with the API provided by TrepnLib, it is possible to estimate the power and resource consumption of such blocks, as well as log other relevant events.

The second implementation, which we call Annotation-Oriented, aims to perform method tracing and temporal localization of the execution of methods in the device's system logs. This approach is based on Aspect-Oriented Programming (Kiczales et al., 1997) and is provided through the automatic injection of annotations in the application's methods and the inclusion of a custom compile-time plugin in the application's code. This plugin is also open-source ("Custom Hunter-Debug Plugin", 2022), being based on the Hunter-debug plugin (Quinn, 2018).

3.3.1.3 Building Tools

This module performs a function that is often a major obstacle to performing empirical studies from source code with a large number of applications: building third-party Android apps. Many studies in the literature related to automated testing in Android rely on open-source repositories to obtain its corpus of applications (Kong et al., 2019b). However, when one intends to obtain and analyze a significant corpus of applications, a procedure to automate their building is essential. Many of these applications were programmed/configured to only be built in the development machine or a certain environment or Android device, requiring a tedious manual setup to be built and executed.

PyAnaDroid offers a building block that allows not only automating the building process of several Android projects but also solving problems of build or integration with the current development environment, instrumentation libraries or profilers. Our framework provides one instance of a building block capable of automatically solving some of the problems encountered in building open-source applications. Once again, support for other building tools can be easily added, by implementing the building block interface for such tools. By providing implementations of such blocks, the process of filtering compilable, buildable

and functional applications in large empirical studies can be improved and streamlined.

The framework contains a single implementation of the building block, that provides support for the Gradle (Gradle, 2023) building tool. This building tool is the default building system for building native Android apps, being used by default by Android Studio and SDK to manage the building and development lifecycle of Android apps. Among the building errors solved by the current implemented block are missing build tools, deprecated configurations, invalid local or NDK properties, and dependencies' incompatibilities, among others. Besides being able to solve these building errors, the block is also able to inject custom configurations in the building scripts, such as *lint* or *dex* options.

3.3.1.4 Execution

This *PyAnaDroid* stage is composed of one or more procedures that involve the interaction and management of the test device with 2 blocks for which the framework provides implementations: the *testing frameworks* block and the *profiler* block. The typical test execution process is illustrated in the following code fragment, where the communication with these components and the Analysis block (whose operation will be detailed in Section 3.3.1.7) is performed. The typical test workflow consists of unlocking the device screen if it is locked, clearing device logs and then starting the profiler and the monitoring process. Then the app is opened and the test is executed on the app. At the end of the execution, the application is closed and the profiler is turned off. The device logs are cleared and the monitoring results are exported and extracted from the device, right before the application cache being also cleared. Finally, the results are validated using the Analysis block, repeating the test if this block rejects the test results, until a maximum of *n_retries* times.

The following sections describe the implementations supported for the Testing Framework, Profiler and Analysis blocks. Section 3.3.1.5 presents the testing frameworks supported by *PyAnaDroid*. Section 3.3.1.6 describes the profilers supported by *PyAnaDroid*. Finally, Section 3.3.1.7 presents some of the software analysis tools that *PyAnaDroid* uses to extract static and dynamic metrics and validate the execution process.

3.3.1.5 Testing frameworks

PyAnaDroid takes advantage of testing frameworks to exercise the Android applications and respective code. This tool currently supports several of the most common testing frameworks from the state-of-the-art, that allow to easily interact and execute applications on Android devices. These tools can be used

Code Listing 3.1: Default Test execution process of PyAnaDroid

```
def test(test_id, device, app, wk_unit, n_retries):
  if n_retries < 0:
      return
  device.unlock_screen()
  device.clear_logcat()
  profiler.start_profiling()
  app.start()
  test_framework.exec_test(app, wk_unit)
  app.stop()
  profiler.stop_profiling()
  device.clear_logcat()
  profiler.export_results(test_id)
  app.clean_cache()
  if not analyzer.validate_test(app, test_id):
      test(test_id, device,
          app, wk_unit, n_retries-1)
```

in the form of interchangeable blocks whose configurations and test parameters can be configured with minimal effort through configuration files. Currently, the tool supports 9 test frameworks, of several test types. Table 4 shows the frameworks supported by *PyAnaDroid*. These testing frameworks are all state-of-the-art reference tools, with industrial and academic applications, having been used in several empirical studies on the Android platform.

Table 4: Testing frameworks supported by *PyAnaDroid*

Framework	**Requires source**	**Description**
Monkey (Google, 2021b)	No	generates pseudo-random system events
JUnit (JUnit, 2022)	Yes	Unit testing
UI Automator (Google, 2022e)	No	functional UI testing
Robotium (RobotiumTech, 2022)	No	UI testing via component abstraction
Espresso (Google, 2022b)	Yes	UI testing of view components
RERAN (Gomez et al., 2013)	No	Record and Replay framework
DroidBot (Li et al., 2017)	No	Model-based UI framework
App Crawler (Google, 2021)	No	GUI and system events
MonkeyRunner ("Monkeyrunner", 2020)	No	functional and regression testing

In addition to these frameworks, others that have not yet been tested based on JUnit might also be supported, such as dexmaker (Linkedin, 2018). Other frameworks can easily be added by implementing the interface defined for the testing block. This interface defines a set of methods that allow setting up and configuring the workload, as well as running the set of tests over an app. Each workload represents a test suite with a set of work units that represent a test to be executed and its parameters. These work units

must also implement an interface defined by us, which allows configuring and executing the respective test that it models.

3.3.1.6 Profilers

To monitor apps' execution, *PyAnaDroid* provides a block that aims to define an interface for communication and management of monitoring tools. Currently, 3 different implementations of such interface are provided, that integrate 3 different energy profilers from the state-of-the-art: Trepn Profiler (Qualcomm Innovation Center, 2016), E-Manafa (Rua & Saraiva, 2022) and GreenScaler (Chowdhury et al., 2019).

Greenscaler is a model-based profiler that estimates energy consumption based on the system calls invoked by the device and the colors emitted by the screen if the device has an Liquid Crystal Display (LCD) screen. This profiler needs to be calibrated for each device, since the weights that its model assigns for each system call are device and hardware-dependent.

Trepn profiler is a power profiler developed by *Qualcomm* that allows, in addition to estimating energy consumption, to obtain a set of performance measures related to the use of device resources, such as CPU/GPU frequency and load, WI-FI and Bluetooth states, etc. This power profiler only returns power results on devices with *Qualcomm* chipsets, as it needs to obtain information from its power management integrated circuits.

E-Manafa is an open-source profiler that allows obtaining performance measurements on Android devices, such as energy consumption, runtime, CPU frequency, or resource usage. To obtain these metrics, it uses as a data source, the power profile file present on each device, the *batterystats* service (to obtain data on resource usage) and *perfetto* (Google, 2022d) (to sample the CPU frequency). Furthermore, it can be used on any device that has these services (all devices running version 9.0 or later of the Android platform).

3.3.1.7 Analysis Tools

This section presents what is typically the last execution block of the *PyAnaDroid* pipeline, the Analysis block. Currently, there are already several implementations of this block in the pipeline, which can be executed consecutively or concurrently to obtain results from the analysis of several elements, such as source code, logs, devices, and services. In addition to producing results, some of these components can also implement the Test Validation functionality introduced in Section 3.3. Currently, several implementations of the Analysis block are provided. The most relevant are:

- *Log Analyzer*: Uses *pylogcatparser* to analyze logs collected from the device during application execution and generate a summary of relevant information detected during execution, as well as to detect errors that occurred during the execution.

- *SCC Analyzer*: Uses *scc* ("Sloc Cloc and Code (scc)", 2022) to extract static metrics from projects' source code, such as the languages contained in project' sources, Lines Of Code, number of files or Cyclomatic Complexity (McCabe, 1976).

- *E-Manafa Analyzer*: Parses the output files of the E-Manafa profiler to validate the results produced, such as whether the monitoring time or the reported energy value is within a predetermined range. It can be used to filter out invalid runs of the monitoring process or to achieve certain testing purposes.

- *Trepn Analyzer*: Parses the output files of the Trepn profiler to validate the results produced and generate a summary of the profiling session.

- *APK Analyzer*: Uses Androguard (Androguard, 2021) to analyze produced APKs and obtain various static metrics, such as the number of methods and classes, the number of instructions or arguments per method, among others.

3.3.2 Preliminary Results

In this section, we present the results of a dynamic analysis performed over a large set of 245 Android projects. With this preliminary study, we intend to evaluate the advantages of using *PyAnaDroid* to carry out large-scale studies and demonstrate some use cases for the framework. Among the presented use cases are Operational Acceptance Testing and comparing the effectiveness of testing approaches and frameworks to detect runtime errors, explore energy hotspots or maximize code coverage. The entire analytical process was made available openly [1], with the respective instructions for its replication. The process carried out on the applications involved transforming and building each of these projects 3 times, which means that it took approximately the same time to build 3 times the number of applications when executed sequentially.

[1] Version of the analytical process used for the preliminary study: https://github.com/greensoftwarelab/PyAnaDroid/tree/icsme_run

The test process consisted of running the tool on applications with 4 different commands. First, to demonstrate the advantage of using *PyAnaDroid* to automate the project building process, the framework was executed with 3 different building approaches:

- ***naive*** approach: Sets up the Gradle wrapper to automate the management and download process of the Gradle and SDK versions used by the project and the generation of the *local.properties* file, which contains essential settings for Gradle to build projects, such as the path to the Android SDK and NDK. This alternative intends to simulate the elaboration of a somewhat limited workflow that could be included in an Android project building automation process (as designed in Couto et al., 2015a).

- ***default*** approach: has all the capabilities of the previous approach, as well as the ability to detect and resolve several building errors.

- ***default+I*** approach: Equivalent to the previous approach, but with source code instrumentation, to trace application methods and estimate their consumption with the E-Manafa profiler. This alternative was used to estimate the intrusiveness of the instrumentation process.

The results obtained by applying the 3 building approaches aim to find answers to **RQ3.2.2**. Finally, to detect execution errors in the analyzed applications (in order to answer **RQ3.2.3**), the last option presented (***default+I***) was also combined with the execution of tests with Monkey and Droidbot frameworks. Since these two frameworks allow black-box testing of applications using various test loads and parameters, running applications with 2 different frameworks not only increases the possibility of detecting errors in applications but also establishes comparisons between the effectiveness of test frameworks for automating the execution of Android applications. Combining the building approach with the 2 dynamic execution processes with success allows elaborating an answer for **RQ3.2.1**.

In short, the empirical process can be summarized by the execution of 4 commands of the framework:

```
$ pyanadroid -d <apps_dir> -nv -i None -p None -bo
$ pyanadroid -d <apps_dir> -i None -p None -bo
$ pyanadroid -d <apps_dir> -t Monkey -p EManafa
$ pyanadroid -d <apps_dir> -t Droidbot -p EManafa
```

All the options used are described in the *PyAnaDroid* documentation. The *-d* option precedes the location of the projects under analysis. The *-nv* option of the first command runs the process with the

naive builder, while the *-i* and *-p* options specify the instrumentation and profiler. Using the *-bo* (build only) option present in the first 2 commands, the processing workflow of each project ends after the building phase. The *-t* option specifies the testing framework for the execution process. The tool rebuilt the projects in the second and third commands since they were built with different configurations. In the last command, the projects are not rebuilt, since the instrumentation and build process is the same.

The process of performing dynamic analysis on a large corpus of applications is a time-consuming process. The time needed to execute an application is influenced by both the performance of the development machine and the Android device on which the tests are performed, as well as the state of the network connections, which are required to download projects' dependencies. The tests were executed on a Google Pixel 3a, running API 29 and version 10 of the Android platform. The building process was performed on 2 machines of the same model, using the same Operative System (OS) version (Ubuntu 20.04) and with the same SDK tools installed.

The following subsections present results related to the execution process. Section 3.3.3 presents the process of collecting applications that are the subject of study and some analysis performed over them. Section 3.3.4 presents the data resulting from the building process with the 3 building alternatives considered. Finally, Section 3.3.5 presents the results related to the dynamic analysis of the applications, in which the applications built with the Monkey and Droidbot frameworks were executed to detect execution errors and also to establish comparisons between the results between the 2 frameworks.

3.3.3 Data Collection

To obtain our corpus of 245 apps, we resorted to using open-source software repositories. Instead of crawling the most common open-source repositories to search for every Android project, we used the information present in AndroZoOpen (Liu et al., 2020). This repository contains links, metrics, and metadata of Android apps which also have their source code present in open-source repositories. To download the apps, we used a custom crawler ("AndroZooOpenCrawler", 2022) that is able to download and filter the dataset applications, as well as to get information regarding project developers and categories according to Play Store (if applicable), among others.

This study considered a large set of 250 apps randomly selected from a larger set of 1000 apps with release packages present on GitHub, which were consecutively downloaded with the custom crawler. We used this selection criterion as we wanted to obtain easily accessible applications, with easily verifiable code and to avoid project versions with limited and unfinished functionalities. When executing the dynamic

Table 5: Languages found in the 248 analyzed projects

Language	#Projects	#Files	#LoC	#CC
Java	213	8713	791453	80205
Kotlin	79	3100	219345	19462
C	9	297	275207	36709
Javascript	7	37	31228	7705
C++	6	129	31431	4637
Python	5	10	654	91
Rust	2	11	545	51
TypeScript	1	13	443	107

procedure, we found that 5 selected applications had their compressed files corrupted, ending the study by considering a final set of only 245 apps. The list of applications used is available at

Table 5 presents a characterization of the applications used in terms of the programming languages used. *PyAnaDroid* is capable of building Gradle projects built in any language, despite only having the ability to instrument Java and Kotlin code. The gathering or accuracy of some metrics (such as code coverage) can be compromised if the projects contain other programming languages. The data collected indicate that Java is the most present language in Android projects, being present in 213 (approximately 86%) of the analyzed projects. Kotlin is present in 32% of projects, with a total of 79 projects. This means that at least 50 applications have Java and Kotlin in their projects. All other languages have a lower presence and representation. C and C++ are typically used to write code to interact with low-level services or for optimization purposes, being interconnected with the remaining framework code via Java Native Interface (JNI). JavaScript and TypeScript are common in web apps, while Rust is present in only 2 projects, as it has only been supported for Android development since 2021 (GOOGLE, 2021). Several programming languages were found in the analyzed projects in addition to those presented in Table 5. However, these languages were excluded from this analysis because they were supposed to be related to building, storage or other tasks outside the application code. Examples of excluded languages are Prolog, Bash, Groovy and SQL.

Table 6 shows some statistics collected from the building scripts of the modules of each project, with their average values being presented. The first 3 lines of the table aim to measure the adaptability of applications to versions of the Android platform. *targetSdkVersion* is an integer value that defines the application's target API version, typically corresponding to the most recent version the application was tested on. When this configuration is not specified, it adopts the value of *minSdkVersion*, which is the

Table 6: Average values for several projects' configurations

Attribute	Average
targetSdkVersion	26
minSdkVersion	17
compileSdkVersion	26
Dependencies	10.87
Test Dependencies	1.42
Android Test Dependencies	1.56

minimum version of the platform where the application can be executed (not implying, however, that it can use all its features). *compileSdkVersion* indicates the API version with which the project was compiled. Evaluating these 3 values and comparing them with the last values reported by Google (Police, 2021) it is possible to observe that the analyzed applications on average support 99.8% of the Android versions that are globally executed. However, they will not have been compiled and run in versions higher than API 26, which represents 78.7% of the executed versions.

The last rows of the table are relevant metrics to measure the complexity of the applications and the respective tests contained in the project. the value of 10.87 for *Dependencies* indicates that, on average, modules contained in projects have 10.87 dependencies for third-party libraries. Test Dependencies have a value of 1.42, which indicates the average dependency for functional tests that can be performed on the workstation. Android Test Dependencies has a value of 1.56, a value slightly higher than the line above, which indicates the average dependency for tests that run on an Android device. This higher value may indicate that Android tests are more complex (Kong et al., 2019a).

3.3.4 Building Results

This section presents the results obtained after applying the 3 building strategies considered in this study: ***naive***, ***default*** and ***default+i***. Each of these strategies was executed independently on the applications, the order in which they were executed being irrelevant to the final result. The process was performed on 2 identical workstations, with identical hardware and with exactly the same version of the operating system and packages installed. Furthermore, they were placed in the same location and shared the same connection, with the same credentials to the same Wi-Fi network. All these precautions were taken into account in order to be able to replicate the process by different machines while minimizing interference.

Table 7 shows the results of applying the 3 building strategies to our initial set of Android projects. Our

Table 7: Building accuracy of different approaches

Approach	#Projects	%
naive	113	46,12%
default	129	52,66%
default+i	102	41,63%

results show that *PyAnaDroid* improves building accuracy by approximately 14% over the *naive* alternative, and was able to process 16 more projects than this alternative. Building with instrumentation(*default+i*) proved to be much more intrusive for the process of compiling and building applications, having only managed to build 102 applications, 11 less than the alternative *naive* and 27 less than the *default* alternative.

3.3.5 Execution Results

This section presents the results obtained after executing the corpus of buildable applications with the ***default+i*** approach with the 2 considered testing frameworks: Monkey and Droibot. For each application, a single test was executed for each testing framework. Each test was run a maximum of 2 times if the results were invalid on the first attempt. If after the second attempt the results return invalid, the results are not considered for this evaluation. In order to validate the results of each test, some *PyAnaDroid* test validation mechanisms were used:

- ***fatal_errors__eq***: This filter was used with the value of 0 to assure that no fatal errors occurred during test execution. In order to determine if an error of such magnitude occurred, a lookup over the log system log tags is performed.
- ***ANR__eq***: This validation filter was used with the value 0 in order to assert that no App Not Responding (Google, 2022a) errors occurred during test execution. To evaluate the occurrence of such errors, the *logcatparser* tool was used, which detects the occurrence of these errors by inspecting the logs collected during the execution of the applications.
- ***total_energy__gt***: Filter used with the value of 0 to assert that the profiler reported a positive value for the energy consumed during the profiling session.
- ***flashlight__eq***: This validation filter was used with the value 0 in order to assert that the flashlight was never used during the execution. The use of this filter was merely illustrative since it was never

verified and served only as an example of a filter that can be used to validate the device's hardware or sensor status.

With the Monkey framework, the same workload was replicated for each application executed, meaning that all applications received the same number and sequence of events, by using the same seed (11) in all tests since Monkey generates the sequence of events through a pseudo-random mechanism. The options for each test to be executed through *PyAnaDroid* can be easily configured via configuration files that are read at each test execution. The tests executed with Monkey were configured to send 1000 system events with an interval of 20 milliseconds, ignoring security exceptions and security exceptions (Google, 2021b). In our execution context, the command obtained for each execution was as follows:

```
$ adb shell monkey -s 11 --ignore-security-exceptions --throttle 20
    --ignore-crashes -p <app_pkg> 1000
```

With Droidbot, a different testing approach was used, based on a Depth-First strategy. Although the tool supports the Monkey for generating events, a different strategy was used to make a comparison that allowed the evaluation of the suitability of 2 different test approaches for the detection of runtime anomalies. Using this strategy, as well as a timeout set to 60 seconds for each test and a 2-second interval between events, resulted in the following event being executed for each application:

```
$ python droidbot/start.py -timeout 60 -interval 2 -policy dfs_greedy
    -a <app-apk> -o <output_dir>
```

The execution of these commands combined with the profiling process with the E-Manafa profiler allows for collecting several results that allow establishing comparisons between the test frameworks and applications executed. Regarding the test process, the graphs in Figure 13 show comparisons between the energy consumed in the test session and by each function during the execution of the 2 frameworks. This Figure show the distribution of energy values in the form of box plots, with the medians labeled inside the boxes. By analyzing the values contained in the figures, it is possible to observe that Droidbot consumed significantly more energy during the test process, having also consistently consumed more energy per method invoked. Comparing medians, in the case of the total energy spent in the testing process, Droidbot consumed 11.43 times more energy during its testing process, with the methods invoked by Droidbot consuming 52.69 times more energy on average than Monkey.

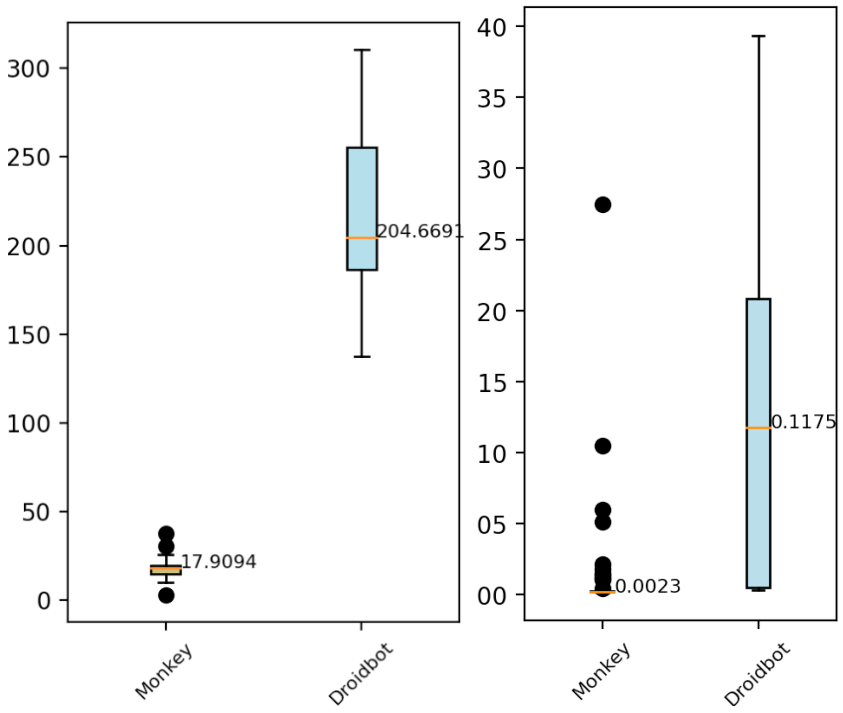


Figure 13: Distribution of the total energy consumed by each test and distribution of the average energy consumption per function, respectively.

Figure 14 presents the comparative results between the 2 test frameworks in the same format, but now for the time and method coverage values. On the left is the distribution of runtime values of the tests carried out, and the distribution of values indicates results consistent with the results obtained in terms of energy, in terms of proportion. Comparing the medians, Droibot tests lasted 11.85 times longer, which helps to explain the energy values shown in the graph on the left of Figure 13. In the graph to the right of Figure 14, the method coverage values are shown, in which, once again, the values obtained by Droidbot are consistently higher.

In terms of errors obtained, Figure 15 presents the distribution of the number of Java exceptions and errors found in the logs during the execution of the 2 frameworks. Only 16 applications did not have any exceptions when executed with Monkey, while with Droidbot 6 they did not raise any Java exceptions. From the graph on the left of the figure, it is possible to see that although the test sessions with Monkey generated more events and are frequently used to perform stress testing, it was consistently less effective in triggering Java exceptions than Droidbot, when comparing the medians and first and third quartiles. However, there are outliers that make Monkey outperform the median values of Droidbot in several tests. Among the most common exceptions are failed web communication attempts with remote and local hosts, since the tests were executed with all the network interfaces disconnected. The graph on the right

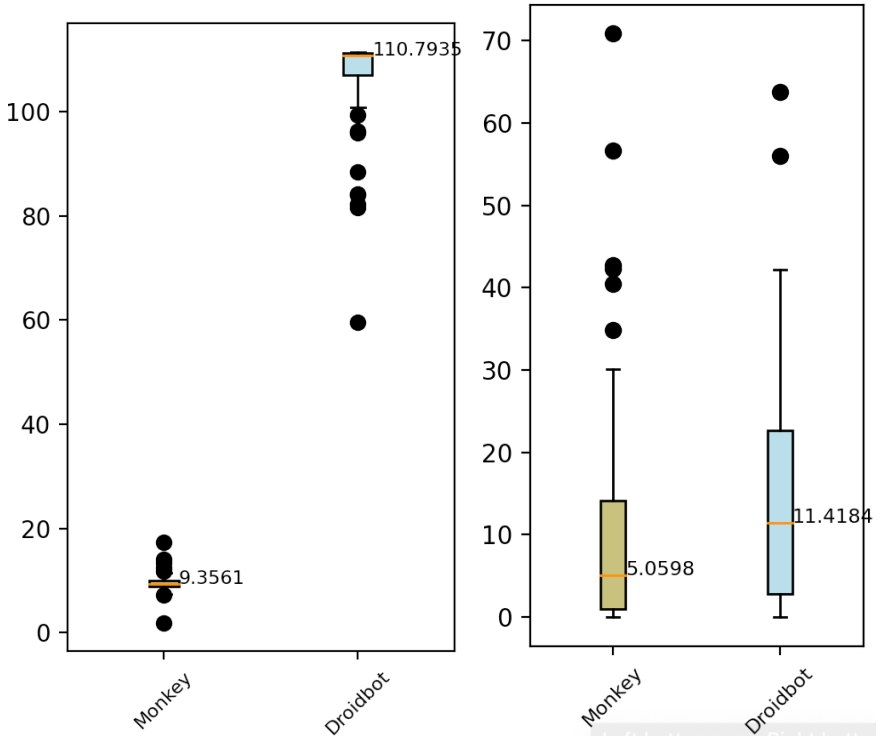


Figure 14: Distribution of the elapsed time by test and distribution of the average method coverage per test, respectively.

shows the distribution of the number of log messages with the tag "*error*" found in each test. Once again, DroidBot consistently manages to obtain more errors, despite Droidbot having several outliers that exceed the maximum values detected in tests with Monkey. Only 4 and 1 applications did not produce any errors when executed with Monkey and Droidbot, respectively.

3.3.6 Discussion

In this section, we discuss the primary results obtained in Section 3.3.2, in order to obtain relevant conclusions about the analysis carried out over the applications, test frameworks and the *PyAnaDroid* execution pipeline itself.

The first results presented in Section 3.3.2 concern the characteristics of the evaluated Android projects, being presented with statistics related to the used programming languages and compilation and building configurations, in the format of tables. The data related to the languages demonstrate the diversity and complexity of the projects considered since projects containing at least 8 different programming languages were analyzed. The configurations presented in Table 7 demonstrate characteristics that

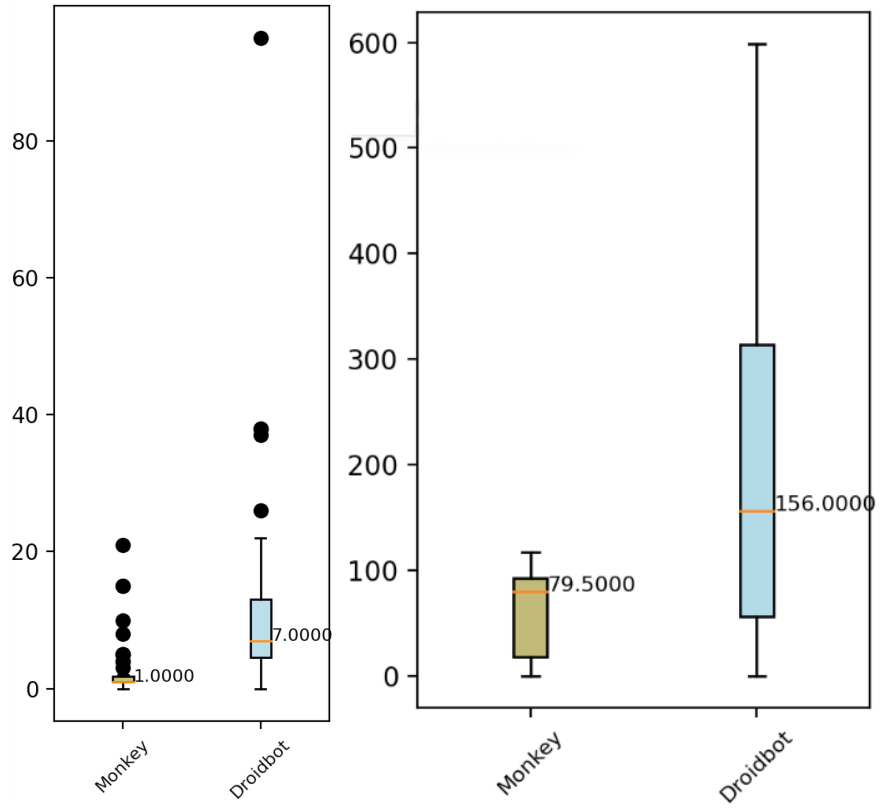


Figure 15: Distribution of the Java Exceptions and errors occurred during the test sessions, respectively.

can limit the system acceptance of applications and that are also illustrative of the complexity of applications. It is important to point out that, for example, the applications target, on average, a different platform version from the one contained in the device where they were tested. This might help to justify the several errors detected during its execution. API changes, access control and permission (Sadeghi et al., 2017) restrictions, as well as other breaking changes might cause unexpected or untested behaviors in the tested applications.

The results presented in the previous section allow us to answer **RQ3.2.1**, since it was shown that *PyAnaDroid* was able to analyze, build and execute a large corpus of Android projects. In terms of building, the alternative ***default*** managed to execute more than half of the considered projects automatically and without any human intervention. The testing process took place over 102 applications resulting from projects successfully built with the building ***default+i*** alternative. This alternative had a lower percentage of success than the version without instrumentation, demonstrating that the building instrumentation process is intrusive to the application code and complicates the process of performing it automatically. It was expected by the authors that the alternative ***default*** would have a higher percentage of success since it has the ability to correct several building errors that we encountered during the elaboration and testing

of the pipeline. However, most of the building errors it solves are issues caused by the instrumentation process. The fact that the pipeline was run with different building alternatives and testing frameworks with minimal effort demonstrates that it is possible to develop a reusable and expandable open-source workflow for analyzing open-source Android applications.

The results of the building process with the 3 distinct alternatives also allow us to respond to the **RQ3.2.2** of this work. The fact that the *naive* approach alone had a building success rate of 46.12% demonstrates that more than half of Android projects are not easily compilable and built in an environment different from the application development environment. *PyAnaDroid*'s *default* building alternative achieved a substantial improvement in the percentage of application building success, having managed to build 129 applications from a total set of . After a manual analysis of many building errors obtained during the execution of the pipeline, it was possible to observe that several errors occur that may eventually be resolved automatically by the tool, namely errors caused by incompatibilities of Java versions to be used for the compilation of the project code. However, we consider that *PyAnaDroid* does more than the minimum effort to build Android projects and we consider as a reference value for the response to **RQ3.2.2** the result obtained by the alternative ***naive***, 46.12%.

In order to answer the last research question of this work (**RQ3.2.3**), in Section 3.3.2 two box plots were analyzed with the distribution of Java exceptions and errors detected in the logs during the execution of the applications with the 2 different test frameworks used. These two graphs show results regarding valid executions of 87 Android applications, where only 18.3% had no Java errors and exceptions when executed with Monkey and 9.9% with Droidbot. In terms of errors obtained, only 5 apps did not present errors with Monkey, while with Droidbot it was only 1. It should be noted that an extensive check was not carried out to verify if the errors were due to the functioning of the app or services on which they depend, or other external components. In Section 3.3.5 the commands used to execute each one of the tools are presented, which allows us to observe that Monkey generated more events on the applications, with a substantially smaller interval between events, when compared to the execution with Droidbot. However, it is possible to observe that Monkey was much less objective in interacting with the application, having obtained less method coverage and occurred in less execution time.

So, the answer to **RQ3.2.3** is that there are significant differences in the ability to detect runtime errors between the 2 chosen test frameworks. The values obtained depend on the execution method, the workload performed, and the runtime error definition. In our setup, no attempt was made to maximize the interaction potential of each of the frameworks with the executed applications. Efforts in this direction,

aimed at carrying out tests of greater length and with a greater load on applications, may be more effective in triggering and detecting anomalies in applications. Our experience demonstrates how *PyAnaDroid* can use different frameworks with configurable workloads to automate app executions, by executing a limited workload that only allows to identify anomalies triggerable by a reduced number of interactions.

The results of our preliminary study allow us to conclude that there are significant differences in the suitability of the 2 test frameworks to detect runtime errors. Only 87 of the 102 (85.3%) applications instrumented with the ***default+i*** alternative were successfully executed, with the remaining 15 applications (14.7%) being rejected by the filters defined in Section 3.3.5 or getting other unexpected errors such as installation failure. However, if we consider Java exceptions or errors detected in the logs during execution, in our context the values vary depending on the test framework. With the workload performed by the 2 test frameworks, the execution of 83 apps presented Java errors or exceptions, and Droidbot was the one that detected it most frequently. As stated in Section 3.3.5, most of the Java exceptions occurred due to the absence of network connectivity, an issue that was already analyzed on open-source Android apps (Escobar-Velásquez et al., 2022).

3.3.7 Threats to Validity

This section analyzes the main threats to the validity of the presented work. This work presents the only tool in the literature that can automatically build, analyze and run Android applications, using different strategies, from its origin (its development project and its code). In the literature, several tools aim to automate the application execution process, but none of them has the ability to analyze and customize the framework presented. As far as the authors acknowledge, there is no other capable of performing source code instrumentation of both Java and Kotlin and performing automatic execution of the code with different profilers and testing frameworks with minimal setup.

In terms of the results obtained, we present below a discussion that aims to evaluate the material presented in this work in terms of internal, external, construct and conclusion validity (van Hezewijk, 1965).

External and internal validity: In this analysis we intend to find out if the results produced can be generalized to different or broader contexts and what factors can introduce entropy in the reported results. As for the framework itself, it was built to be able to process any native Android project and run applications on any device. However, several blocks used for the execution pipeline limit the applicability of the tool in various contexts. For example, the Trepn profiler has not been updated since 2016 and

reports invalid or null results on many recent devices. As for the results reported in terms of language distribution and reported building and execution errors, these may vary between execution setups and contexts.

Construct and conclusion validity: In this analysis we intend to assess to what extent the results presented serve to assess the conclusions drawn and to which extent the conclusions reached are valid. Regarding the evaluation of the applicability and success of *PyAnaDroid* in building and executing applications, we believe that the evaluation procedure took the appropriate measures to carry out an adequate evaluation of these results. The corpus of applications under study was selected at random. The fact that the process was not executed over applications with certain characteristics that could improve the results produced (such as reduced complexity, programmed with certain languages, etc), in our view, is a sufficient guarantee to ensure that these results are illustrative of the effectiveness of the tool to perform this type of automation. Regarding the results presented for the evaluation of **RQ3.2.2** and **RQ3.2.3**, we consider that their results have more limited validity since they depend on the execution context and the applications' dataset. Regarding the last research question, different workloads applied to applications, different test frameworks or executions on different devices might reveal different results.

3.4 Summary

In this chapter, we have presented 2 different tools that provide a valuable contribution to the existing state-of-the-art approaches for estimating or benchmarking the performance of mobile apps. Section 3.2 presented *E-MANAFA*, an open-source, plug-and-play profiler capable of providing fine-grained energy measurement on Android devices. In order to present its main features, a comparison with other similar tools and some illustrative results were presented to demonstrate the tool's capability to detect minor changes in Android applications' behavior.

E-MANAFA is the first energy monitoring tool for the Android ecosystem that can provide quantitative estimates of the energy consumed by each device component. It can be easily installed as a Pypi package and used as a 3rd party library or command-line tool. It can be combined with instrumentation techniques to perform method-level consumption estimates of Android applications.

E-MANAFA was developed to help developers diagnose energy hotspots of their applications and for researchers aiming to carry out empirical studies regarding energy performance. To encourage its adoption, the tool was integrated into the *PyAnaDroid* pipeline in order to benchmark Android apps'

performance.

Section 3.3 presented *PyAnaDroid*, a tool to automatically build, analyze and run Android applications, using different testing strategies. In the literature, several tools aim to automate the application execution process, but none of them has the ability to analyze and customize the framework presented. As far as the authors acknowledge, there is no other capable of performing native source code instrumentation and performing automatic building and execution of the code with different profilers and testing frameworks. The tool is completely open-source and its operation and results can be easily validated and extended by the community. *PyAnaDroid* is also available as a Pypi package in order to be easily installed with a single command in any Python development environment. The tool repository has adequate installation instructions and the tool is offered as a Pypi package, in order to be easily installed with a single command in any Python development environment. *PyAnaDroid* is under continuous development. In addition to the development by the authors, it is expected that the scientific community and Android programmers can also contribute to its development.

The results presented in this work demonstrate that that although Android projects and apps present a wide diversity in the way they are programmed, it is possible to use a generic pipeline that can transform and analyze source code, build applications and execute them automatically, with a high degree of customizability. *PyAnaDroid* can be used by practitioners and researchers for several purposes, such as detection of bottlenecks or inefficient coding practices, calibration of energy monitoring tools, comparison of testing frameworks and energy profilers, mutation and regression testing, among others. Furthermore, the performance benchmarking step is optional, being also suitable for automating the dynamic execution of non-performance-related tasks over sets of applications, such as accessibility checking, GUI testing, among others.

4

Empirical Analysis of native Android source code

Abstract: Several energy performance guidelines from the state-of-the-art lack a more extensive validation since their impact hardly depends on the execution context and can negatively impact other performance indicators or even increase energy consumption in different execution contexts. In this chapter, we present two different empirical studies specially focused on energy performance that evaluate or reevaluate state-of-the-art performance guidelines that were materialized in two different publications. In the first study, we present a preliminary evaluation of the impact of using memoization to save energy in Android applications, where we demonstrate evidence that, in some cases, is possible to save up to 90% of the energy consumed by using memoization techniques. In the latter study, we present a large-scale study that analyzed open-source applications to analyze the impact of several software development and maintenance practices on performance. The conducted empirical analysis allowed us to observe that semantic program changes such as adding functionality and fixing bugs are the changes more associated with relevant impact on energy performance and we demonstrate that several coding practices previously identified as energy-greedy do not replicate such behavior in our execution context and can have distinct impacts across several performance indicators.



"Towards Using Memoization for Saving Energy in Android"
Rui Rua, Marco Couto, Adriano Pinto, Jácome Cunha, João Saraiva
In: 22nd Ibero-American Conference on Software Engineering (CIbSE 2019)
pages: ***279 – 292***



"A large-scale empirical study on mobile performance: Energy, Runtime and Memory"
Rui Rua, João Saraiva
In: Empirical Software Engineering (EMSE), 2023
pages: ***67 pages*** *|* *DOI:* ***10.1007/s10664-023-10391-y***

4.1 Introduction

Battery life is known to be one of the major factors influencing the satisfaction of mobile device users (Thorwart & O'Neill, 2017). A survey with 1,894 smartphone users in the United States of America placed battery life as the most important factor impacting smartphone purchasing decisions (Richter, 2021), with 9 out of 10 users suffering from low battery anxiety (Mickle, 2018).

As a result of the advent of mobile computing, a relevant number of best practices have been delivered to optimize the performance of Android applications. However, some of these guidelines fall short when addressing energy consumption, especially in a mobile device setup. As mobile software applications operate in resource-constrained environments, guidelines to build energy-efficient applications are of utmost importance. Furthermore, several energy performance guidelines from the state-of-the-art lack further validation, since their (beneficial) impact hardly depends on the execution context and can negatively impact other performance indicators or even increase energy consumption in different execution contexts.

Mobile device users have also been showing concerns regarding performance, reporting energy-related issues, and avoiding applications (apps for short) that they identify as energy-greedy (Pinto & Castor, 2017) in order to increase the device's uptime. Also, a research study (Ma et al., 2013) reported that most of the energy issues detected in mobile devices are caused by apps (more than 47%) rather than by the system itself. Often, software developers are not aware of possible performance issues (Li et al., 2020) or do not have the knowledge to correct them (Manotas et al., 2016; Pang et al., 2016; Pinto et al., 2014b). Furthermore, bad performance results in UX degradation and consequently less app usage. Such degradation can also indirectly harm developers, especially those who monetize their apps. Thus, mobile software developers have shown a growing interest in optimizing their applications, while also being energy conscious (Pinto & Castor, 2017).

As far as mobile platforms are concerned, a relevant number of best practices have been delivered to optimize the performance of Android applications from both academia and industry, with the objective of

analyzing different aspects of the energy consumption of software, such as architectural components (Bangash et al., 2021; Linares-Vásquez et al., 2014; Ortiz et al., 2019), programming languages (Couto et al., 2017b; Lima et al., 2016; Pereira et al., 2017, 2021a), or libraries (Linares-Vásquez et al., 2014; Pathak et al., 2011). Even the energy consumed at the testing phase can be a concern and was previously evaluated in prior works (Li et al., 2014). In fact, these works allowed developers to evaluate the impact of common programming practices on the energy consumption of the devices. However, some of these guidelines fall short when addressing energy consumption, especially in a mobile device setup. As mobile software applications operate in resource-constrained environments, guidelines to build energy-efficient applications are of utmost importance. Furthermore, several energy performance guidelines from the state-of-the-art lack further validation, since their (beneficial) impact hardly depends on the execution context and can negatively impact other performance indicators or even increase energy consumption in different execution contexts.

The current studies of the literature have also an evident limitation: each study is validated in one specific combination of hardware/software that can limit the generality of the reported results (Vilkomir & Amstutz, 2014) and do not consider (at least) several performance indicators. A recent survey (Hort et al., 2022) analyzed 156 publications published between 2008 and 2020 focused on mobile applications' performance and identified several gaps in the literature that are still unexplored. For instance, this study evidences the need for an evaluation of whether anti-patterns that exist for several performance indicators such as responsiveness, memory or energy usage, exist for other indicators, such as application launch time.

In this chapter, we present two distinct research efforts that resulted int two distinct scientific publications:

- In the first effort, presented in Section 4.2, we study the influence of incremental computation, obtained via method memoization, on energy consumption in the context of the Android ecosystem. Memoization is an optimization technique used in software development to speed up the execution of a function by caching previously computed results to avoid the computation of expensive deterministic function calls. In this study, we present a systematic study that shows the energy gains of applying memoization techniques to real-world Android applications. In order to perform method-level memoization, we considered side-effect-free methods, i.e., methods that implement functions prone to memoization (Yang et al., 2014). After detecting such methods in the application's source code, we refactored them into semantically equivalent ones that memoize/reuse its

calls. Consequently, by comparing the energy consumption of the non-incremental and incremental versions of several Android applications with the same inputs, we elaborate on the impact of memoization on energy consumption. Our preliminary results show that memoization can greatly improve the energy efficiency of an Android application without threatening its normal functioning and efficiency.

- In the latter research effort, we present a large-scale study aiming at understanding the real-world energy performance impact of several programming practices reported in the literature. To the best of the authors' knowledge, we present the largest empirical study on mobile software and source-code performance known to date. In order to define guidelines to help both software and compiler developers write/generate efficient code, we discuss the following performance indicators: memory, execution time, and energy consumption, with a special emphasis on the latter. In this way, we intend to validate assumptions on the performance impact of several known programming practices, while also presenting new conclusions on practices not yet analyzed until now in terms of all performance components considered in this study. For instance, we evaluate the resultant performance impact of several development/maintenance-related activities performed at source-code level. In order to gather data to synthesize our conclusions, we used model-based GUI testing frameworks in a black box-testing setup in order to evaluate if it is possible to identify relevant changes in an application's energy performance throughout its various releases. After confirming the validity of this approach, and in order to identify the main causes of such performance changes, each change was cataloged and matched with the project source's modifications performed since the last release. Furthermore, we used the data from this analysis to classify and evaluate the performance of statically detected issues by widely-used tools like Android Lint (Google, 2021a), as well as energetically inefficient programming practices (also known as Red APIs) presented in a previous study (Linares-Vásquez et al., 2014). In addition, we carried out an analysis on similarly competing libraries that perform typical programming tasks, in order to assess which libraries might be most efficient or inefficient for certain programming purposes. Our conclusions suggest that many of the analyzed coding practices previously labeled as inefficient have distinct performance impacts across the considered performance indicators and its classification in terms of performance is not evident. We also concluded that different statically detected project source modifications such as fixing bugs, adding layouts or strings and performing refactors are the changes most associated with relevant changes in app performance.

In summary, the results of the studies presented in this chapter aim to answer the following questions:

Research Question 4.2.1:

[RQ4.2.1] *Can memoization help reduce energy consumed by Android applications?*

Research Question 4.3.1:

[RQ4.3.1] *Which program changes have more impact on an Android's applications' energy consumption?*

Research Question 4.3.2:

[RQ4.3.2] *Do previously identified inefficient programming practices exhibit the same behavior on larger datasets and different contexts?*

Research Question 4.3.3:

[RQ4.3.3] *Are there significant performance differences between apps that use different competing libraries implementing typical programming tasks?*

Moreover, our work also presents additional contributions that we consider relevant to the research community:

- A set of performance-oriented guidelines on how to improve source-code performance, which was defined from the observation and analysis of our results.
- a publicly available online appendix (Rua, 2022), where all the analyzed results from the study are available in both graphical and visual form. There were some results that are not covered here due to space restrictions and because we believe they should be analyzed in a dedicated study.
- a dataset resulting from this large-scale analysis, in order to provide the community with data from which more relevant conclusions can be drawn regarding performance and Android code programming practices. This dataset contains both dynamic and static metrics, obtained from our automatic source code analysis and the execution of the applications in physical devices.

As a result of our analysis, we present answers for the presented research questions and a set of guidelines that emerged from the analysis of the millions of measurements gathered in our context of execution. For instance, our study allowed us to reach the following conclusions/findings:

- Using memoization can have a significant impact on the energy peformance of Android applications. Our results of this study evidenced that in some cases is possible to save up to 90% of the energy consumed by application methods just by using memoization techniques.

- Adding functionalities, fixing bugs, adding layouts or strings and performing refactors are the changes most associated with relevant changes in app performance;

- Many coding practices/patterns previously identified as inefficient did not evidence such behavior in our tests. For instance, a relevant part (40%) of the APIs identified as energy-greedy in Linares et. al (Linares-Vásquez et al., 2014) was even more correlated with energy-efficient tests.

- The levels of Severity with which Lint issues are cataloged have little correlation with the correspondent performance impact.

- The evaluation of the performance of apps using different competing libraries that perform similar common programming tasks allowed us to classify the efficiency of each library on each performance indicator. This evaluation suggests that there are libraries that favor one or more performance indicators but end up being the most inefficient in other components. For example, for web communication via HTTP, Volley was the library used in the tests with the best memory and energy efficiency, being the one present in the tests with the worst execution time.

4.2 Towards Using Memoization for Saving Energy in Android

This Section presents a research effort published as a conference paper (Rua et al., 2019). This section is structured as follows:

- Section 4.2.1 introduces the concept of memoization for Android applications and describes in detail our experimental setting.

- Section 4.2.3 presents the results of our preliminary experiment.

- Section 4.2.4 provides a discussion of the results presented in the previous section.

- Section 4.2.5 presents the threats to the validity of this work.

4.2.1 What Can be Memoized?

Memoization is an optimization strategy employed in computer programming to enhance the efficiency of a function by caching previously computed results. The technique involves storing the outcomes of expensive function calls in a data structure, typically a hash table or associative array, where the input parameters serve as keys and the corresponding outputs as values. This cache enables rapid retrieval of precomputed results when the same set of inputs is encountered again, thereby circumventing redundant computations.

The key principles of memoization include the use of a cache for result storage, a check before performing computations to determine if a result is already cached, and its applicability to both recursive and iterative algorithms. However, for effective memoization, functions should be stateless, implying that their outputs should be solely determined by their input parameters, without reliance on external state or global variables. The following example contains an illustrative example of the use of memoization to optimize the calculation of Fibonacci numbers. In this example, The memo dictionary is used to store previously computed results, avoiding redundant calculations for the same Fibonacci number.

Code Listing 4.1: Example of memoization in a recursive function in Python

```
1
2  def fibonacci_memo(n, memo={}):
3      if n in memo:
4          return memo[n]
5      if n <= 2:
6          return 1
7      memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2,
           memo)
8      return memo[n]
9
10 # Example usage
11 result = fibonacci_memo(5)
12 print(result) # Output: 5
```

For the particular case of Object-Oriented Languages, Yang et al. (Yang et al., 2015) proposed a set of three pre-conditions to classify a Java method as memoizable:

1. The method must be pure, that is, it must be a function;

2. The arguments must be immutable;

3. The return value cannot depend on static fields, public member fields nor publicly exposed member fields.

As we will see in the next section, in our experiment, we have simplified the application of pre-condition (2) and used only methods that have as arguments primitive types or the *Context* object of an application (since it is immutable).

4.2.2 Experimental Setup

This section describes the study conducted, describing the Android applications and methods used, how methods were refactored, and the experiment setup. First, we start by describing the data extraction and characterization process (Section 4.2.2.1). Subsequently, we describe the process of refactoring the memoizable methods and we specify the methods that were refactored (Section 4.2.2.2). Finally, in Section4.2.2.1, we present the execution procedure followed to conduct our experiment.

4.2.2.1 Android applications and methods analysis

In order to collect data for our experiment, we resorted to open-source repositories, manually inspecting popular Android projects in search suitable for memoization. We selected 3 different applications: two Android applications from the MUSE repository (Martins et al., 2018) (`Pixate Freestyle`[1] and `Android Demos`[2] and another, `Chanu`[3], from the F-Droid repository "F-Droid - Free and Open Source Android App Repository", n.d. The distribution of methods analyzed from each of these applications is described in Table 8.

`Pixate Freestyle` is an open-access framework that lets users style their native Android views with stylesheets. This application contains 219 classes. `android-demos` is an application from a popular Android project that implements examples of typical tasks that are implemented in Android native applications. This app contained a total of 34 classes. `Chanu` was a well-known application with 538 classes. It consisted of an application where users can browse images of various contents. This is the largest application among the three and so it was where we most applied the technique of memoization.

With the objective of systematically evaluating individual methods susceptible to memoization, we deemed it optimal to isolate these methods from their respective applications, facilitating a controlled

[1] Pixate Freestyle app: https://github.com/Pixate/pixate-freestyle-android
[2] Android-Demos apphttps://github.com/novoda/android-demos
[3] Chanu app: https://f-droid.org/en/packages/com.chanapps.four.activity

experimentation process. To achieve this, we meticulously extracted the candidate methods and incorporated them into a dedicated demo application, maintaining an exact replication of their original functionalities. Subsequently, we produced a secondary iteration of the application, wherein all 18 identified methods underwent a transformation to integrate our memoization technique. This approach ensured a comprehensive assessment of the performance impact associated with the application of memoization to the specified methods under investigation.

Table 8: Characterization of methods used in experiments.

Method	**Input**	**Output**	**Application**
createIntent	Context,String,String	Intent	Chanu
countLines	String	int	Chanu
replyText	Long[]	String	Chanu
join	List<String>, String	String	Chanu
threadSubject	String, String	String	Chanu
textViewFilter	String, boolean	String	Chanu
getUrl	Context, String	String	Chanu
planifyText, quoteText, exifText	String	String	Chanu
getNumeral	String, String	String	Pixate Freestyle
removeLocaleInfoFromFloat, addNegativeSign, addPositiveSign	String	String	Pixate Freestyle
isMobile	String	boolean	android-demos
readableFileSize	long	String	android-demos
dip2px; px2dip	Context,float	int	android-demos

4.2.2.2 Methods Refactoring

In order to select the methods suitable for applying memoization, a meticulous manual analysis of the methods within the three identified applications was conducted. Following the identification of memoizable methods through this analysis, an overarching application was devised to encapsulate these selected methods. Two distinct versions of this application were subsequently crafted: one preserving all methods in their original state, and the other incorporating the memoized counterparts.

The employed memoization technique entailed the utilization of a `HashMap` to store method inputs as keys and their corresponding outputs as values. Consequently, if a given method had been previously invoked with a specific input, its output could be efficiently retrieved from the map and returned. Conversely, for unencountered inputs, a new map entry was generated, and the resultant output was returned. In scenarios where methods received a solitary input parameter, this parameter served directly as the key in the map. Conversely, for instances involving multiple input arguments, a library named javatuples (Yang et al., 2015) was employed. This library enabled the aggregation of input values into tuples, which then functioned as keys within the `HashMap`.

A comprehensive characterization of the methods employed in the experimental setup is presented in Table 8. The table delineates pertinent details for each method, encompassing its name, associated application, input parameters, and output parameters. Notably, instances of multi-parameter methods, such as the *getURL* method in the Chanu application, are exemplified in the table. For these scenarios, the *javatuples* library played a pivotal role in facilitating the storage of input values in tuples, subsequently serving as keys within the `HashMap`.

After finding memoizable methods, we created an application to encompass all those methods. We have then created two versions of the same application: one with all methods in their original form and the other with the memoized ones.

4.2.2.3 Experiments execution

To assess the impact of memoization in the energy consumption of the identified methods, we designed a specific usage scenario. This scenario involved the iterative invocation of a method, repeating the process 50 times with distinct parameters for each iteration. Consequently, 50 entries were successively inserted into the method's map structure. To systematically gauge the impact of memoization, this procedure was iteratively replicated for 10, 20, 30, 40, and 50 repetitions, with each repetition set constituting an individual *test case*.

To ensure a robust dataset, each test case underwent 25 repetitions for every method in both the original and memoized versions, alternately. The experiments were conducted on a Nexus 4 device running Android version 5.1.1 - API level 22. The energy monitoring task was facilitated using the Trepn Profiler (Qualcomm Innovation Center, 2016), an accurate energy profiler for smartphones containing Snapdragon chipsets that was developed by Qualcomm.

For each iteration of the test case, a pre-established setup procedure was executed. This encompassed the removal of the previously installed version of the application under examination, installation of the new version, activation of the Trepn profiler, and a 5-second warm-up period. Subsequently, Trepn commenced profiling concurrently with the execution of the test. Upon completion of the test, energy consumption results were recorded, and both the application and Trepn were deactivated. This process ensured the systematic measurement and analysis of energy consumption across multiple repetitions and variations of the test cases.

4.2.3 Results

In this section, we provide a detailed presentation of the results derived from the experiment outlined in the preceding section.

Figure 17 exhibits distinct box plots corresponding to each of the 13 methods for which a statistically significant difference was observed between the energy consumption of the original and memoized versions. The statistical significance was established when the memoized method demonstrated lower energy consumption than its original counterpart[4]. Each box plot encapsulates the energy measurements, quantified in millijoules (mJ), derived from 25 executions of the respective method. Key components of the box plot include the minimum and maximum values (indicated by the whiskers), potential outliers (denoted by small circles), the 1st and 3rd quartiles (located at the bottom and top of the box), the median (represented by the red horizontal filled-in line), and the average (marked by the blue horizontal dashed line). The yellow/green pairing within each box plot signifies the original/memoized version of the method.

The inclusion of notches in each box plot serves as an informal test of the null hypothesis, indicating whether the medians of the original and memoized versions are statistically equivalent. Specifically, if the notches overlap, it suggests that the null hypothesis cannot be rejected with 95% confidence (Chambers et al., 2017). Further inference of the statistical differences in energy measurements will be expounded upon in Section 4.2.4.

Note that these charts illustrate the energy values resulting from 10 executions of the test suite. The first execution involved new parameter values, while the subsequent executions featured repeated parameter values. Further analysis of varying numbers of executions will be presented in Section 4.2.4.

In Figure 18, box plots depict the energy consumption of the only two methods where the energy expended by the memoized versions exhibits statistical dissimilarity from the original methods, albeit in favor of the original ones. Once again, a more comprehensive examination, involving test executions exceeding 10 iterations, will be elaborated upon in Section 4.2.4.

Figure 19 presents once more box plots, but now for the three methods where no statistical difference was found for original and memoized methods.

For the three methods where no statistical difference was observed, we conducted an extended test suite, varying the number of executions from 10 to 50 times. The objective was to maximize the benefits

[4]The segregation of methods into separate charts is motivated by the distinct scales of values associated with each method. Such segregation enhances the visibility of details across all methods.

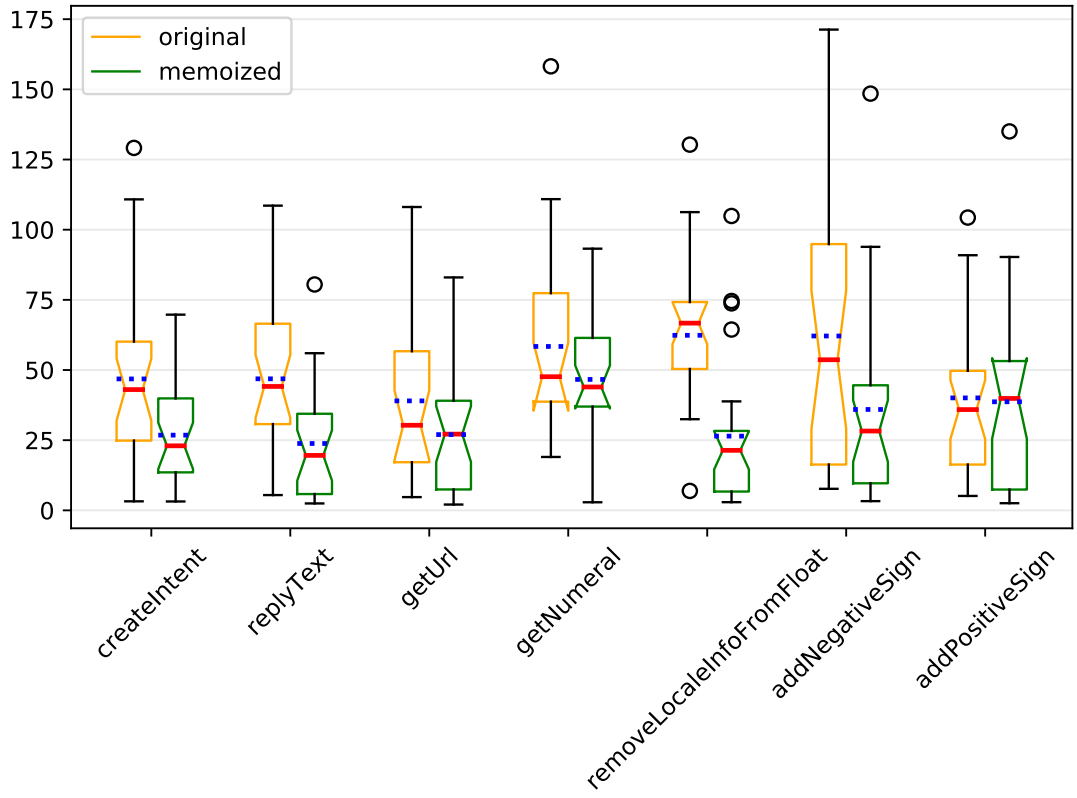
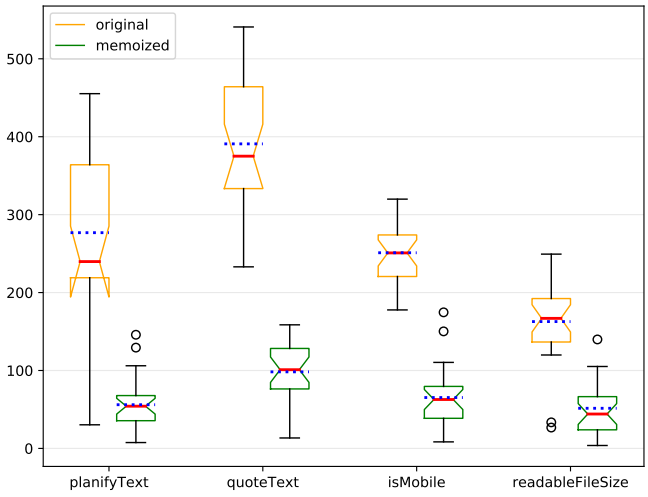
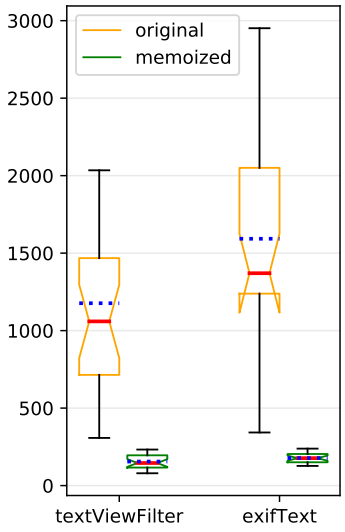


Figure 16: Methods with consumption between 0 and 175 mJ.



(a) Methods with consumption between 0 and 600 mJ.



(b) Methods with consumption between 0 and 3000 mJ.

Figure 17: Box plots representing the energy spent by the original and memoized methods, considering 10 executions of the tests, where the memoized version spent less energy.

of memoization and ascertain whether this would consistently improve or potentially degrade the performance of the method. With each incremental set of 10 executions, the memoized method solely retrieves

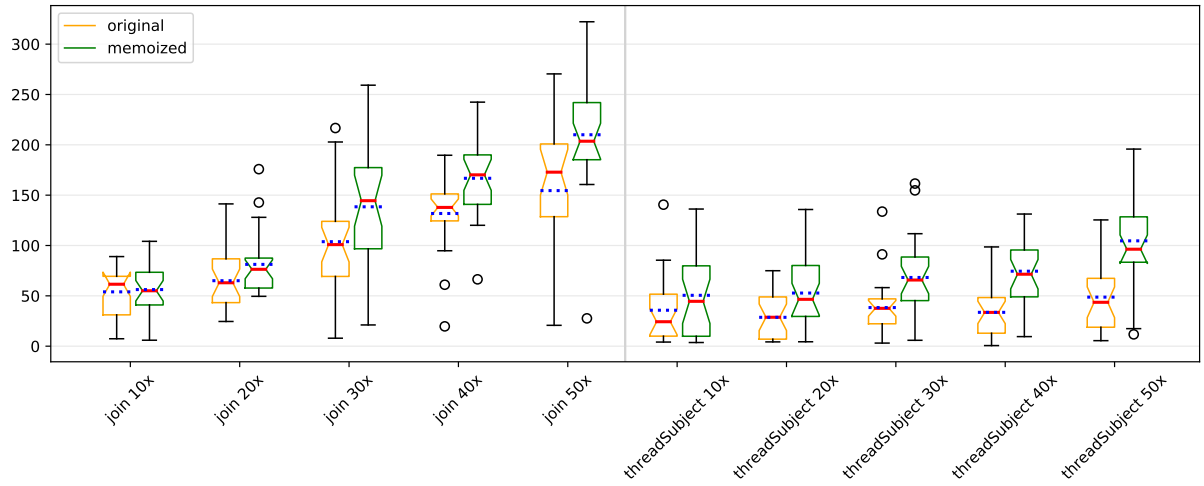


Figure 18: Box plots representing the energy spent by the original and memoized methods, considering 10 to 50 executions of the tests, where the original version spent less energy.

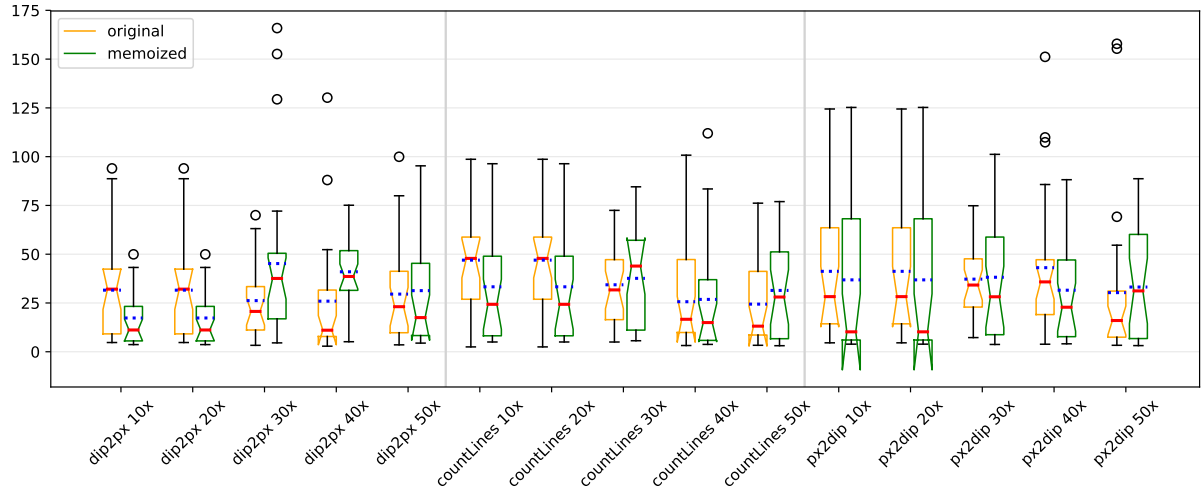


Figure 19: Box plots representing the energy spent by the original and memoized methods, considering 10 to 50 executions of the tests, where no consistent statistical difference among the different runs of tests was found for original and memoized methods.

stored values, while the original version necessitates complete execution. However, the recorded outcomes for these methods indicate variability, wherein the memoized version is occasionally superior, but at times the original version exhibits comparable or superior performance. A more in-depth exploration of these observations will be revisited in Section 4.2.4.

The cumulative results presented thus far suggest a positive impact on energy consumption when memoizing methods.

In Figure 20, the percentage of energy savings is delineated for methods where memoization yields positive results (methods represented in Figure 17). To calculate the percentage energy decrease for each method, we independently sorted the original and memoized energy consumption values for each

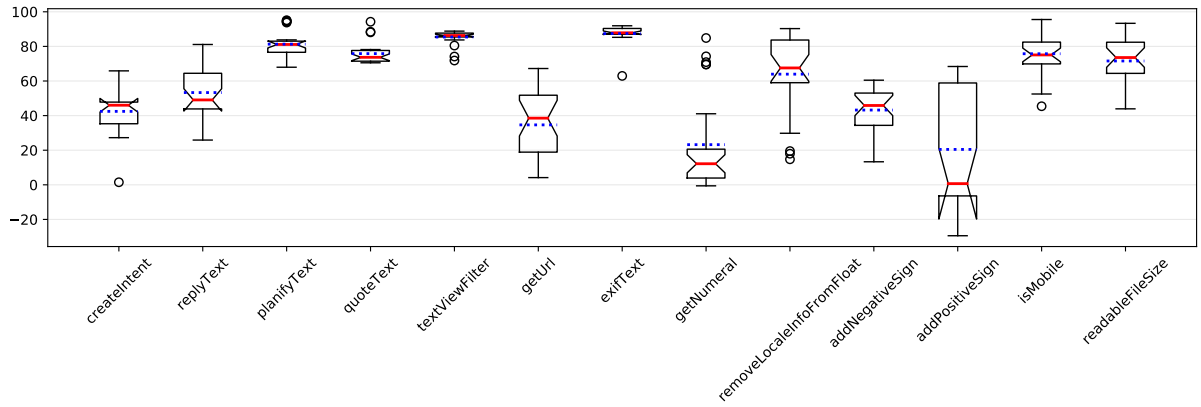


Figure 20: Energy losses from the original method to the memoized one of the methods that have positive results.

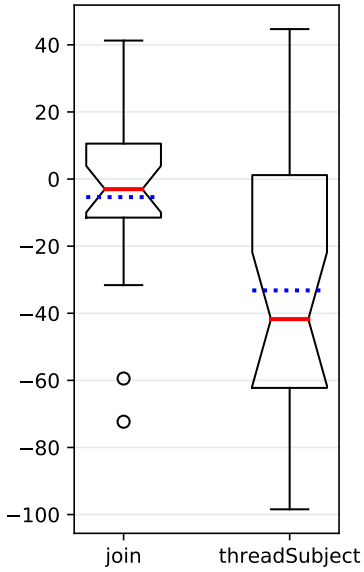


Figure 21: Energy losses from the original method to the memoized one of the methods that have negative results.

of the 25 executions. The formula used is $\frac{\text{original}-\text{memoized}}{\text{original}} \times 100$. Positive values signify a reduction in energy consumption, endorsing the efficacy of memoization. Negative values, on the other hand, indicate an increase in energy expenditure, suggesting instances where memoization may not result in energy savings. The prominently positive gains, even with a test suite executed 10 times, provide initial insights.

As previously observed, two methods exhibited unfavorable results when subjected to memoization. Figure 21 illustrates the percentage of losses, which in this context primarily represent gains, signifying an increase in energy consumption from the original to the memoized version. These outcomes underscore situations where memoization did not yield a reduction in energy expenditure.

Finally, as before, in Figure 22, we present the values for the three methods where there is no clear

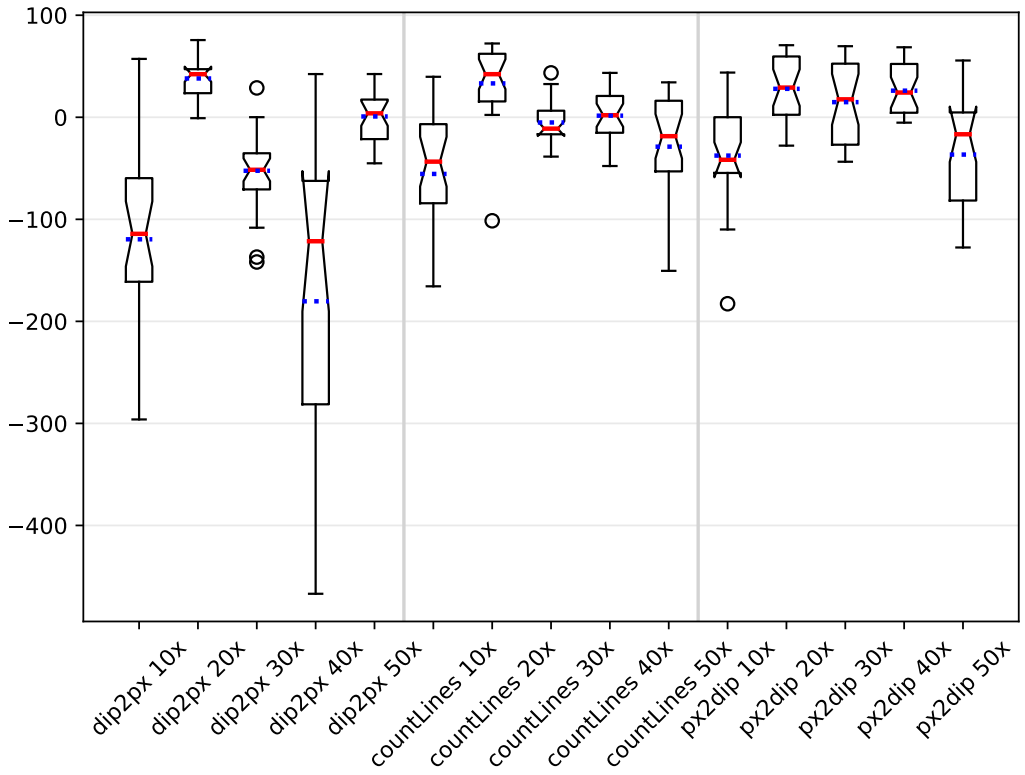


Figure 22: Energy losses from the original method to the memoized one of the methods that have no clear positive or negative direction.

indication of losses or gains, considering 10 to 50 executions of the test suite.

To enhance the interpretability of these results and provide a concise summary, Table 9 delineates the percentage of energy savings by comparing the memoized method to the original one for the 1st and 3rd quartiles. These values correspond to the central tendencies of the losses observed in the box plots.

Finally, Table 10 provides a comprehensive overview of the percentage of instances where the energy expended by the memoized method is lower than the original one. This analysis spans 25 runs for each method, varying from 10 to 50 executions of the test suite. Notably, underlined values denote cases where no statistically significant difference was identified between the original and memoized measurements. For all other methods, a statistically significant difference in energy consumption between the original and memoized versions was observed. Values in **bold** highlight instances where more than 50% of the 25 runs resulted in higher energy expenditure for the memoized methods compared to the original ones. Values without special formatting indicate statistical significance in favor of the memoized versions.

Method	**10x**	**20x**	**30x**	**40x**	**50x**
exifText	(87, 90)	(92, 95)			
textViewFilter	(85, 88)	(94, 94)			
planifyText	(77, 83)	(86, 89)			
quoteText	(71, 78)	(88, 92)			
isMobile	(70, 82)	(84, 96)			
readableFileSize	(64, 82)	(82, 89)			
removeLocaleInfoFromFloat	(59, 84)	(66, 71)			
replyText	(44, 64)	(-9, 17)	(41, 76)	(41, 71)	(48, 76)
createIntent	(35, 48)	(-21, 6)	(35, 54)	(21, 43)	(49, 66)
addNegativeSign	(34, 53)	(62, 80)	(61, 75)	(68, 94)	(71, 86)
getUrl	(19, 52)	(32, 66)	(45, 57)	(37, 56)	(49, 62)
getNumeral	(4, 21)	(-21, 2)	(9, 30)	(27, 45)	(60, 75)
addPositiveSign	(-6, 59)	(40, 48)	(51, 65)	(36, 57)	(10, 26)
join	(-11, 11)	(-40, -8)	(-46, -18)	(-30, -22)	(-36, -19)
px2dip	(-55, 0)	(2, 60)	(-27, 53)	(4, 52)	(-82, 5)
threadSubject	(-62, 1)	(-99, -54)	(-92, -75)	(-280, -101)	(-219, -102)
countLines	(-84, -7)	(16, 62)	(-17, 6)	(-15, 21)	(-53, 16)
dip2px	(-161, -60)	(24, 47)	(-71, -35)	(-281, -62)	(-21, 17)

Table 9: 1st (on the left-hand-side of the column) and 3rd (on the right) quartiles of the energy losses from the use of the original method to the use of the memoized version.

Table 10: For each method, and for each test setting, we present the percentage (of the 25 runs) that the memoized version consumes less energy than the original one. Underlined values have statistical significance while all the others have; bold values represent more than 50% of the 25 runs spend more energy in the memoized methods than the original ones.

Method	**10x**	**20x**	**30x**	**40x**	**50x**
createIntent	72	56	68	68	72
getUrl	60	72	68	76	76
replyText	68	52	84	92	84
getNumeral	56	56	64	68	84
addNegativeSign	56	88	84	100	100
addPositiveSign	56	72	80	84	80
planifyText	100	100	-	-	-
quoteText	100	100	-	-	-
textViewFilter	100	100	-	-	-
exifText	100	100	-	-	-
removeLocaleInfoFromFloat	80	88	-	-	-
isMobile	100	100	-	-	-
ReadableFileSize	88	100	-	-	-
join	48	**32**	28	**28**	**24**
threadSubject	**40**	36	**20**	**12**	**20**
countLines	**36**	68	48	**44**	44
dip2px	**24**	68	**32**	**28**	56
px2dip	**40**	56	48	64	**44**

4.2.4 Discussion

In this section, we present a detailed analysis and discussion of the results collected in our experiment and reported in the previous section. From a holistic point-of-view, the primary observation is evident: memoization exerts a discernible impact on energy consumption in Android applications. Remarkably, for the majority of scenarios, this impact is notably positive. Across the 18 tested methods, energy consumption consistently decreased for 13 of them, as illustrated in Figures 16 and 17a. Notably, for these methods, the average consumption across the 25 measurements consistently decreased with memoization, as did the minimum and maximum values, and the values for the 1st and 3rd quartiles. Figure 17b specifically highlights the methods where the impact of memoization was most pronounced.

These observations provide a foundation for addressing **RQ4.4.1**. To lend statistical significance to these results, we aim to ascertain whether there is consistent statistical evidence indicating that the energy consumed by the memoized version of these methods is reliably lower than that of the original version. To this end, we formulated and tested the following hypothesis:

$$\boldsymbol{H}_0 : P(A > B) = 0.5$$

$$\boldsymbol{H}_1 : P(A > B) \neq 0.5$$

where B and A represent the act of randomly drawing a value from the set of 25 measurements, with and without using memoization, respectively. Hence, $P(A > B)$ represents, when drawing from both A and B, the probability of getting a value from A (without memoization) larger than the one drawn from B (using memoization). Our null hypothesis is then obtaining a probability of 50% while obtaining one different than 50% consists in the alternative hypothesis.

To assess the overall significant relevance between the distributions of A and B for the 13 methods where memoization led to improvements in energy consumption, we employed the Wilcoxon signed-rank test with a two-tailed p-value and considered $\alpha = 0.01$ for statistical significance. The test was conducted for each of the 13 methods, and the p-value was found to be less than 0.01 for 11 of the 13 cases, indicating significant relevance. The exceptions were `getUrl` and `addPositiveSign`, both exhibiting parametric distributions, and these cases will be excluded for now, with further explanation to follow.

To calculate a non-parametric effect size, Field (Field, 2013) recommends using Rosenthal's formula (Rosenthal et al., 1994) to compute a correlation. The resulting correlation values can then be compared against Cohen's suggested thresholds: 0.1 for small, 0.3 for medium, and 0.5 for large significance.

Out of the 11 scenarios with non-parametric distributions, 8 had calculated effect size values. Specifically, `createIntent` and `replyText` methods demonstrated a medium effect size (0.4), while the remaining methods exhibited a large effect size (0.6). For the remaining 3 methods, identified with normal distributions through D'Agostino and Pearson's test (D'Agostino, 1971), Cohen's d coefficient was calculated to determine the magnitude of the effect size. Sawilowsky's reference thresholds were employed, indicating a small effect size (0.4) for `getNumeral`, a medium effect size (0.6) for `addNegativeSign`, and a large effect size (1.4) for `removeLocaleInfoFromFloat`. These values provide statistical support for asserting that, for these methods, using memoization leads to energy savings.

The effect size values were initially calculated for scenarios where each method was invoked 10 times in each measured test. To explore potential variations, the experiment was repeated with 20, 30, 40, and 50 invocations for each method, resulting in updated significance values and effect sizes (see Table 10). It was observed that all 13 methods deemed prone to memoization maintained more than 50% of the 25 tests in favor of memoization. Despite some instances where the statistical experiment yielded no significance, the percentage of memoization-favorable tests increased with the growing number of calls for all these methods. This pattern persisted even for previously excluded methods, `getUrl` and `addPositiveSign`. Consequently, these 13 methods were categorized as *prone to memoization*.

However, the application of memoization does not universally guarantee energy savings. In some cases, it is challenging to conclusively determine whether memoization results in energy conservation. Three methods in our experiments fit this criterion. The results of these experiments, where each method was called 10 times, are illustrated in Figure 19. Examination of the box plots reveals striking similarity in values between the original version and the version employing memoization.

Upon applying the same statistical experiment to the 13 methods where memoization decreased energy consumption, the results for `dip2px`, `px2dip`, and `countLines` methods were found to be statistically insignificant. This implies that the observed decrease in energy consumption cannot be attributed to either the use of memoization or the original version of the method. Moreover, the percentage of tests in favor of memoization fluctuated around 50%, displaying unpredictable variations as the number of invocations increased. Consequently, these methods were categorized as *unpredictable*.

For the remaining 2 methods (`threadSubject` and `join`), energy consumption actually increased in the majority of cases when memoization was employed. By examining Figure 18, it is evident that the memoized versions of these methods tended to consume more energy when invoked 10 times. Table 10 corroborates this trend, revealing that the percentage of instances where the memoized version consumed

less energy than the original one was consistently below 50% for all test scenarios. A repeated statistical test confirmed that the values were consistently worse for the memoized versions, with significance observed for all methods except `px2dip`. The effect size varied from "small significance" to "medium significance," leading to the categorization of these methods as *unfit for memoization*.

Table 9 provides additional insights into the calculated gains (positive values) or losses (negative values) when using memoization. By comparing the lower/higher energy consumption values of the original and memoized versions, we calculated gains pairwise and presented them in a box plot format. These gains/losses are shown for various scenarios, with the number of invocations per test varying.

As expected, the majority of the values are positive, indicating that memoization is a suitable technique for saving energy in most cases. For the first 7 methods, the impact is particularly significant, as the memoized version consistently exhibits significantly lower energy consumption across all 25 measurements. Repeating the experiment with 20 invocations per test amplifies this impact. The significance values also increase, leading to the categorization of these methods as *strongly prone to memoization*.

For the *prone to memoization* methods, the gains range from 3% to 90%, and these values tend to increase with an increased number of invocations. In scenarios with 20 invocations, excluding cases with negative values due to lack of statistical significance, the gains range from 31% to 96%. For other scenarios (30, 40, or 50 invocations), the values are consistently positive, with maintained significance.

In contrast, for the *unpredictable* methods, the gains are erratic. While the 3rd quartile value is negative (loss), the 1st quartile is positive (gain). Despite this unpredictability, statistical tests indicated no significance for these methods, and as the number of invocations increased, the significance tended to decrease proportionally.

4.2.5 Threats to Validity

Measuring the energy consumption of a mobile device poses inherent complexities (Banerjee & Roychoudhury, 2017). The challenge arises from the difficulty in fully isolating the code or application under measurement. To address this, we executed our application 25 times, allowing for sufficient slack to obtain results that, on average, correspond to the truth with minimal side effects. Additionally, the application was run on a factory-reset device, in airplane mode, and with the lowest brightness level to minimize the energy consumed by the display. As our memoization primarily reduces the computations/-work needed by a program, our evaluation focused on measuring energy savings for the CPU. However, a detailed study on how memoization affects the energy consumption of the screen and communication

mechanisms (known energy-intensive aspects in mobile devices) is warranted.

To ensure the consistency of the retrieved data, both the memoized and non-memoized applications were run in the same environment. Half of the runs started with the memoized version, and the other half with the non-memoized one. This approach instills confidence that the observed differences between the two versions are solely attributable to the usage or non-usage of memoization.

The energy consumption values were obtained using Trepn, a tool developed by Qualcomm, a manufacturer of processors and chipsets for mobile devices. Trepn directly obtains power values from the Power Management Integrated Circuit (PMIC) (Power Management Integrated Circuit) in chipsets, ensuring accuracy (Hoque et al., 2015). The tool has been widely utilized in various works (Hegde et al., 2016; Hu et al., 2017; Ribeiro et al., 2021). Moreover, Trepn exclusively measures energy for the monitored application, ensuring realistic measurements.

It is important to note that our experimentation was conducted on one smartphone and one version of the operating system. While absolute values may differ across smartphones and Android versions, the observed differences between memoized and original versions are consistent. Therefore, significant changes in differences are not expected when altering the evaluation settings.

Another consideration involves the executed tests. Finding suitable tests for Android applications can be challenging (Couto et al., 2015a; Couto et al., 2014; Cruz & Abreu, 2017). To address this, we created a test suite for each method under analysis and executed it varying numbers of times to understand the conditions under which memoization proves beneficial. Our findings indicate that, for most cases, reading a memoized value approximately 9 times is sufficient. If a method is predominantly called with different values, the utility of memoization diminishes.

4.3 A large-scale empirical study on mobile performance: Energy, Run-time and Memory

This Section presents a research effort published in a journal (Rua & Saraiva, 2023a). To the best of the authors' knowledge, we present the largest empirical study on mobile software and source-code performance known to date. As far as we acknowledge, there is no equivalent empirical study, in terms of magnitude, that involved real-world app execution built from source in physical Android devices. This study analyzed 1322 versions of 215 different apps, which implies that 1322 different APKs have been installed and executed, through different tests with different inputs, totaling 27900 executed test scenarios. These

were gathered after filtering 708 apps across 6071 versions based on certain criteria (Section 4.3.1.5), such as if they could be automatically compiled in our workstation with our automatic procedure or executed in the target devices. All analyzed versions were present in the Google Play Store and had their respective source code available in non-archived, open-source repositories. The process of gathering and filtering such a number of compilable and executable apps, through dynamic execution, lasted for several weeks, with a total processing time of approximately 28 days just to execute all tests on the final set of filtered apps.

The present work differs from what exists in the state of the art in several ways: in terms of magnitude and in terms of the procedure and conclusions it aims to draw. Firstly, it is the study of the literature by far that dynamically and automatically analyzes more applications. Taking advantage of its magnitude and the nature of the corpus of applications on which the analysis was based, this study's aim was to analyze and validate on a large scale and in a different context the impact of practices previously identified as inefficient in terms of performance (Google, 2021a; Linares-Vásquez et al., 2014). In addition to these differences, the present study also aims to evaluate aspects not yet addressed by other studies in the literature. Namely, aspects such as the impact of certain changes in the energy consumption of applications (for instance, changes at project building/resources level) and to evaluate differences in the impact in terms of performance of competing libraries not yet analyzed in terms of the considered performance indicators that are widely used in Android.

The remainder of the document is as follows: Section 4.3.1 describes the building blocks used to design the processing procedure followed for gathering the collected apps and automatically analyzing their performance. Afterward, we present and discuss the main results from our study in Section 4.3.2. In Section 4.3.3, we answer the last three research questions presented in Section 4.1 and present several guidelines for developers building performance-friendly apps and the resulting dataset. Finally, Section 4.3.7 presents some of the threats to the validity of our work.

4.3.1 Methodology

This section describes the method followed to collect and process the large corpus of apps used in our empirical study. We start by describing the methodology designed for processing a large corpus of variable sizes of applications, by describing the auxiliary tools used to perform the automatic app testing procedure. Afterward, we describe the respective testing workflow to dynamically execute each app. Finally, we describe the methodology followed for obtaining and filtering the collected corpus.

4.3.1.1 Performance Analysis Tooling

To execute and monitor the performance of such a large app corpus, an automated testing framework was adapted using state-of-the-art tools that have been used in the Android ecosystem to automatically execute apps. This section describes each used analysis tool to build our execution pipeline, justifying the reasons for its use.

The most common technique to trace the execution of an app is to perform code instrumentation. Several techniques have been used in various research works, from the instrumentation of source code (Couto et al., 2015a), bytecode (Liu et al., 2017), and the Android framework itself (Machiry et al., 2013). However, many of these approaches do not support the Kotlin language, which is present in 23,4% of the executed versions. The presence of this language is still problematic for applying outdated static and dynamic analysis tools in studies with significant size (Das et al., 2020). The alternative approach consists of using the jInst (Couto et al., 2015a) tool, which was recently updated to also support Kotlin instrumentation in addition to Java. JInst was used in previous works (Couto et al., 2015a; Rua et al., 2019; Rua et al., 2019) to instrument Android code, insert calls to energy profilers and consequently locate energy hotspots in the source code.

In order to monitor the energy consumption while the apps are running, there are several alternatives used by the Android community (Chowdhury et al., 2019; Couto et al., 2015a; Monsoon, 2021; Nucci et al., 2017a). One of the most accurate and used ones is the Trepn Profiler (Qualcomm Innovation Center, 2016): a software-based artifact developed by Qualcomm that works on any Snapdragon chipset-based Android device. Trepn Profiler allows not only to measure power consumption but also other system resources, such as GPU and CPU load and frequency, sensors/stage usage, among many other hardware components. Furthermore, Trepn was already used alongside source code instrumentation to estimate the energy consumption of Android apps (Rua et al., 2019; Rua et al., 2019, 2020).

While manually testing each app with end users would be the most ideal solution to analyze an app, it is impractical for such a large set of apps. In addition, in order to minimize manual work and possible conflicts with app code, devices, or platform versions, we followed a black-box testing approach. Indeed, the use of testing frameworks to evaluate Android apps' performance is widely reported in the literature (Chowdhury et al., 2019; Hu et al., 2017; Jabbarvand & Malek, 2017; Li et al., 2020; Nucci et al., 2017a). Thus, we chose the following two frameworks: UI/Application Exerciser Monkey (Google, 2021b), and App Crawler (Google, 2021). Both of these exercise and test apps through events performed over the GUI, following a depth-first approach.

The former (Monkey for short) has been used in many previous works (Chowdhury et al., 2019; Hu et al., 2017; Rua et al., 2019) to detect energy/performance-related issues, being the most widely used tool by the researching community to automatically test applications (Choudhary et al., 2015). It allows the generation of pseudo-random streams of user events such as clicks, touches, or gestures, as well as several system-level events. It also allows repeating tests and the respective sequence of events, which is essential to replicate the same work over different apps to draw comparisons among them. The latter, App Crawler, is a recent tool provided by Google that allows developers to crawl app GUIs. This tool considers contextual behavior, being able to efficiently locate app GUI widgets and properly interact with them.

To combine all these systems, we reused the AnaDroid tool (Rua et al., 2019). This tool follows the typical automated testing benchmarking process (Kong et al., 2019a), thus allowing the execution of any Android app while monitoring its energy performance. This tool supports the Kotlin dialect, several energy profilers (Trepn and GreenScaler Chowdhury et al., 2019), and most Android testing frameworks (JUnit-based testing frameworks, RERAN Gomez et al., 2013, monkeyrunner, Monkey, App crawler). AnaDroid computes both dynamic energy-aware performance metrics and static energy-related source code metrics.

Finally, for our physical testing setup, we connected 2 workstations to 2 `LG Nexus 5` running the same customized system image which removes unnecessary apps/services and keeps only a minimal set of apps/processes needed to conduct our study. The main idea is to reduce to the minimum the performance measurements' noise. We chose to consider 2 devices of the same model with the same execution context, in order to parallelize the execution and reduce the execution time of the test process in half. Since the measurements that Trepn profiler estimates are obtained at system level, through this method it is still possible to establish comparisons between results obtained in both devices.

4.3.1.2 Testing Workflow

This section describes the testing procedure to prepare, analyze and execute apps while collecting performance metrics during their executions. This process is fully automated and independent of the app's domain. All developed and re-used tools to build the execution pipeline are publicly available. This pipeline is divided into two phases which will be described next.

Table 11: Auxiliary tools used for Android projects' instrumentation and compilation

Task	Tools/Artifacts
Data gathering sources	AndroZooOpen, GitHub, F-Droid
Code Instrumentation	JInst (JavaParser, Kastree, TrepnLib)
Building	Anadroid, Gradle
API Analysis	Androguard

4.3.1.3 Instrumentation and Building

For the source code instrumentation task of the collected apps of our corpus, we used the JInst tool (Couto et al., 2015a; Rua et al., 2019). Source code written in Java and Kotlin is parsed into an Abstract Syntax Tree (AST), which is then traversed to locate tree nodes corresponding to method calls. At method call nodes we insert the code fragment (subtree) needed to perform energy monitoring at run-time. Because in this study we rely on Trepn to monitor energy consumption, the inserted code consists of calls to TrepnLib (Rua et al., 2019; Rua et al., 2019). A dependency to TrepnLib is added to the building scripts which are also automatically adapted to the context of our running platform. JInst performs method-level instrumentation and uses both JavaParser [5] and Kastree [6] to instrument Java and Kotlin code, respectively.

As a final step of this phase, the AnaDroid tool compiles and builds the app's APKs. To guarantee the correct build of the instrumented applications (both their source code and scripts), we follow a regression test-like approach that detects and corrects a set of typical building errors such as API/dependencies conflicts (Scalabrino et al., 2020), Gradle and Gradle-plugin version mismatching, etc. Finally, we used *Androguard* [7] for detecting the APIs present in the APK built. The set of tools used so far from the application gathering to the compilation stage is presented in table 11.

Figure 23 shows the flow of work performed in every app of our corpus. As can be seen, by the diagram, the workflow used for this empirical analysis is equivalent to the *PyAnaDroid* pipeline presented in Section 3.3, since we used for this study an embrionary version of the tool.

4.3.1.4 Execution and Analysis

[5] Java Parser: https://javaparser.org
[6] Kastree: https://github.com/cretz/kastree
[7] Androguard: https://github.com/androguard/androguard

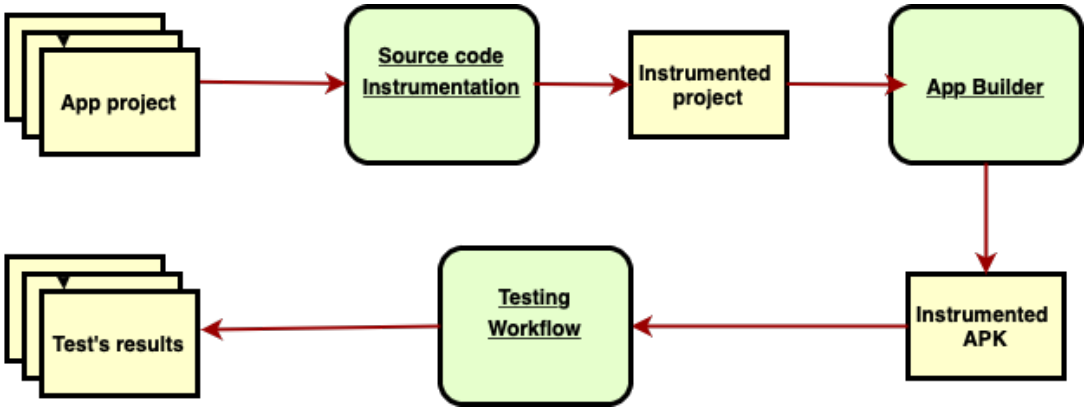


Figure 23: Overview of the execution workflow

After instrumentation and compilation, the generated APK containing the apps' Dalvik bytecode is installed on the Device Under test (DUT). This device is connected to a workstation through the ADB interface, which allows the interaction between each device and the execution of remote actions. Upon installation, the device initially has all hardware sensors (Wi-Fi, GPS, Bluetooth) turned off.

An app, however, may require the use of sensors to be fully functional and allow it to fully explore its core functionalities. In order to minimize external interference to the monitoring process while not limiting apps' execution, we designed a simple heuristic that enables certain sensors according to the permissions specified by each app. Furthermore, for each app, we inspect the *Android Manifest* file to infer the permissions required by the app and we enable sensors whose usage is frequently associated with some permissions before the test execution. The list of permissions considered to activate sensors and the respective sensors are detailed in table 12.

Table 12: List of permissions that activate sensors

Permission	Sensor enabled
ACCESS_FINE_LOCATION	GPS
ACCESS_COARSE_LOCATION	GPS
BLUETOOTH	Bluetooth
BLUETOOTH_*	Bluetooth
INTERNET	Wi-fi
NFC	Near Field Communication (NFC)
NFC_TRANSACTION_EVENT	NFC

Therefore, we guaranteed that at the beginning of each app's execution, the required sensors are

turned on, according to the permissions specified by the the app developers in the manifest file. Controlling the status of these sensors is programmatically done via *ADB*. This makes our process permission-aware (Sadeghi et al., 2017), while reducing testing effort and increasing code coverage.

To automatically run each app, the testing frameworks App Crawler and Monkey were used. Using App Crawler, we performed 3 tests per app, following a completely black-box approach. With the Monkey tool, a set of 25 tests were replicated for each app. The discrepancy between the number of tests considered for each testing framework is due to the fact that tests executed with App Crawler produce more consistent results in terms of method coverage and performance, being its execution more deterministic, unlike what happens with Monkey. The Monkey tests were executed with the same seeds for each application, which leads the tool to generate the same set of events for the apps, given its pseudo-random nature to generate event streams. The system log is cleared before each test is executed and is collected at the end of the run. Log analysis allows the detection and exclusion of executions with errors.

Each test execution, independent of the testing framework, follows the same overall execution workflow, as presented in Algorithm 1.

Algorithm 1: Dynamic Execution workflow

```
for app_project : {apps_projects} do
    for app_project_version : app_project do
        instrumentAppVersion(app_project_version)
        app = buildApp(app_project_version)
        installApp(app)
        for testingFramework : {monkey,crawler} do
            for test : testSet do
                initProfilerService()
                detectAndEnableRequiredAppSensors()
                startApp(app)
                recordDeviceState()
                startProfiler()
                executeTest(test,testingFramework)
                stopProfiler()
                stopApp(app)
                recordDeviceState()
                cleanDalvikArtCache(app)
                exportResults()
    generateResults()
```

In the *initProfilerService* step, the device cache is cleared, so that no data from previous executions

may cause biased results. Afterward, the Trepn service is started, and the app is initialized. The device resource status is collected (number of running processes, CPU and Random Access Memory (RAM) usage, etc) just before starting the monitoring process ('*startProfiler ()*') and the system log is cleaned. This step consists of using Trepn states to temporarily delimit the beginning of the process. The next step consists of starting the testing framework and executing the tests. Finally, the monitoring process is finished and the device status is recorded again. The app is closed, the collected monitoring data is exported to a *csv* file, the system log is extracted, the profiler is turned off, and the data is exported to the workstation.

4.3.1.5 Data Collection and Validation

This section describes the process followed for collecting our corpus of executable applications and the empirical procedure followed to collect the results that are analyzed in the following section. Furthermore, we present the methodology followed to extract and filter the applications used, providing a taxonomy of the resultant set of selected applications. We also present details regarding the procedures followed to dynamically exercise the applications using 2 different testing frameworks.

In order to obtain our final corpus of 215 apps and respective 1322 versions which were analyzed in our empirical study, we resorted to using open-source software repositories. Instead of crawling the most common open-source repositories to search for every Android project, we used the information present in AndroZoOpen (Liu et al., 2020). This repository contains links, metrics, and metadata of Android apps which also have their source code present in open-source repositories such as F-Droid, Github, Gitlab and Bitbucket. To download the apps, we developed a custom crawler that allowed us to extract the URLs for the remote repositories, all released versions, and their source code. Through this process, we identified 1441 apps with 3 or more versions (average of 11 versions/releases per app), across 46 different Play Store categories.

One of the main objectives of this work was to focus on quality apps, avoiding projects which were deprecated/sample, or with minimal and unfinished functionalities. Thus, we only selected apps that were present in the Google Play Store. This assured us a minimum level of peer approval, as to be present in the Google Play Store an app must be approved and meet certain quality and security standards. Although the application admission process is not as rigorous as the Apple App Store process, all applications on the Google Play Store undergo an analysis process that assesses their correctness, security, among others. [8].

[8]Google Play Publishing Checklist: https://stuff.mit.edu/afs/sipb/project/android/docs/distribute/googleplay/publish/

Furthermore, as we also wanted to track the evolution of app source code and have a minimum number of app versions/releases to draw comparisons from, we selected those that had at least 3 versions also present in the repositories.

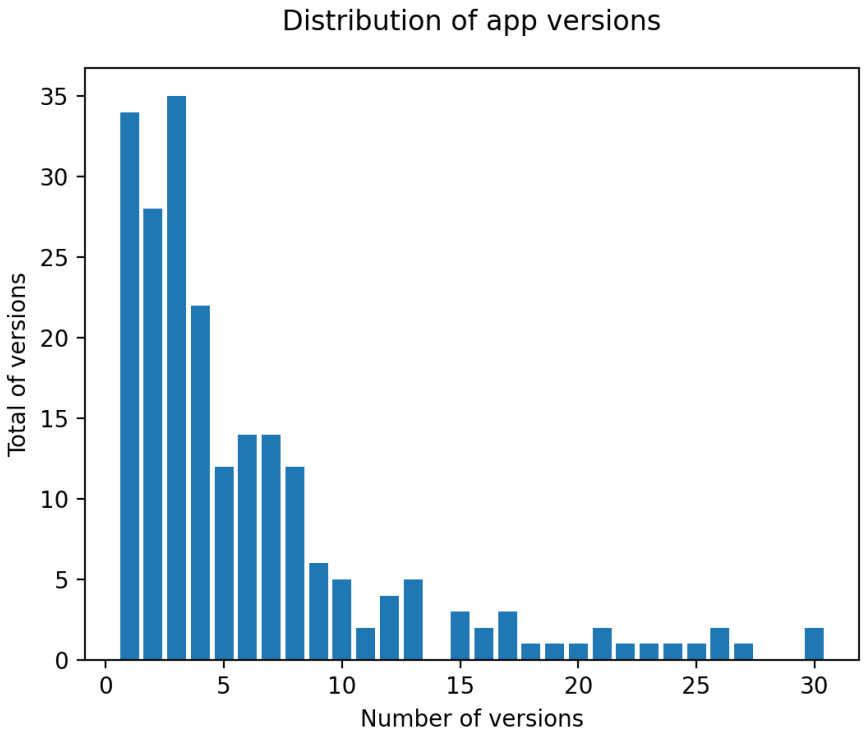


Figure 24: Distribution of number of versions per app

After extracting the source code of each existing version, it was necessary to guarantee that each app was compilable in our development environment and executable on the mobile devices we had at our disposal. Often times developers customize their development environment, some of the available software in open-source repositories only compile on the machines they were originally developed. Since manually adapting the build scripts for each app/version is impractical, to filter the apps that did not compile/execute on our devices, we followed the automatic procedure described in Section 4.3.1.2. Thus, we used AnaDroid (Rua et al., 2019) tool to automate the building process and to execute a simple test: building the APK without any transformation performed, installing the app on the device, and testing whether it launched without errors on the device. After executing this automated procedure, we obtained a set of 565 executable apps containing 2629 versions in total.

Our dynamic execution pipeline, shown in Figure 23, involves an elaborated application analysis process, whose complexity is the accuracy of the automatic execution process. The execution of the

preparing.html

building process outside the typical application development machine/environment can cause failures in the automatic building process. The application instrumentation process changes source code and building scripts, which can cause problems during project building. Including instrumentation libraries can cause incompatibilities between libraries and SDK versions, among other errors. After replicating the previous process of building and installing the APK, but now with the instrumentation process, the percentage of applications fully capable of being processed by our pipeline drops to 270. This reveals that AnaDroid's automated application instrumentation process has a success rate of 47.8%.

To execute the applications we rely on two black-box test frameworks, namely *Crawler* and *Monkey* tools. The former was used to produce a set of 3 tests per version (making a total of 3062 executions) and the latter an average of $17,57$ tests per version (with a total of 23225 executions). After a preliminary experimental setup, we observed that tests executed with Crawler resulted in longer testing sessions. Nevertheless, tests executed with Crawler produced more deterministic results in terms of events generated during its tests and respective method coverage obtained. Furthermore, we only executed this framework 3 times per version. Since the entropy generated by the interaction with Monkey was higher, this framework was executed a more significant number of times, in order to increase the coverage of invoked code.

The execution of the execution pipeline and respective tests allowed us to gather millions of measurements of a large diversified set of metrics (discussed later in Section 4.3.6), obtained through both static and dynamic analysis of the application code and execution of the respective executable/APK. In order to draw fair comparisons among the collected apps, versions and respective tests, for the analysis present in this section we only considered tests performed with Monkey, since the number of events and test duration obtained using App Crawler strongly depends on the app GUI complexity, being considerably divergent among apps. Although the tests performed with Crawler are not analyzed in this section and considered to answer the research questions we intend to answer, their inclusion in this study and the corresponding dataset is still relevant, as we will illustrate in Section 4.3.3.

Each testing framework was invoked via command line, with a defined set of arguments for each executed application. For Monkey, the seeds used to generate the pseudo-random tests can be found in the online appendix. For each test, 1000 events were generated with a delay of 100 ms between them, avoiding system keys that generate events that can interfere with the test process, invoking functions outside the application (volume buttons, return, home, etc). Also, these commands were executed with the –ignore-security-exceptions switch, to avoid stopping the tests when run-time permissions are asked

by the apps under test. Executions of both frameworks were limited by a timeout of 5 minutes (300 seconds), in order to prevent errors that occurred during execution from blocking the execution process. After the defined time, if an error occurred during the execution, the test was discarded. Next, we present both commands used for the execution of the testing frameworks:

```
$ timeout -s 9 300 adb shell monkey -s <monkey_seed> -p <package_name>
--pct-syskeys 0 --ignore-security-exceptions --throttle 100 1000

$ timeout -s 9 300 <path_to_jar>/crawl_launcher.jar --apk-file
<path_to_installed_\gls{APK}> --app-package-name <package>
```

Many of the tests revealed run-time errors or returned corrupted results during test execution. Thus, we needed to analyze such execution in order to discard apps with a 0% method coverage, and tests with invalid values given by the profiler. As a result of this analysis, the set of 270 applications was reduced to 215, containing 1322 different versions.

In order to select a set of results representative of the typical behavior of an Android application, it was necessary to perform filtering to put aside abnormal results that did not reflect such behavior. For this purpose, it was necessary to detect and remove executions in which there were fatal errors in either the execution of the application or the monitoring process. During the execution of the applications, several errors occurred that needed to be filtered so as not to bias our results. Among the criteria used to exclude anomalous executions are App Not Responding (ANR) bugs, test runs that obtained 0% method coverage or null/invalid energy consumption values, which indicates that something problematic occurred during the startup or installation of the application, with the energy profiler or on the device itself. In order to avoid analyzing these problematic executions, we only considered runs where the list of traced methods is not empty, the device logs recorded during the execution did not contain any ANR bugs, and the exported results from the profiler did not contain invalid measurements.

The distribution of the number of apps/releases per year presented in Figure 25 aims to illustrate the diversity of our dataset in terms of the analyzed versions' age, in order to classify the novelty of our dataset and respective applications. According to this analysis, the most common release years are 2018 and 2019, with 70 and 46 applications, respectively. Our dataset has also at least 20 and 40 app versions from 2020 and 2017, having only 1 versions from 2015 and 17 from 2017. This data demonstrates the diversity of our dataset and that our dataset is not composed of legacy or deprecated software.

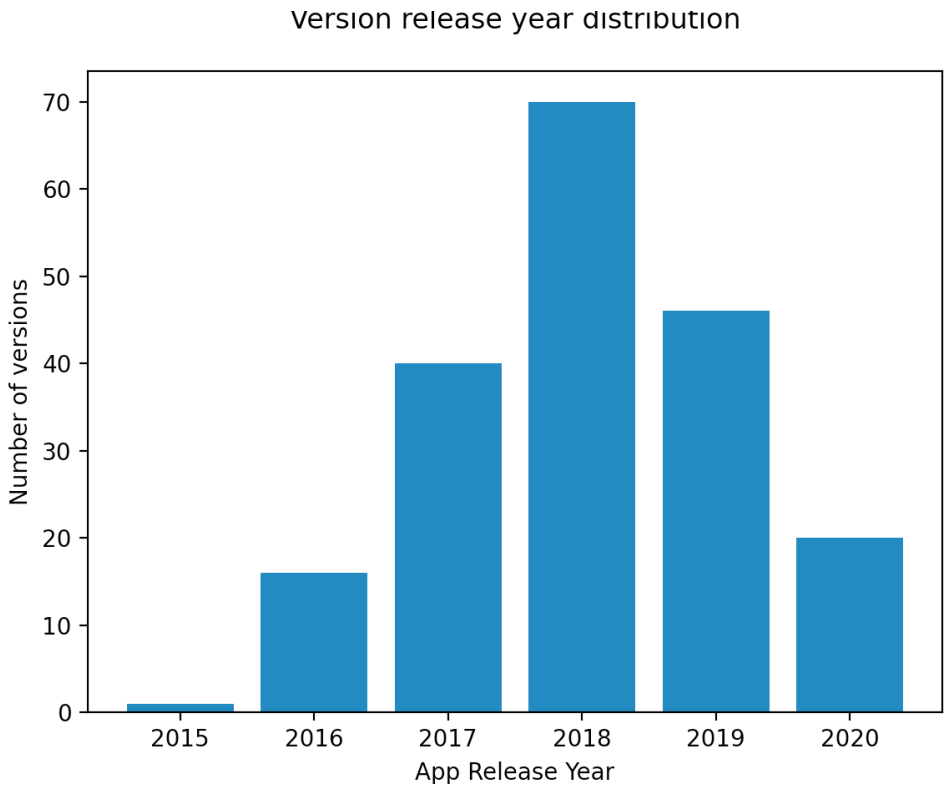


Figure 25: Distribution of number of app/versions per release year

Figure 24 presents a histogram that illustrates the distribution of the number of versions per app. As can be observed, the most common number of versions per app is 3, with 35 apps having such a number of versions. Nearly 85% of the apps executed have 10 or fewer versions executed and 60.9% of the applications have 5 or fewer versions executed. This data demonstrates that our global results are relative to a diverse set of apps, which helps to mitigate the possibility of being influenced by specific apps with a high number of executed versions, since there are very few such cases.

Table 13: Analyzed App Categories

Category	**#Apps**	**Category**	**#Apps**
TOOLS	63	SOCIAL	6
PRODUCTIVITY	23	MUSIC	6
LIBRARIES	20	HEALTH	5
UNKNOWN	20	COMMUNICATION	5
ENTERTAINMENT	9	WEATHER	4
EDUCATION	9	VIDEO	3
GAME	8	OTHERS	34
PERSONALIZATION	7	TOTAL	215

In order to evaluate if our final set of applications is representative of the set of applications present

in the Play Store, we used the Yamane sample size method (YAMANE, 1973). As a reference value, we considered the total number of apps present in the App Store in December of 2021 (the time of writing of this document), 2605000 apps. If we consider this value as the total number of our population, the Yamane method tells us that we would need a sample of approximately 400 applications to obtain a margin of error of 5%, at a confidence level of 95%. Our set of 215 different applications gives a margin of error of approximately 6.68%. However, if we consider the number of versions as distinct applications, assuming that each application corresponds to a unique software artifact and is distinct from the rest of the population, according to Yamane's method our sample would already be representative.

Nonetheless, this (still) large set of apps requires an execution time of more than 28 days to execute every app with the considered Monkey tests. This excludes the time of building and the installation time, setup, warm-up and cool-down times, and exporting results, among others inherent to the proper execution of the automatic procedure described in Section 4.3.1.2.

4.3.2 Large-Scale Android Performance Analysis

This section presents the results and analysis of the execution pipeline presented in Section 4.3.1.2. We performed an empirical study using real devices, in which 2 testing frameworks were executed on a large set of applications. We present the results of executing a set of 215 different apps from the original set of 708, whose filtering process was described in Section 4.3.1.5.

Figures 26 and 27 illustrate the distribution of the average test values for the evaluated performance indicators across the app versions' creation year. The data presented in these Figures allows us to observe that there is no clear trend in the energy consumption of the executed apps. When using as reference the median values of the bloxplots, as well as the lower/upper quartiles and whisker values, our results suggest that the release year does not hint that the performance has been changing throughout the years. Although there is a noticeable increase since 2017 when comparing the medians, the values from the quartiles and whiskers values do not follow the same trend. In terms of memory and run-time, we also observed the same behavior.

As shown in Figure 38 there is a clear decrease in energy consumption when using the *Kotlin coroutines* mechanism. *GreenRobot* also shows gains when compared to both *Executor* and *RxJava*. The most inefficient energy consumption to come from *RxJava*. *Executor* has a substantial gain over *RxJava*, obtaining similar results to *GreenRobot*.

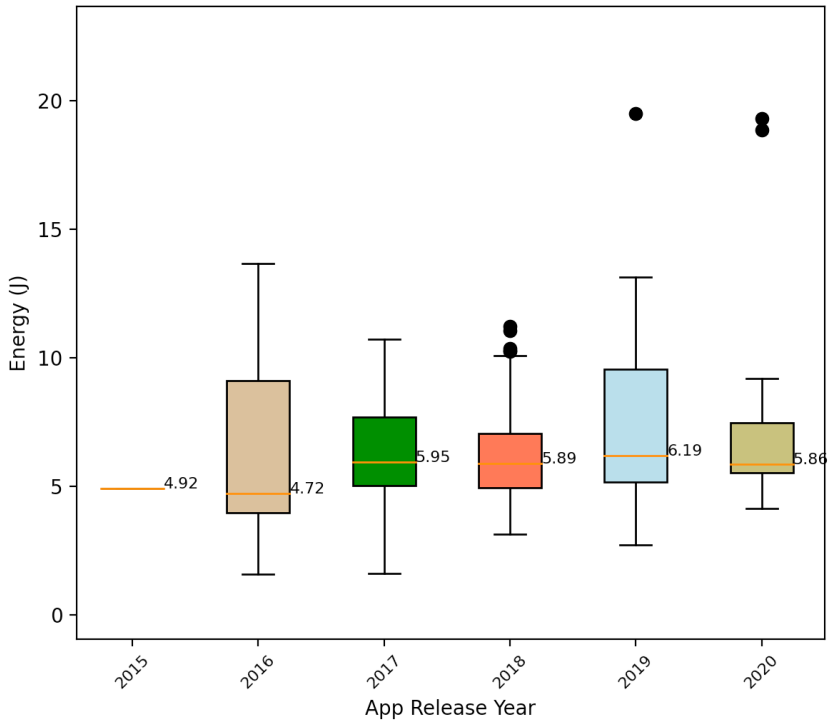


Figure 26: Average Energy consumption across apps versions' release years

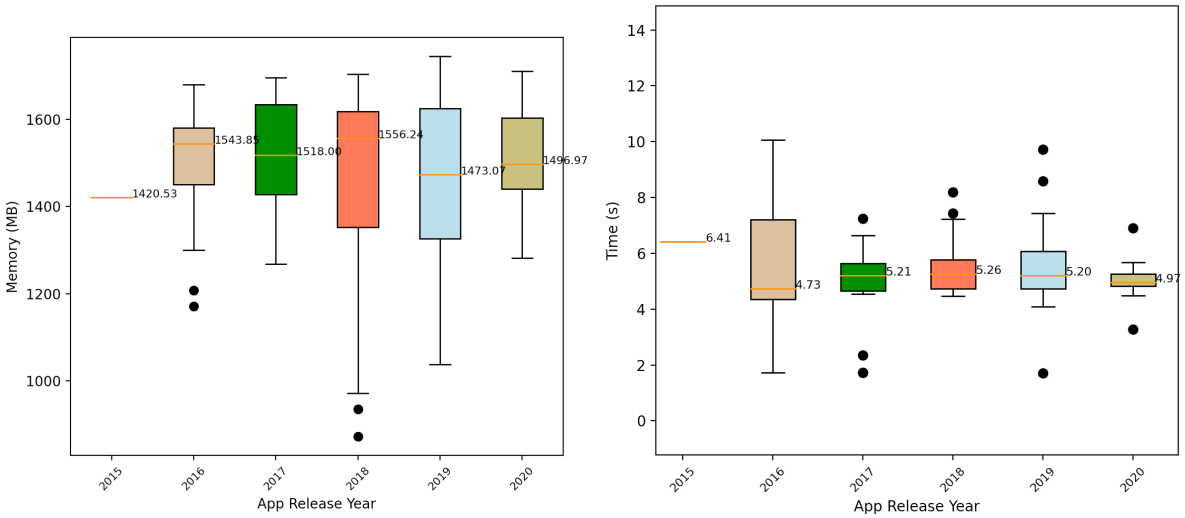


Figure 27: Memory and run-time across apps versions' release years

After filtering all the invalid data, they were grouped in order to answer the research questions presented in Section 4.1. The next sections present the results and analyze the data gathered in order to answer the research questions, which answer is later presented in 4.3.3.

4.3.2.1 Performance Behavior Results

In this section, we present the results and analysis performed to answer **RQ4.3.1:** *Which program changes have more impact on an Android's applications energy consumption?*. In order to detect and categorize significant changes in the energy behavior of the applications as required to answer this question, we have executed Monkey tests across the different versions of the applications. To select applications where the impact on energy may be significant we defined the following set of criteria:

- **1)** Tests executed with Monkey framework. Tests performed with this tool allow the establishment of fairer comparisons between versions since it runs the same set of tests and events for each version and app.

- **2)** Apps with average energy consumption above 5.2 Joules. We considered apps with consumption above the lower percentile of app average consumption;

- **3)** Apps with average method coverage above 5%. In order to ensure that the tests exercised a minimum amount of code from the application.

- **4)** Apps with a variation of at least 20% between consecutive versions concerning the application's maximum consumption. For this purpose, the versions were sorted according to their version number according to the semantic versioning format[9]. This criterion helps to ensure that a certain version has an abnormal performance behavior compared to the adjacent versions and compared to the expected app behavior;

- **5)** Versions with more than 20 app method invocations, in order to avoid considering simple demo apps or tests with little interaction that are not representative of real-world apps or real user interaction.

These criteria were not chosen in order to reduce the domain of analysis but to ensure that only comparisons are made in versions of applications whose consumption is relevant and in which there was a guarantee that the tests mostly capture the performance of the applications and not of the device's system. Since the performance measurements obtained are captured at the system level, in order to impute the measurements with app execution we imposed the criteria to assure that the executions that were considered in our analysis are relevant. Thus, we consider only executions with relevant consumption

[9]Semantic Versioning: https://semver.org

(2) whose metrics were less susceptible to being affected by system events, that executed a relevant portion of the application (3,5). Furthermore, we also ensure that we only draw comparisons between versions with very significant divergence in terms of consumption and with irregular behavior as expected in the application itself(4).

By considering these criteria, we obtain a set of 31 applications and 46 versions where we detect suspicious energy variations. In fact, we detect such suspicious behavior in 14.2% of the executed applications. When we exclude the applications with minimal energy and coverage, which is observed in 52 apps (24.9%), then we observe that 1 in 4 applications has one of its versions with an abnormal energy performance (when compared to the other versions). Figure 28 illustrates the filtering process performed by the previous sequence. Filters **1)** and **5)** from the previous criteria are not illustrated, as they did not remove any application from the set to which they were applied.

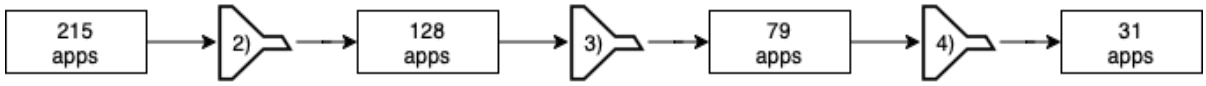


Figure 28: Filtering process followed to answer **RQ4.3.1**

Having identified 46 applications' versions with suspicious energy consumption, we need now to understand the cause of such performance changes. To better understand the changes that are affecting the energy consumption of such app versions, we manually analyze the source code of each of the 46 versions and their predecessor versions.

To perform this manual inspection we analyzed the information available in the open-source repositories of those applications. We discarded two app versions since they are not available on GitHub anymore, leaving us with a final set of 44 transitions between versions to analyze. All other applications were analyzed via the GitHub website GUI that reports the changes made between project commits and releases. Thus, we classified every change occurring in the version with anomalous consumption in relation to the predecessor version. We structured the changes between versions in four types, namely *performance*, *semantic changes*, *project code/resources*, and *project building*, as presented in Table 14.

Performance: We collected the energy, run-time, and memory consumption of the 44 applications and their predecessor versions. Since we wish to understand which app changes influence positively/negatively the performance, for our comparisons we only evaluate if the performance indicator was improved or degraded and we do not consider the magnitude of the change.

Table 14: Criteria considered for the manual analysis

Type	Criteria	Description
Performance	Energy	whether energy went up or down and in what proportion
	Time	whether elapsed time went up or down and in what proportion
	Memory	whether memory went up or down and in what proportion
Semantic Changes	Bugfix	Whether the new version fixes a bug introduced in the previous version
	Refactor	Whether the new version refactors code introduced in the previous version
	Functionalities	Whether the new version introduced new functionalities
Project Code/Resources	Activities/Fragments	If new Activities/Fragments were introduced/removed in app code
	Drawables/Mipmaps	If graphics with bitmaps or XML were introduced/removed
	Layouts / Menus	Whether application UIs or menus were introduced/removed
	Strings	whether string resources were introduced/removed
	Colors/Values	If color or styles were introduced/removed
	DB/Data	Whether resources containing application data such as sqlite databases or JSON files were changed
	Permissions	Whether the app introduced/removed permissions to operate
	#Commits	Number of commits since previous version
	#Files Changed	Number of files changed since last version
Project Building	Dependencies	If new dependencies were added or updated in the project
	Gradle dependencies	If Gradle tools were added or updated in the project
	Android SDK	If Android SDK was updated

Semantic Changes: we consider three categories of code changes that may impact the app's performance, namely, *Bugfix*, *Refactoring*, or *Functionality*. *Bugfix* and *Refactors* were manually identified by looking at changes made to the code of the current and previous versions of the application. *Functionalities* were identified by looking at commit messages between versions, and the introduction of new methods/classes which implement new behavior to the application. If such new code does not introduce new functionality, then we classify it as a *Refactoring*.

Project Code/Resources: We considered changes to *Activities*, *Fragments* and *Permissions*. The first two are the most used classes to build the various "pages"/"screens" that each application can assume during its execution. Thus, a change in activities/fragments may impact the app's performance. Besides changes at the source code level, we also considered changes that affect view components at resource levels, namely layouts/menus and drawables and mipmaps, colors, values and strings. In terms of data to be consumed by the application, we also looked for changes in .db, JSON or text files. Furthermore, we considered the permissions requested by the new versions, so that we could analyze the impact on app performance of these changes. Finally, in a more macroscopic view, we also looked for the number of files changed and commits performed between versions, to correlate such changes with performance measurements. It is important to notice that for these kinds of changes, we analyzed when the project suffered relevant changes, namely, by adding or removing files or by updating existing ones. We did not consider minor changes such as renaming elements, changes in indentation, or fixing typos in strings.

Project Building: For this criteria we considered code dependencies since they have an impact on the performance of programs, and, thus, we evaluate upgrades/downgrades of dependencies of third-party libraries. We also considered changes to the Android SDK since it includes the standard libraries which are frequently updated in such an evolving mobile ecosystem. Finally, we also consider changes in the default building system itself (Gradle and Gradle-plugin versions), since they may affect the generated bytecode.

Having manually extracted this information for every application, we wish now to understand which criteria are affecting the performance of the mobile apps. Because we need to compare two data samples with dichotomous/categorical values, the comparison is presented in tables containing the proportion of agreement between criteria as a contingency table. The tables show the total number and the percentage of times that a criterion simultaneously occurred when a significant change in the application's performance is detected. In these tables, the bolded values on the *Raise* columns signifies that the performance value raised in at least 50% of the app versions (implying a potential degradation of performance). Bolded values on *Drop* columns signifies that the performance value was improved in at least 50% (implying a potential performance improvement) of the analyzed app versions.

In Table 15 we present a comparison between the up/down pairs of performance indicators detected in our 44 applications. Each cell in the table shows the total and percentage of times that a rise or fall in the value of a performance indicator occurred simultaneously with a rise/fall of another performance indicator.

Table 15: Relation between raises and drops of the 3 performance indicators

	Energy raise		**Time raise**		**Memory raise**		**Energy drop**		**Time drop**		**Memory drop**	
	Total	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**
Energy raise	X	X	**19**	**79,2**	**15**	**68,2**	X	X	5	20,8	9	37,5
Time raise	**19**	**79,2**	X	X	**15**	**68,2**	5	25,5	X	X	7	31,8
Memory raise	**15**	**62,5**	**15**	**62,5**	X	X	7	31,8	7	31,8	X	X
Energy drop	X	X	5	20,8	7	31,8	X	X	**15**	**75,0**	**13**	**65,0**
Time drop	5	20,8	X	X	7	31,8	**20**	**100**	X	X	**13**	**59,09**
Memory drop	9	37,5	9	37,5	X	X	**13**	**65,0**	**13**	**65,0**	X	X
Total	24		24		22		20		20		22	

As we can see in Table 15 the greatest relationship between changes occurs between energy and time, either between rises in consumption of both components (79.16%) and between reductions (75% and 100%). This correlation between time and energy was already observed in several studies (Chowdhury et al., 2019; Pereira et al., 2017, 2021a). Although these two performance indicators tend to be highly

correlated, we observe that a rise or drop in one of the components does not mean a change of the same magnitude in the other. It is also possible to see that it is much more common to observe a decrease/increase of all performance indicators at the same time, rather than in a disjoint way. However, a rise in one performance indicator does not always lead to the rise of another component. In fact, they are not perfectly correlated. Regarding energy consumption, its rise was only accompanied by a decrease in run-time or memory in 20.84% and 37.5% of the time, respectively.

Table 16 compares the occurrence of the Project Resources, Project Source and Building Code type criteria with the occurrence of energy rises/drops. The percentage (%) reported for each change leading to a rise and drop is relative to the total of energy raises and drops presented in Table 15 (24 and 20, respectively).

Table 16: Impact of changes in energy consumption

Change	**Energy Raise**		**Energy Drop**	
	Total	**%**	**Total**	**%**
Add Functionalities	**17**	**70,8**	6	30,0
Add Layout	**14**	**58,3**	5	25,0
Add Strings	**3**	**54,2**	8	40,0
Refactor	**13**	**54,2**	**10**	**50,0**
Gradle Update	**12**	**50,0**	5	25,0
Update dependencies	8	33,3	6	30,0
BugFix	7	29,2	**12**	**60,0**
New dependencies	7	29,2	7	35,0
Add Colors/Values	7	29,2	7	35,0
Add Drawables	7	29,2	7	35,0
SDK Update	6	25,0	4	20,0
New Activities	4	16,6	3	15,0
Add mipmap	3	12,5	4	20,0
Update Permissions	2	8,3	5	25,0

As Table 16 shows the criterion that has the strongest impact on the rise of energy consumption is the addition of functionalities. Indeed, in 70.8% of the apps' versions where new functionality was added the energy consumption increased (at least 20%), while in only 30% of them, the consumption decreased. On the positive side, the fixing of bugs shows a reduction of energy consumption in 60% of the applications, and only in 29.2% the energy raised.

To better understand why the addition of *functionalities* and *bugfixes* have a strong impact on energy consumption, let us analyze in detail which other changes occur simultaneously with these two criteria. We start by analyzing the impact on the energy, time, and memory consumption in all apps where new

functionality has been added. This is shown in Table 17.

Table 17: Impact of changes in performance on versions with increased functionality

	Energy Raise		**Energy Drop**		**Time Raise**		**Time Drop**		**Memory Raise**		**Memory Drop**	
Change	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**
Layout	**12**	**66,7**	2	33,33	**8**	**57,1**	**6**	**60**	**8**	**57,1**	**6**	**60,0**
Strings	**12**	**66,7**	**3**	**50,0**	**8**	**57,1**	**7**	**70,0**	**9**	**64,2**	**6**	**60,0**
Refactor	**10**	**55,6**	**6**	**100**	**9**	**64,3**	**7**	**70,0**	6	42,9	**10**	**100**
Update Gradle Dependencies	**10**	**55,6**	**3**	**50,0**	**8**	**57,1**	**5**	**50,0**	**8**	**57,1**	**5**	**50,0**
Update Dependencies	7	38,9	2	33,3	5	35,7	4	40,0	4	28,6	**5**	**50,0**
Styles/Colors/values	8	44,4	**3**	**50,0**	6	48,9	**5**	**50,0**	6	42,9	**5**	**50,0**
Drawable	6	33,3	**3**	**50,0**	3	21,4	**6**	**60**	4	28,6	**5**	**50,0**
New dependencies	6	33,3	**3**	**50,0**	5	35,7	4	40,0	5	35,7	4	40,0
BugFix	4	22,2	2	33,3	2	14,3	4	40,0	5	35,7	1	10,0
New Activities/Fragments	4	22,2	2	33,3	3	21,4	3	30,0	3	21,4	3	30,0
Permissions	2	11,1	**4**	**66,6**	3	21,4	3	30,0	3	21,4	3	30,0
Mipmap	2	11,1	**3**	**50,0**	2	14,3	3	30,0	1	7,1	4	40,0

As we can see in Table 17 the changes that more often occur when adding new functionality are *Layout* and *String*, *Refactor* and *Update to Gradle Dependencies*. *Layout* does contribute to the rise in energy in 66.7% of the apps. This change also shows an impact on run-time and memory consumption. Such impact, however, is in opposite directions since it is (almost) equally divided between time/memory rises and drops.

String and Gradle dependency updates end up having a little distinct impact when accompanied by features, occurring with a frequency equal to or greater than 50% in drops and rises of all components of the analyzed app versions. As for *refactorings*, they are associated with both rises and falls in energy and time. However, the impact is more accentuated in the descents of all components, and all such descents in energy and memory with refactors happened when accompanied by the addition of *functionalities*. Let us analyze now the occurrence of *bugfixing* changes as shown in Table 18.

There are a few criteria that occurred concurrently with a significant performance impact. This can be justified by the fact that most *bugfixes* are quick changes that do not involve major changes to Android code and building resources. All occurrences with a significant impact had a negative impact on performance, with the Layout changes having a negative impact on all performance indicators and the addition of functionalities with a negative impact only on energy consumption. These changes were already the changes with the most negative impact on energy consumption, as presented in Table 16.

4.3.2.2 Android Programming Practices

Table 18: Impact of changes in performance on versions with *bugfixes*

	Energy Raise		**Energy Drop**		**Time Raise**		**Time Drop**		**Memory Raise**		**Memory Drop**	
Change	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**	**Total**	**%**
Layout	**6**	**85,7**	1	9,1	**4**	**57,1**	3	27,3	**6**	**54,6**	1	14,3
Functionalities	**4**	**57,1**	1	9,1	2	28,6	3	27,3	5	45,5	0	0
Strings	3	42,9	2	18,2	2	28,6	3	27,3	2	27,3	2	28,6
Update Dependencies	2	28,6	3	27,2	2	28,6	3	27,3	4	36,4	2	28,6
Update Gradle Dependencies	2	28,6	3	27,3	1	14,3	4	36,4	4	36,4	1	14,3
Drawable	1	14,3	2	18,2	0	0	2	27,3	2	18,2	1	14,3
New dependencies	1	14,3	3	27,3	0	0	4	36,4	2	18,2	2	28,6
Permissions	1	14,3	3	27,3	1	14,3	3	27,3	2	18,2	2	28,6
Mipmap	1	14,3	0	0	1	14,3	0	0	1	9,1	0	0
Styles/Colors/values	1	14,3	1	9,1	1	14,3	1	9,1	1	9,1	1	14,3
New activities / Fragments	1	14,3	0	0	0	0	1	9,1	1	9,1	0	0
Refactor	0	0	4	36,4	0	0	4	36,4	2	18,2	2	28,5
SDK Update	0	0	5	45,5	0	0	5	45,5	3	27,3	2	28,6

This section presents the procedure and results obtained to answer **RQ4.3.2:** *Do the previously identified inefficient programming practices behave the same with larger data sets and different contexts?*. To answer this question, we analyzed 2 programming practices widely studied as energy inefficient in the literature: The energy greedy API (Linares-Vásquez et al., 2014) and then Android performance problems Lint (Google, 2021a). An analysis of these practices is presented in Sections 4.3.2.3 and 4.3.2.4, respectively.

4.3.2.3 Energy Greedy APIs

In the context of green software, one of the most relevant studies was presented in 2014 by Linares *et al.* (Linares-Vásquez et al., 2014) where several energetically greedy APIs were identified in the Android platform. The Android ecosystem, however, has drastically evolved since this study was performed. Consequently, those results may not report the current performance of Android APIs. Moreover, it is also important to study those APIs within a large set of applications representing the the current state-of-the-art of software development in Android.

In this section, we analyze in detail Linares *et al.* energy-greedy APIs, the so-called red APIs. Although the original study reported a set of 129 APIs, in our repository we detected the use of 60 different red APIs only. The APIs that each method invokes are collected with *Androguard* from the Dalvik bytecode contained in the apps' APK. In order to determine if a certain test invoked a certain red API, we verify if each app method invoked during a test contains the red API in its code. Table 19 summarizes the 60 red APIs found in the analyzed apps. For each API we associate the number of unique apps, versions, tests and methods where it is called.

Table 19: Red APIs invoked in Monkey tests

Red API	#Apps	#Versions	#Tests	#Methods
Global	161	852	27674	35802
android.widget.TextView.setText	117	589	4239	15414
android.widget.Toast.show	74	250	1996	3414
android.widget.Toast.makeText	72	262	2071	3492
android.util.Log.e	54	260	2455	14404
android.database.sqlite.SQLiteDatabase.execSQL	27	123	846	1973
android.util.Log.i	24	85	471	1804
android.widget.ImageView.setImageResource	22	93	524	1878
android.content.Intent.getIntExtra	20	104	375	767
android.database.sqlite.SQLiteDatabase.rawQuery	19	76	442	1226
android.database.sqlite.SQLiteDatabase.query	18	68	388	447
android.view.ViewGroup.addView	17	62	416	424
android.app.Activity.findViewById	14	69	419	734
android.graphics.Bitmap.createBitmap	14	85	277	309
android.database.sqlite.SQLiteDatabase.insert	10	47	503	845
android.view.View.setEnabled	10	26	216	426
android.content.res.Resources.openRawResource	9	50	323	326
android.graphics.Canvas.drawText	7	51	470	974
android.widget.EditText.setSelection	7	15	53	53
android.webkit.WebView.loadUrl	7	23	68	87
android.view.View.startAnimation	6	20	191	409
android.database.sqlite.SQLiteDatabase.delete	6	39	306	462
android.view.View.setLayoutParams	6	14	209	1080
android.content.ContentResolver.query	6	20	231	475
android.database.sqlite.SQLiteQueryBuilder.query	6	20	231	251
android.widget.ArrayAdapter.notifyDataSetChanged	5	11	30	30
android.webkit.WebView.loadData	4	8	0	0
android.widget.ProgressBar.setMax	4	35	182	211
android.database.sqlite.SQLiteDatabase.update	4	27	195	196
android.webkit.WebView.loadDataWithBaseURL	4	8	57	57
android.app.NotificationManager.cancelAll	3	10	24	24
android.view.ViewGroup.removeAllViews	3	8	7	7
android.app.Dialog.show	3	6	26	26
android.database.sqlite.SQLiteDatabase.openDatabase	3	4	3	3
android.app.Activity.finish	3	18	271	481
android.util.Log.getStackTraceString	3	15	201	358
android.app.Dialog.dismiss	3	8	4	4
android.view.View.setClickable	2	4	1	4
android.database.sqlite.SQLiteDatabase.insertOrThrow	2	2	26	26
android.text.format.DateFormat.getDateFormat	2	3	1	1
android.text.format.DateFormat.getTimeFormat	2	11	112	114
android.database.sqlite.SQLiteDatabase.endTransaction	2	15	133	236
android.widget.ArrayAdapter.clear	2	8	150	150
android.app.Service.onStartCommand	2	22	214	214
android.database.sqlite.SQLiteOpenHelper.getWritableDatabase	1	8	0	0
android.app.Activity.startActivityForResult	1	12	207	216
android.view.Window.findViewById	1	4	7	14
android.location.LocationManager.getGpsStatus	1	2	2	2
android.view.View.performClick	1	2	23	23
android.database.sqlite.SQLiteDatabase.openOrCreateDatabase	1	1	1	1
android.os.Handler.dispatchMessage	1	12	12	12
android.graphics.Bitmap.getPixel	1	6	71	71
android.graphics.BitmapFactory.decodeStream	1	2	2	2
android.content.ContentResolver.update	1	2	20	23
android.app.Activity.setContentView	1	4	4	4
android.database.sqlite.SQLiteStatement.executeInsert	1	2	50	50
android.database.sqlite.SQLiteOpenHelper.getReadableDatabase	1	8	0	0
android.database.sqlite.SQLiteDatabase.getVersion	1	1	1	1
android.telephony.TelephonyManager.getPhoneType	1	1	1	1
android.graphics.Paint.getTextBounds	1	4	4	5
android.view.ViewConfiguration.getLongPressTimeout	1	8	69	69

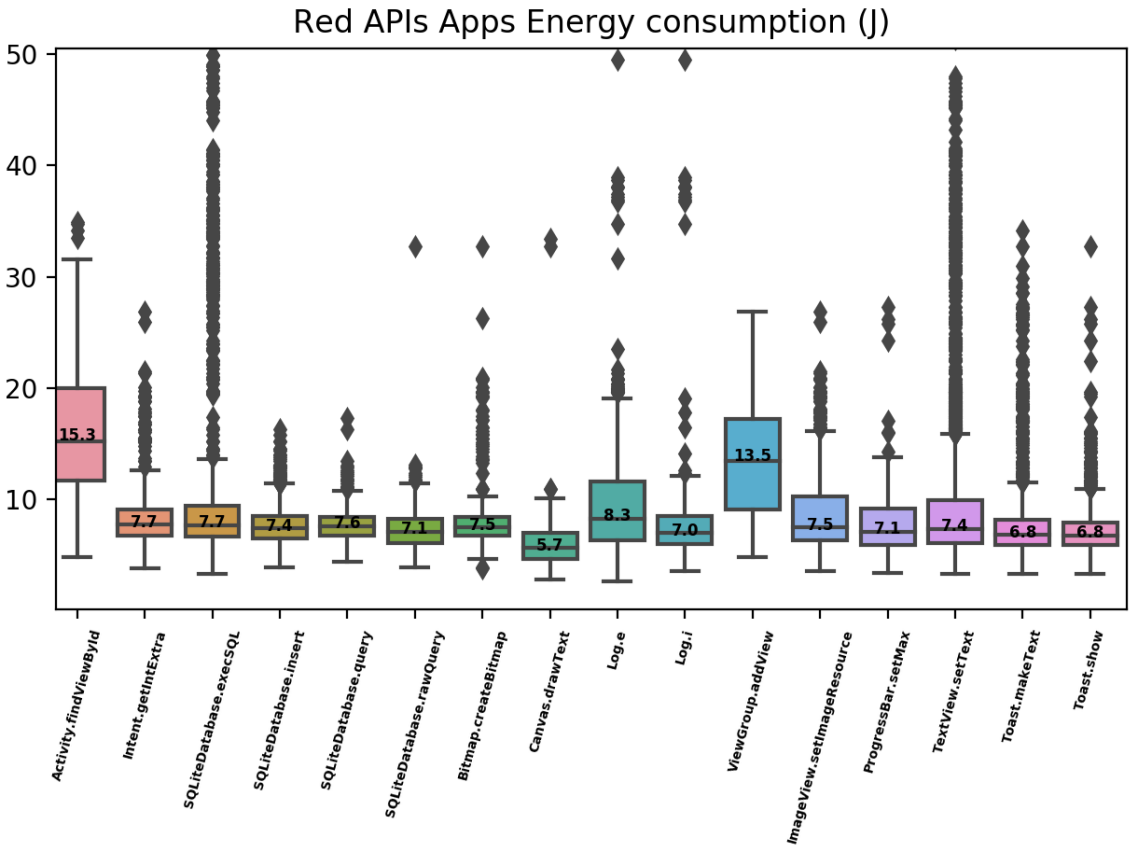


Figure 29: Energy consumption of tests with Red APIs

Next, we present results regarding energy, memory, and run-time associated with the invocation of several red APIs. Our results include the APIs that were executed by more than 100 test cases, across at least 25 different versions of 10 or more different apps. Figure 29 shows a comparison in terms of energy consumption (in Joules, Y axis) of such red APIs. As we can immediately see, several APIs appear to have a strong negative impact on energy consumption, such as *findViewById* and *addView*.

In order to assess the impact of directly invoking these red APIs on app energy consumption, we present in Figure 30 the energy consumption of the tests (of the same apps) that did not exercise those APIs. As we can see, the most notable energy footprint is shown by the same *findViewById* and *addView* APIs. By looking at the medians, the value of *findViewById* for example more than doubles most of the medians of all APIs illustrated in the figure, except for *addView* and *Log.e*. Comparing *findViewById* with the API with lower consumption, *drawText*, its consumption in terms of the median is 2.58 times higher.

If we look in more detail at Figures 29 and 30, we see that there are red APIs whose impact on energy consumption is negligible, or more surprisingly the energy consumption decreases. For instance, just by looking at the medians, tests now not containing *drawText*, *createBitmap* and *rawQuery* have increased

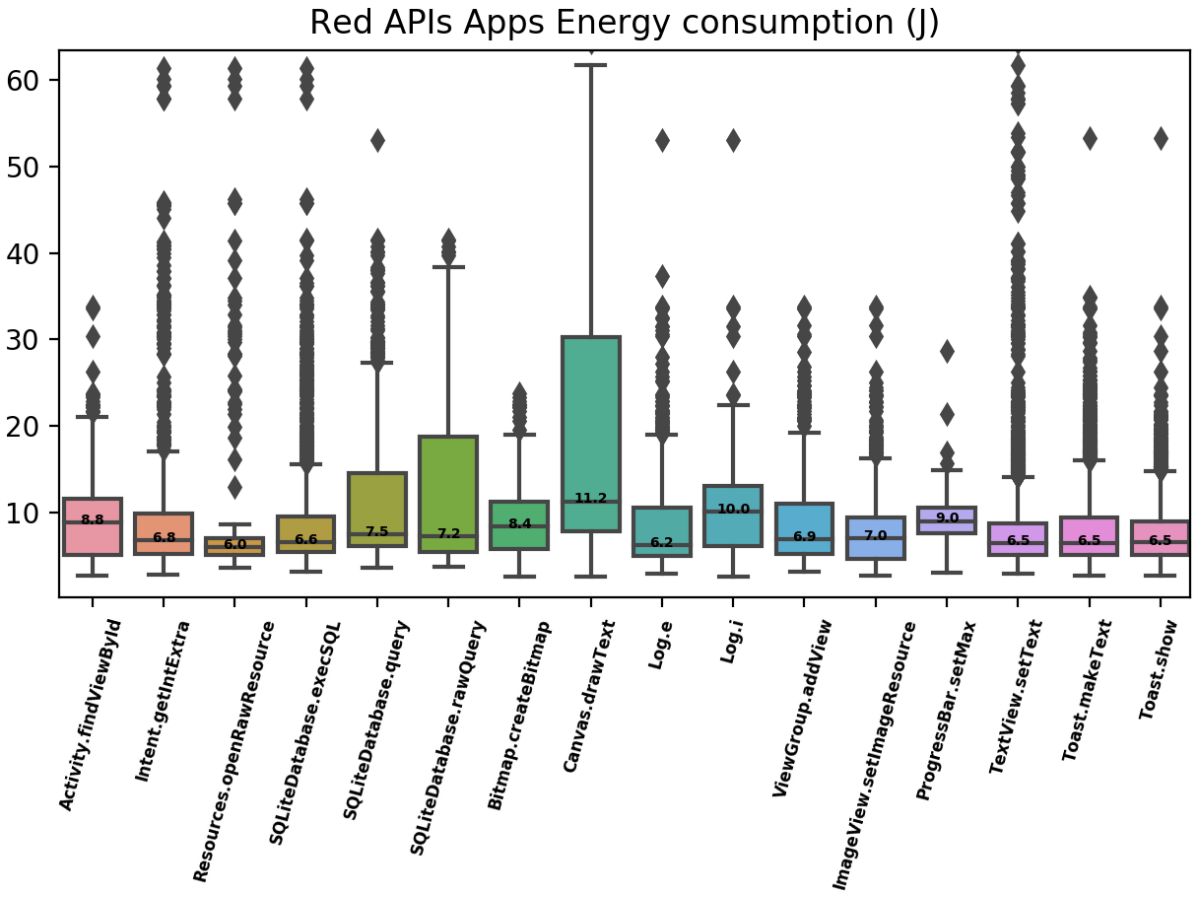


Figure 30: Energy consumption of tests without Red APIs

their energy consumption. More particularly, tests not containing *drawText* increased their energy consumption by nearly 96.5% when comparing the medians. To clarify these issues, we perform a statistical test on all pairs with and without red APIs to verify if there are statistically significant differences in our results. We begin by evaluating whether the values of the tests for each API followed a parametric distribution. After rejecting this hypothesis for all the APIs, through both Shapiro (Shapiro & Wilk, 1965) and D'Agostino-Pearson (D'Agostino, 1971), we apply the Man-Whitney (Mann & Whitney, 1947) U test to compare pairs with/without the invocation of each Red API. In order to evaluate if there are significant and statistically supported differences between the pairs, we evaluate the following hypothesis for each performance indicator under analysis: $\boldsymbol{H}_0$: The values obtained for tests with or without Red APIs invocations follow the same distribution; $\boldsymbol{H}_1$: The values obtained for tests with or without Red APIs invocations do not follow the same distribution.

The results of these hypothesis tests are shown in Table 20 (left columns), as well as a classification of each API as *Red* or not, obtained by comparing the medians of the pairs with and without Red API. If the tests using the API were less efficient than the ones not using it, it was classified as *Red* (marked by ✓).

Otherwise, the APIs are marked with ✗ and didn't evidence performance-greedy behaviour in our context. In regards to energy, the Mann-Whitney U test rejected H0 in practically all pairs, except for the *rawQuery* and *createBitmap* methods. This represents the case where we could not draw any conclusions regarding its performance (marked by =). This shows, with statistical support, that there are indeed differences in energy consumption when using, or not using, these APIs. However, based on the median and average values, the tests using the *drawText* and *Log.i* APIs show that such differences might sometimes even be positive within apps. In fact, we see that calling these two APIs does reduce energy consumption, thus, making their classification as red API invalid in our context.

Table 20: P-values and resultant Hypothesis of assessing if there were differences in test performance using Red APIs

	Energy			**Memory**			**Time**		
API	**p-val**	**Hypot.**	**Red?**	**p-val**	**Hypot.**	**Red?**	**p-val**	**Hypot.**	**Red?**
widget.TextView.setText	0	H1	✓	0.426	H0	=	0	H1	✓
graphics.Canvas.drawText	0	H1	✗	0.012	H1	✗	0	H1	✗
graphics.Bitmap.createBitmap	0.36	H0	=	0	H1	✗	0.058	H0	=
widget.ImageView.setImageResource	0	H1	✓	0	H1	✓	0.069	H0	=
app.Activity.findViewById	0	H1	✓	0	H1	✓	0	H1	✓
view.ViewGroup.addView	0	H1	✓	0	H1	✗	0	H1	✓
util.Log.e	0	H1	✓	0	H1	✓	0	H1	✓
sqlite.SQLiteDatabase.query	0.349	H0	=	0	H1	✗	0.436	H0	=
widget.Toast.makeText	0	H1	✓	0	H1	✓	0.007	H1	✓
util.Log.i	0	H1	✗	0	H1	✓	0	H1	✗
content.Intent.getIntExtra	0	H1	✓	0	H1	✗	0	H1	✓
sqlite.SQLiteDatabase.rawQuery	0.005	H1	✗	0	H1	✗	0.188	H0	=
widget.Toast.show	0	H1	✓	0	H1	✓	0.279	H0	=
widget.ProgressBar.setMax	0	H1	✗	0	H1	✗	0	H1	✓
sqlite.SQLiteDatabase.execSQL	0	H1	✓	0	H1	✓	0	H1	✓

In Figure 31 we present results of the memory consumption of the same APIs. Once again, we see considerable differences in memory consumption in these APIs. The API methods *findViewById* and *addView*, which were the most energetically inefficient ones, have shown a memory consumption consistently above most of the others. Moreover, the *Log.i*, *Log.e* and *setImageResource* APIs are also the most memory-demanding ones, placing their median and 1st/3rd quartile values significantly above the others. Similarly to energy consumption, we have also compared tests that invoked red APIs with tests that did not invoke such APIs. Memory consumption does not follow a parametric distribution when using the previously defined tests. After evaluating the hypotheses, only the *setText* API does not show significant statistical differences between the invocation and non-invocation of the Red APIs. We conclude that the invocation of these red APIs does in fact have a large impact on memory usage.

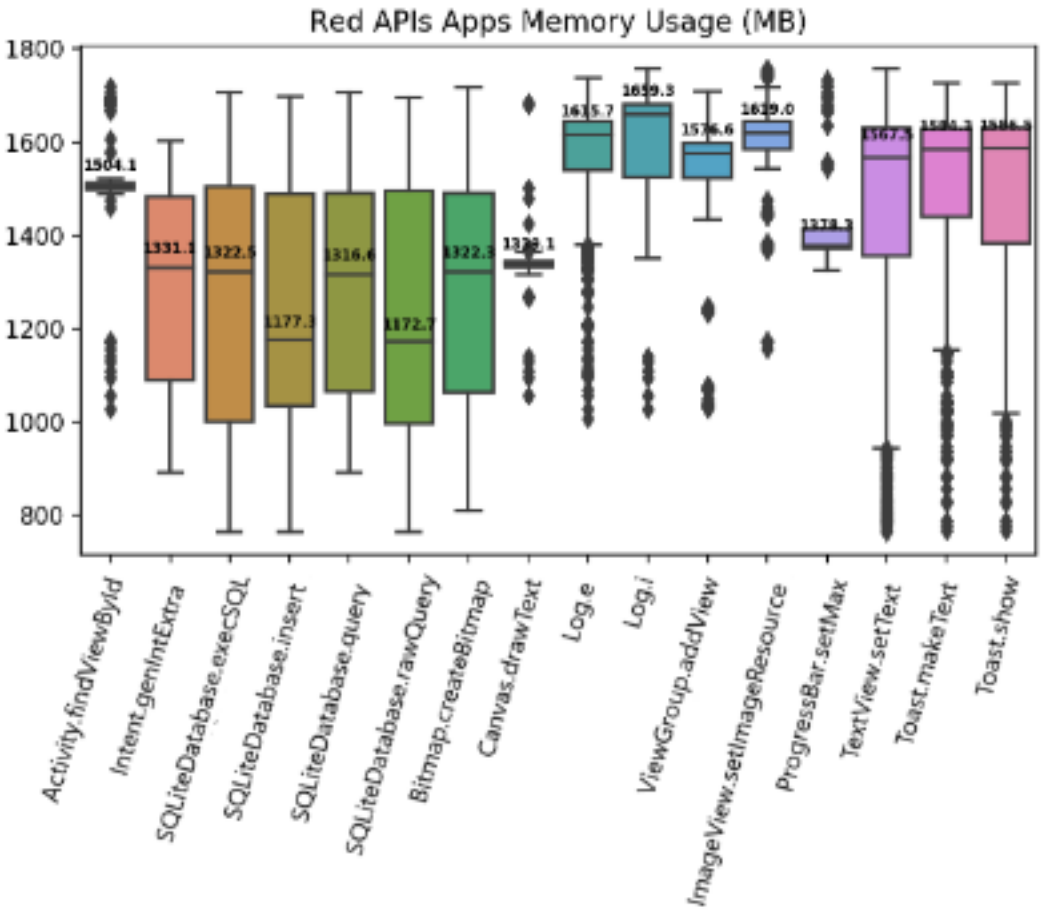


Figure 31: Memory used by tests with Red APIs

Figure 32 presents the results of our third performance indicator: run-time. The *findViewById* and *addView* APIs stand out negatively from the other red APIs considered. The former presents tests with median values twice above that of many other red APIs (15.3 Joules), with the latter standing out from the others, but not in the same magnitude (values between 30 to 40% higher in terms of median). Execution time is the measured performance indicator in which the Red APIs demonstrate less impact during our performed tests. The results in Table 20 (rightmost columns), for the evaluation of H0 and H1, show that there are 5 Red APIs that do not show statistical differences in terms of execution time with their invocation.

4.3.2.4 Android Lint Issues

Android Lint is a static code analysis tool that checks your Android project source files for potential bugs and performance issues. Such performance issues have been used in several research studies which

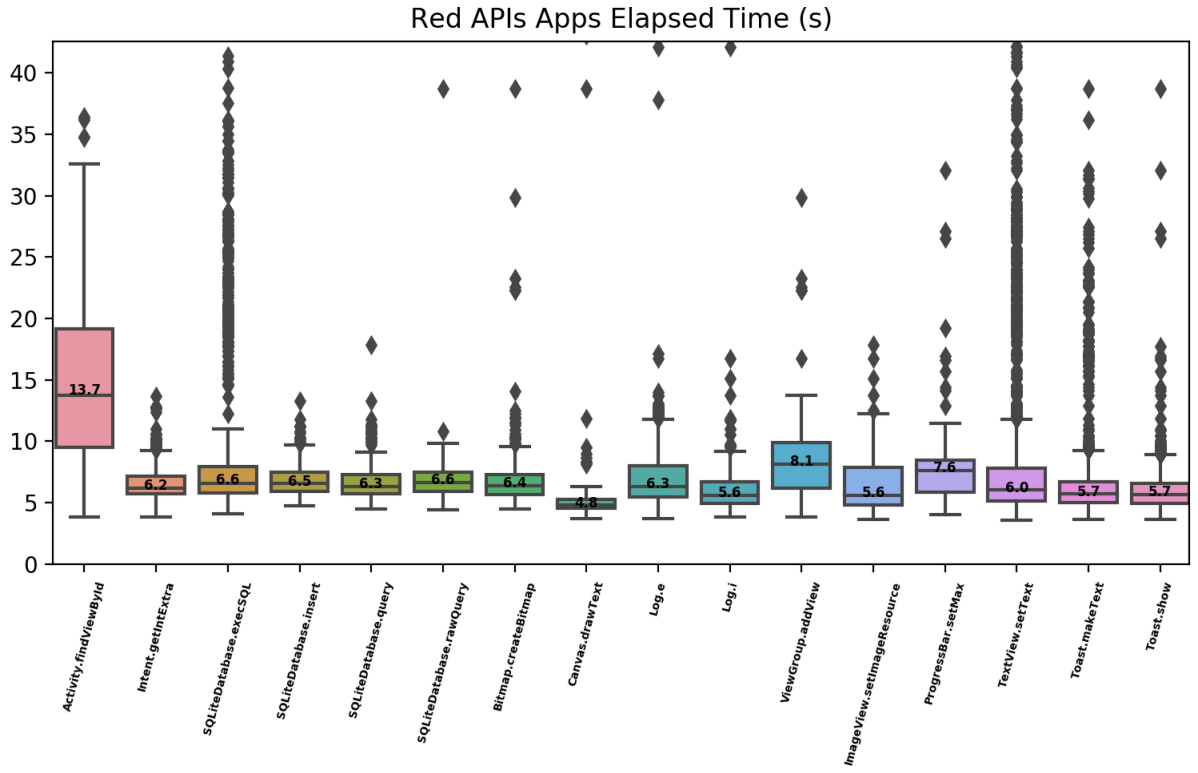


Figure 32: Elapsed time of tests with Red APIs

mainly focus on memory/run-time performance. In this section, we analyze and compare Lint performance issues in terms of run-time, memory and energy consumption. As of this date, there are 36 performance issues reported by Lint. In the context of our application repository, we identify 18 such issues that are present in at least 25 versions of more than 10 different applications.

Figures 33,34,35 present boxplots for the three performance indicators for each one of the 18 evaluated issue. The presented values are in Joules, MegaBytes (MB), and seconds, respectively. In the middle of each box plot we show the median value. In cases where no performance issues were detected in an app under test, it is shown as *None*.

In terms of energy consumption, as we can see in Figure 33, the *DisableBaselineAlignment*, *ObsoleteLayoutParam*, and *UselessLeaf* Lint issues stand out negatively, with higher whiskers, 1st quartile and median values. On the other side, the *VectorPath* and *Recycle* issues are more energy efficient, but do not have consistently lower consumption values when compared to the others.

In terms of memory usage, tested apps with no Lint issues (*None*) have the most dispersed values. *OverDraw* and *UseSparseArray* also have more dispersed memory usage values than the others, but consistently are higher than *None*. *MergeRootFrame* is the Lint issue least associated with high memory usage. On the other hand, *WakeLock* and *InneficientWeight* are the most memory-greedy.

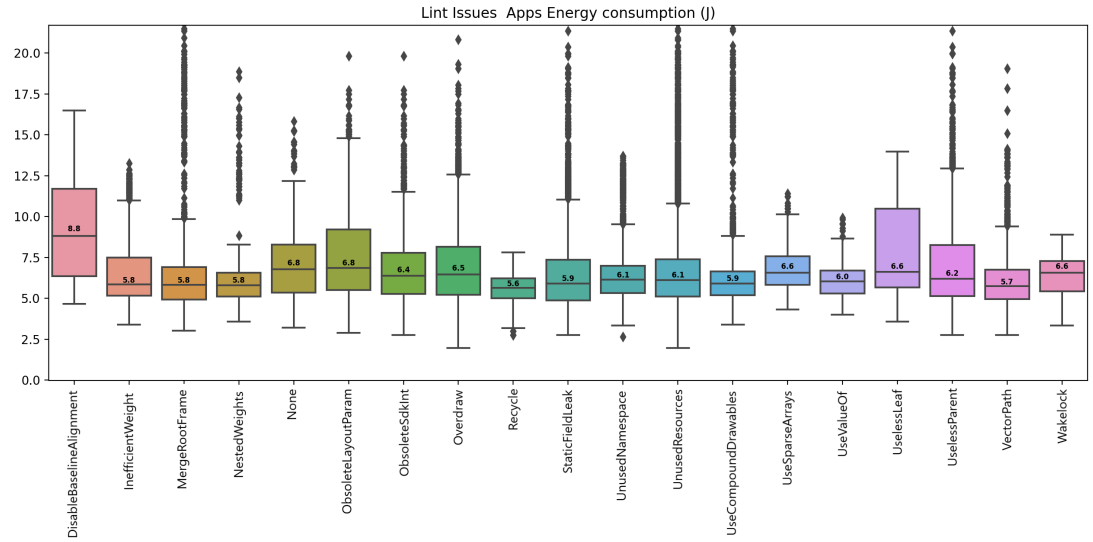


Figure 33: Lint Issues energy consumption

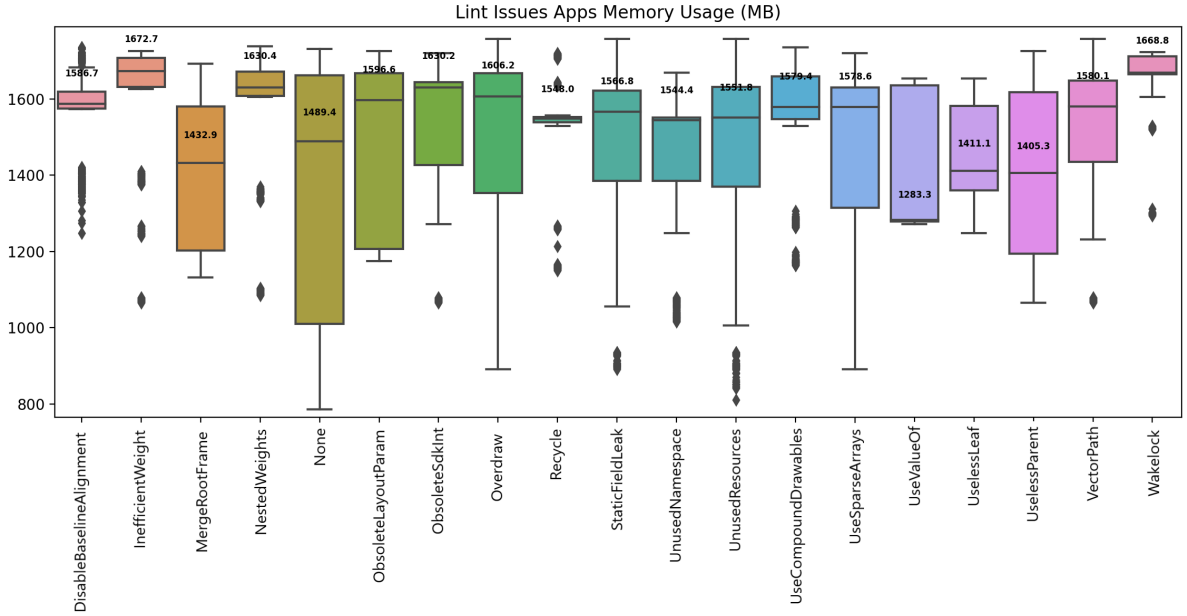


Figure 34: Lint Issues memory consumption

Finally, when it comes to execution time, the *DisableBaselineAlignment* and *UselessLeaf* Lint issues stand out in the negative, just as they did in terms of energy. Both display higher whiskers, 1st quartile, and median values over all others. *UseValueOf* and *Recycle* have a consistently lower run-time than the others. Additionally, most of the issues also have a lower run-time than *None*. This means that in our study, these issues did not have a noticeable impact on app run-time performance.

To assess whether there are statistically significant differences in behavior, in terms of performance across the 3 analyzed components, we evaluated the following hypothesis for each performance indicator:

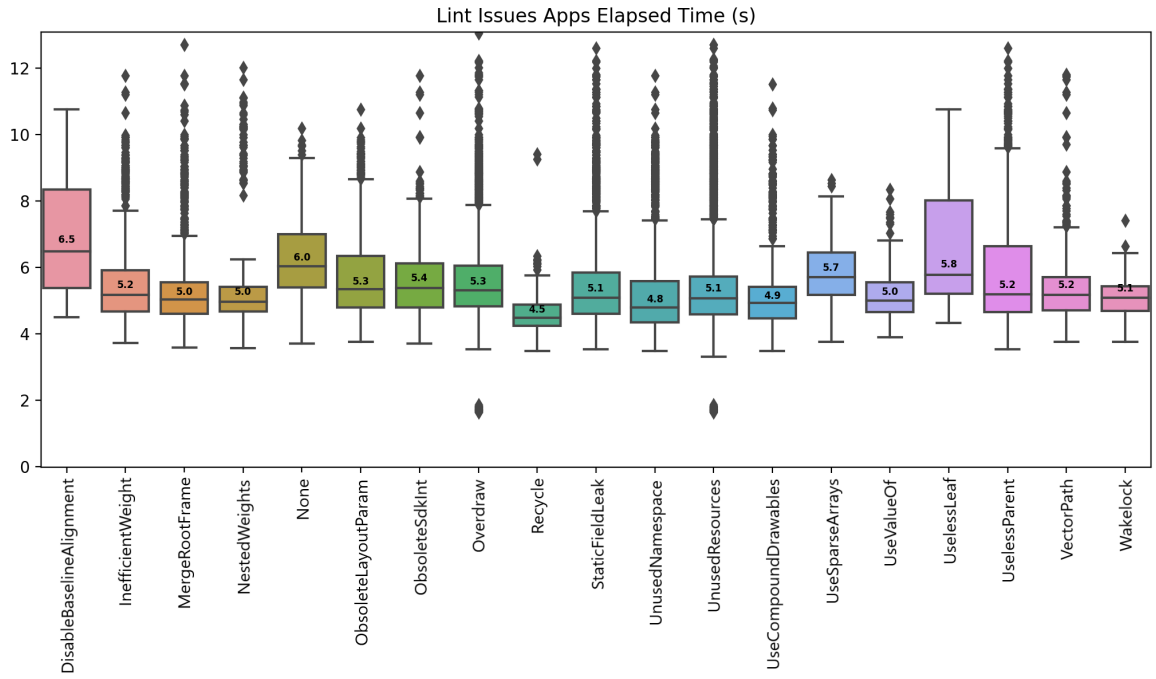


Figure 35: Lint Issues run-time

Table 21: Correlation results between performance indicators and Lint issues priority

Component	**Spearman ρ**	**Parker class.**
Energy	0.034388	Negligible
Memory	0.104777	Weak
Time	-0.044851	Negligible

$\boldsymbol{H}_0$: The values obtained for apps with Lint issues follow the same distribution; $\boldsymbol{H}_1$: The values obtained for apps with Lint issues do not follow the same distribution.

Having first assessed the data distribution using 2 different methods (Shapiro Shapiro & Wilk, 1965 and D'Agostino D'Agostino, 1971), we applied the Kruskal-Wallis (Kruskal & Wallis, 1952) method. This allowed the comparison of three or more groups of independent samples. Using Kruskall-Wallis, our H_0 was rejected, concluding that there are in fact statistically significant differences, across all performance indicators, between the issues.

Since the Lint issues are accompanied by different levels of *Severity* and *Priority*, we conceived the possibility of measuring the correlation between these levels and each of the performance indicators under analysis. As all the issues we evaluated in this section are of *Severity: Warning*, we assessed the correlation between *Priority* levels (scale from 1 to 10) and each of the components. For this purpose, the Spearman's Rank Correlation Coefficient (Spearman, 1904) of each pair was calculated (*Component*,*Priority*). Using the obtained ρ, we applied the Parker (Rea et al., 2016) rule to obtain a classification of the level of correlation between the pairs. The results of this analysis are shown in Table 21, showing that the performance indicators have little correlation with the *Priority* of these issues.

4.3.2.5 Alternative Libraries

One key aspect of the success of a programming language is the set of powerful libraries it offers: they do not only improve developers' productivity but also influence software's performance. The Android ecosystem is no exception, and the Android offers under its SDK a large set of libraries for networking, data structures, web data transfer, parallel processing, etc. Moreover, there are several third-party libraries offering alternatives and improvements to the already provided by the Android SDK.

Given the diversity of the analyzed apps in this study, we present a comparative study of alternative libraries for common Android programming tasks. In order to answer **RQ4.3.3** *Are there significant*

Table 22: Evaluated Libraries

Library	#Apps	Versions	Tests
HTTP			
Volley	6	25	79
OkHttp	16	87	447
HttpURLConnection	7	57	232
Retrofit	16	85	399
Picasso	9	34	164
Collections			
Sparse*Array	9	38	20
*List	159	865	244
*Set	129	692	147
*Map	71	325	127
Apache Collections	1	1	1
JSON			
Logan	0	0	0
Moshi	1	5	22
GSON	22	133	133
Jackson	1	3	3
org.JSON	22	96	405

Library	#Apps	#Versions	#Tests
I/O			
java.IO	89	400	1771
java.NIO	6	34	96
Okio	0	0	0
Apache IO	1	3	3
Logging			
util.Log	90	489	1807
Slf4j	1	3	3
Timber	6	21	158
Threading			
GreenRobot	6	14	86
RxJava	3	13	102
Executor	5	13	152
Kotin Coroutines	9	29	223
Image rendering/loading			
Volley	6	25	25
Glide	3	23	16
Picasso	9	34	164

performance differences between apps that use different competing libraries implementing typical programming tasks?, we selected a set of 7 programming tasks typically performed by Android apps. These libraries are listed in Table 22, organized in networking over HTTP, Collections, Image rendering/loading, JSON manipulation, I/O, Logging and Threading.

Taking into account the extent of the analysis carried out, we needed to select a limited set of tasks to be considered. Furthermore, we considered tasks with at least 2 alternatives/libraries, which have methods to be invoked in at least 5 different apps, across 10 different versions, and with 25 tests different test scenarios. The libraries under analysis are some of the most used libraries for Android for each category, according to AppBrain (AppBrain, 2020b).

In the following sections, we present results of evaluating the performance of HTTP, JSON and Threading libraries. We selected these three tasks because they were the tasks with the largest number of alternative libraries to meet our selection criteria when excluding Image rendering/loading and Collections. We did not present a deep analysis of the Image rendering/loading task since its libraries are also contained in the set of the HTTP libraries and we also didn't consider Collections since these were already analyzed

in terms of energy efficiency on the Android platform (Oliveira et al., 2019). The results obtained for the remaining libraries present are available in the Appendix A.

HTTP Libraries

HTTP requests are the main technique for transferring data in web apps, a key aspect of a software ecosystem targeting mobile devices. In this section we analyse the most widely-used HTTP libraries to perform communication via HTTP: *HTTPURLConnection*, *Volley*, *Okhttp*, *Retrofit*, and *Picasso*. *HTTPURL-Connection* was the first of these libraries to appear in Android development. *Volley* was released in 2013 as an alternative to *Okhttp*, offering more options such as multiple connections, request scheduling and cancellation APIs, working over *HTTPURLConnection*. *Retrofit* is a type-safe HTTP Client, more oriented to REpresentational State Transfer (REST) API access offering conversion to many common formats, while *Picasso* is optimized for downloading and caching media formats. Figures 36,37 shows a comparison of the performance values for 5 different libraries:

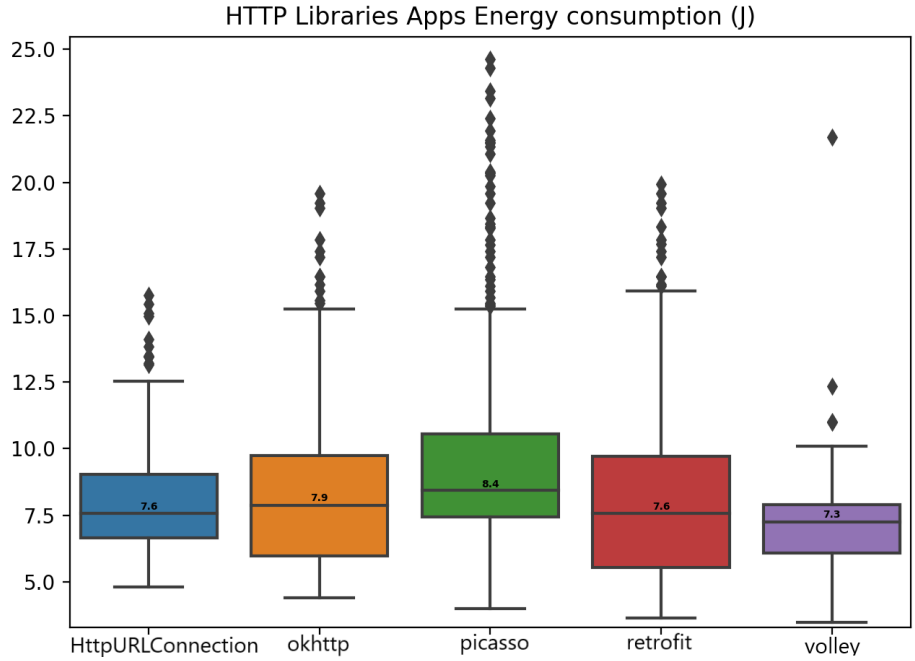


Figure 36: HTTP Libraries energy consumption

As shown in Figure 36, *Volley* is the most energy-efficient library, while *Picasso* is the most inefficient library in our study. The median values show that *Okhttp* consumes 43% more energy than *Volley*. *HTTPURLConnection* also has significant energy gains over the 3 most ineffective ones, resulting in an 18% energy reduction when compared to *Okhttp*. By looking at the medians of the boxplots, *Volley* is the most energy-efficient alternative with 7.5 Joules, with Picasso being the most inefficient with a consumption 12% higher.

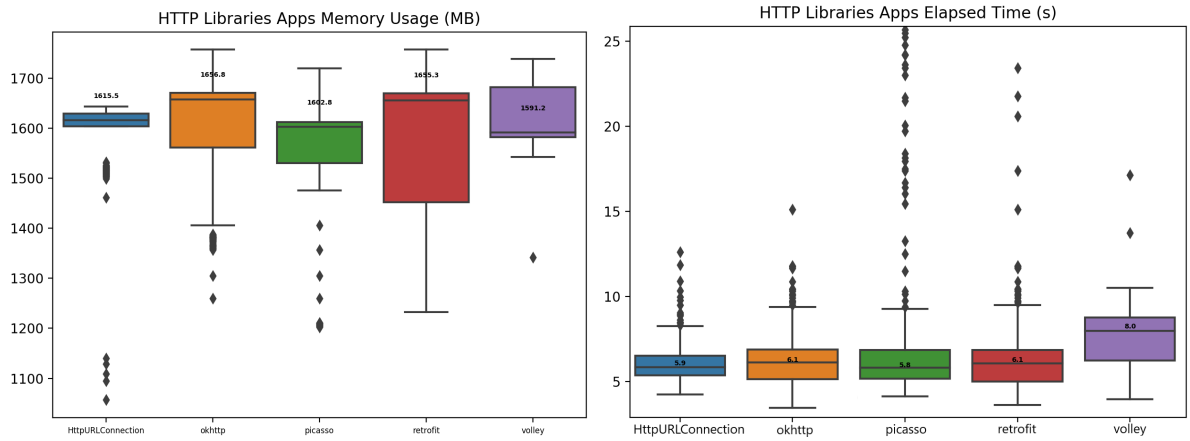


Figure 37: HTTP Libraries memory consumption (left) and run-time (right)

In terms of memory usage, *Okhttp* again presents the worst median, 1st quartile and top whisker, followed by *Retrofit*. *Volley* obtained the best median values, indicating it is the most efficient choice. Finally, in terms of run-time performance, *Okhttp* yet again obtained the worst median value, showing that it does not perform well across any of the 3 performance indicators. However, *Volley* remains at second place in the highest performance overhead but is superior in the top whisker and 1st quartile. *Picasso* is the most consistent, with the least variability and the lowest median, top whisker and upper quartile.

Testing the hypothesis that the distributions of the test results of each API are significantly and statistically different, we used the Kruskal-Wallis method with an α=0.05, after verifying the data as non-parametric. The conclusions support that there are significant statistical differences in the performance of the evaluated HTTP libraries.

Threading Libraries

Android developers are strongly encouraged to use threading/background processing to perform operations of storage access, networking, bitmap decoding, among others, to avoid main thread overloading. The main thread is responsible for updating the app's GUI and using it for processing heavy operations has an immediate impact on responsiveness and UX. In order to implement such a mechanism, several alternative libraries are included in the Android SDK. The three libraries we consider are *GreenRobot*, a third-party library that offers communication between components and threads via a publish-subscribe even bus, *RxJava*, a library aiming to abstract tasks such as low-level threading, synchronization, and thread-safety, through the use of the Observer pattern, and, *Executor*, which is typically used to manage a startup thread pool.

The recent emergence of *Kotlin*, however, is also influencing how developers use threading mechanisms. In fact, *Kotlin* has an asynchronous coroutine mechanism, which is lighter than traditional Java mechanisms for performing background and parallel processing. This mechanism does not require scheduling, and it makes use of *suspension*, which does not block the thread where the coroutine is running. The goal is to reduce memory usage and run-time by conveniently using such a mechanism.

In Figures 38,39 we show the results of three threading libraries and the (re)use of Kotlin coroutines.

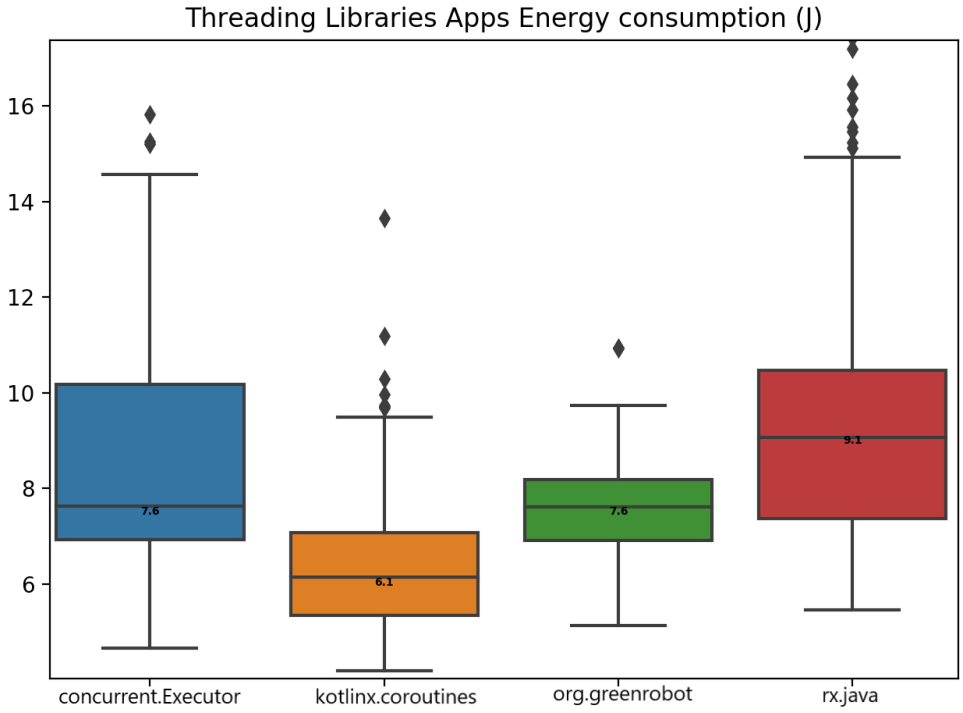


Figure 38: Threading Libraries energy consumption

As shown in Figure 38 there is a clear decrease in energy consumption when using the *Kotlin coroutines* mechanism. *GreenRobot* also shows gains when compared to both *Executor* and *RxJava*. The most inefficient energy consumption behavior comes from *RxJava*. *Executor* has a substantial gain over *RxJava*, obtaining similar results to *GreenRobot*.

With regards to memory usage, the scenarios invoking *Kotlin coroutines* do not show significant gains when compared to the alternatives. In fact, it ranks as the second-worst position, only behind *RxJava*. *GreenRobot* and *Executor* have similar median values, however, the former is consistently more memory-efficient than the latter.

When we consider run-time, the results are similar to those from the energy consumption analysis. *Kotlin coroutines* presents a significant gain in relation to the other libraries, presenting a run-time efficiency gain of approximately 32% when compared to *RxJava*, which is the worst evaluated alternative.

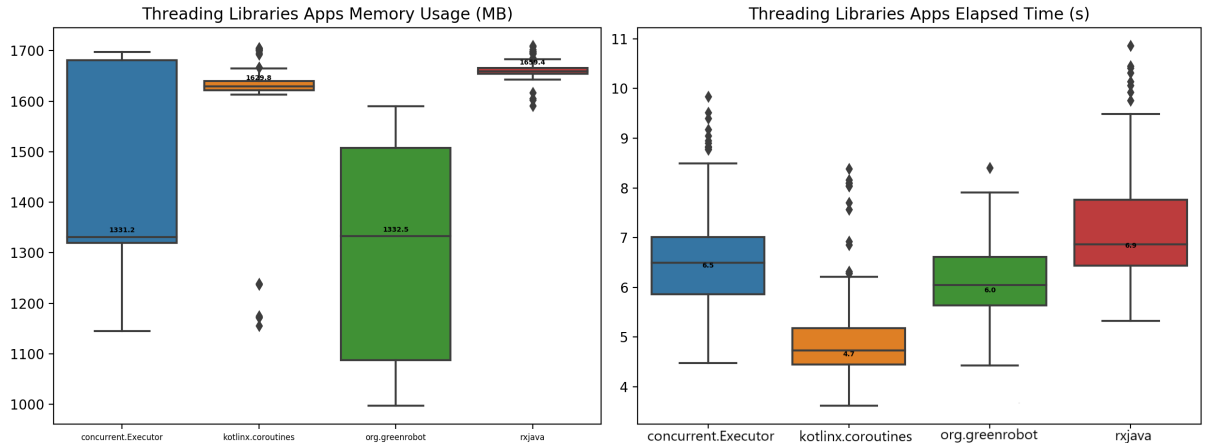


Figure 39: Threading Libraries memory and run-time

Just below *Kotlin coroutines, GreenRobot* is once again distinguished from *Executor*, as the second most efficient library in terms of execution time.

We once again tested the hypothesis that the distributions of the test results of each API are significantly and statistically different, using the Kruskal-Wallis method with an α=0.05, after verifying the data as non-parametric. The conclusions support that there are significant statistical differences in the performance of the evaluated threading libraries.

JSON Libraries

Android provides several libraries to parse, manipulate and serialize JSON data: a widely used lightweight data interchange format. In our study, we considered two libraries: the Java standard library *org.JSON* and *GSON* a JSON library developed by Google as a reliable, fast, and efficient extension to *org.JSON*.

The results of our study are presented in Figures 40 and 41.

Figure 40 shows that *GSON* has a marginal gain of 2.6% in terms of energy efficiency (considering the medians for comparison) when compared to *org.JSON*. *GSON* is also the fastest library: it is 12.6% faster than *org.JSON* as shown in Figure 41. In terms of memory consumption, however, *GSON* shows a slightly higher memory usage (0.94% considering the medians).

Testing our ongoing hypothesis that there are statistically proven differences between these 2 libraries, we used the Man-Whitney (Mann & Whitney, 1947) method (α=0.05) after confirming the data as non-parametric. The data concluded that while there were differences for run-time and energy consumption (values of p=0 and p=0.007, respectively), the same did not stand for memory usage (p = 0.112).

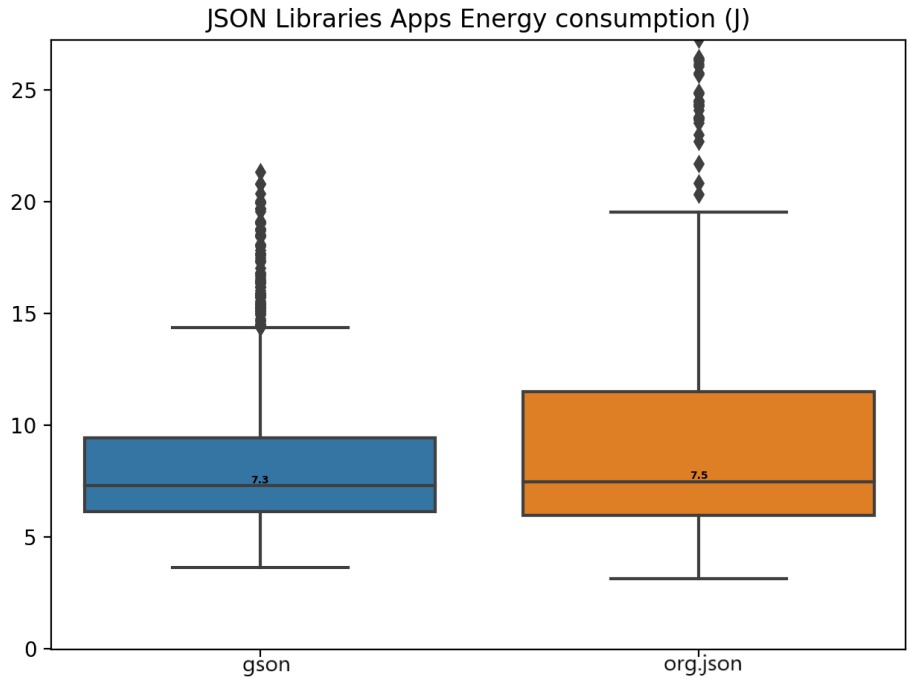


Figure 40: JSON Libraries energy consumption

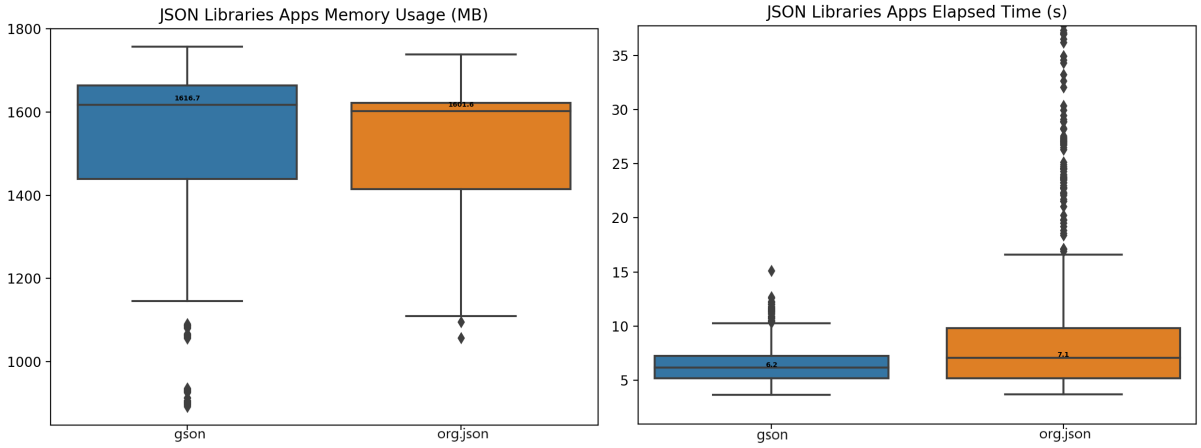


Figure 41: JSON Libraries memory and run-time

4.3.3 Contributions

In this section, we discuss the results whose analysis allowed us to answer the research questions presented in Section 4.1. We provide answers to these questions and also elaborate on the reasons for some of the observed results. Moreover, we define a set of programming guidelines based on the results extracted from the process of conducting this study. Finally, as a result of our study we compiled a resume of some of the information contained in our dataset, which contains all the data produced when conducting the study. This large dataset is also presented as a contribution and it is openly available to the scientific community for further analysis.

4.3.4 Research Answers and Related Findings

4.3.4.1 Research Question 4.4.1

In Section 4.3.2.1 a set of changes of various types that could occur in the behavior and in the source code of the applications were presented. These changes were evaluated in a set of 44 application version changes with relevant energetic behavior change (greater or greater than 20%). After analyzing the 44 application version changes, we synthesized the conclusions that allow us to answer research question (**ARQ4.3.1**). We concluded that changes more associated with relevant impact on the app's energy performance are changes that add *functionality* and *bugfixes* in applications, followed by adding *layout* and/or *strings* and *refactors*. Furthermore, we can conclude that semantic changes are the type of changes (according to the criteria presented in Table 14) more associated with a significant impact on energy consumption, while changes such as *Project Building* reveal an unclear impact. A negative impact on performance was expected for changes such as the addition of *functionalities*, as it typically results in an increase in computation and resources to the application which translates into an expected loss of performance. Also, since the testing procedure with the Monkey framework only interacted with the application for a few seconds, the tests mostly capture the app startup and initial setup workflow. Thus, since these changes are translated into more class variables to be loaded, they have a negative impact on app performance. Since a greater use of resources and code to be loaded by the virtual machine incurs additional effort, even if the code of the new functionality was not directly executed, additional functionality can impact negatively the apps' performance. An evaluation of which types of bugfixes or addition of functionalities have more (evident) impact on performance was not evaluated. This was due to the fact that we concluded that the amount of data that we collected to categorize different types of functionalities or bugfixes was not sufficient the derive general conclusions regarding its impact.

[ARQ4.3.1]: Which program changes have more impact on an Android's applications energy consumption? The changes more associated with relevant impact on the app's energy performance are changes that add *functionality* and *bugfixes* in applications, followed by adding *layout* and/or *strings* and *refactors*.

[F1]: Semantic Changes are the type of changes with the most significant impact on energy consumption. On the other hand, changes at **Project Building** level have an unclear impact on the energy consumption of the respective applications.

The impact of *bugfixes* can also be explained by the fact that its prevention avoids unnecessary computations to report the exception (e.g. logging the exception) and any system interruptions to stop and kill the application process, among others. The performance loss could be expected when it comes to significant changes in layout and strings since the emergence of new entities of this type results in static fields generated in the Android R.class Java class. This is an auto-generated class by AAPT that contains resource IDs for all the resources of the *res/* directory. When a component is declared in the XML file, aid for the corresponding component is automatically created in this class. However, this generation occurs during the compilation process and to translate into a significant loss of performance, it would have to be due to the creation of a large number of components of this type. In the case of adding new layout components (often associated with increased *functionality*), their creation and use translate into an expected loss of performance, since it is necessary at run-time to inflate an XML file in a View class, which is a heavy operation and has to be performed on single-thread [10]. The impact of including new views on energy consumption was also pointed out in Linares-Vásquez et al. (Linares-Vásquez et al., 2014).

For changes at Project Building level (according to Table 14), the impact is unclear and lacks a deeper analysis with a more extensive dataset. For example, the observed behavior for the impact of updating dependencies on the project is inconsistent, since it was observed that changes at this level have a significant positive impact on energy in 30% of cases and negative in 33% of cases. The addition of new dependencies reveals a similar behavior, with updates at the Android SDK level being associated with a negligible energy impact in 55% of the time (**F1**).

4.3.4.2 Research Question 4.4.2

In Section 4.3.2.2, we present results for two distinct types of widely-studied practices in the literature labeled inefficient ones: Linares-Vásquez et al. Red APIs and Android Lint Issues. These practices were evaluated in our execution context, with different applications and interactions. Based on these results, both the analysis of the Red APIs and the Android Lint Issues, we were able to provide our answer to RQ4.3.2 (**ARQ4.3.2**).

[10]Layout Inflater: https://developer.android.com/reference/android/view/LayoutInflater.html

[ARQ4.3.2]: Do previously identified inefficient programming practices exhibit the same behavior with larger datasets and different contexts? No, previously identified inefficient programming practices do not always exhibit the same behavior with larger datasets and different contexts. For instance, many of the Red APIs did not show abnormal performance behavior in our execution context.

In our executed apps and environment, some of the mentioned programming practices did not show abnormal performance behavior. In the case of Red APIs, some of the methods classified as energetically inefficient showed little difference in terms of energy consumption. The Man-Whitney test was used to assess whether the performance of apps invoking these APIs was different from the same apps that did not invoke such APIs. This test revealed statistically significant differences between these pairs but does not indicate whether such differences translate into a gain or loss in performance. Looking at the boxplots and respective Red classifications in Table 20, it is clear that these practices have obvious impacts, of different magnitude, on performance, and that several APIs (40%) are more often associated with energy efficient executions in these apps, such as *setMax* or *Log.i*. In the case of the evaluated Lint issues, while their presence does impact performance, it is not always a clear gain or loss in performance. Additionally, the *Priority* levels have very little correlation with actual performance measurement values (**F2**). For example, in terms of energy consumption and run-time, the vast majority of the tested apps, with identified Android Lint issues, have lower measurements than apps that do not have any of these issues. The performance indicator with the most apparent impact on these practices is memory usage, which displayed much higher median values when compared with tested apps with no identified issues.

Our results also allowed us to conclude that programming practices previously labeled as inefficient can have a positive impact on other performance indicators (**F3**). We compared several red APIs and Lint issues across 3 performance indicators and we observed that there are many cases of these practices that seem to be very inefficient in terms of one of the performance indicators while being among the most efficient on other performance indicators. Examples of such cases are the Inefficient Weight Lint Issue or the Log.i API. Inefficient Weight is the second most memory-greedy performance Lint issue evaluated when compared to the remaining in terms of median values while being one of the less energy-greedy. Log.i was the red API with more energy usage (on median) on our tests, but when compared to other red APIs, its energy consumption in median terms is one of the least energy-greedy ones. This evidences the fact that many programming strategies imply trade-offs in terms of performance indicators and the need to evaluate the impact of previously identified performance smells or APIS in several performance

indicators. Such evaluation can help practitioners select the practice more suitable to their requirements and execution contexts.

[F2]: Performance indicators have little correlation with the *Priority* of Performance Lint issues. Besides these issues being labeled with Priority values, these values appear to not have a statistical correlation with the measured performance indicators.

[F3]: Programming practices previously labeled as inefficient can have a positive impact on other performance indicators. We compared several red APIs and Lint issues across 3 performance indicators and we observed that there are many cases of these practices that seem to be very inefficient in terms of one of the performance indicators while being among the most efficient on other performance indicators.

4.3.4.3 Research Question 4.4.3

Section 4.3.2.5 presents the performance comparison between apps that used competing, widely-used libraries that are used to implement three typical programming tasks in mobile applications: JSON, HTTP and Threading libraries. The analysis of the results of these libraries allows us to answer RQ4.3.3 (**ARQ3**).

[ARQ4.3.3]: Are there significant performance differences between apps that use different competing libraries implementing typical programming tasks? There are, in fact, significant and macroscopic differences in the performance of real-world Android apps that use alternative libraries for common Android programming tasks. This evidence suggests that these libraries have significantly different performance impacts and the choice of the library can impact significantly the performance of the implemented task.

In the case of libraries for HTTP, apps that use *Volley* tend to have more energy-efficient tests, suggesting that this API might be the most energy-efficient API, despite being the worst in terms of run-time when considering the median and top whisker of the box plots of the Figures 36,37. In our experiment, *Picasso* seems to be the most memory and run-time efficient, being the worst in terms of energy. These results are different from the ones from another research study (Lachgar et al., 2018) that analyzed the

run-time performance of *Volley* and *Retrofit*, having concluded that *Volley* is faster and offers more features than *Retrofit*. This divergence can be again justified by the fact that our setup consists of a different execution context, considering different real-world applications and many execution contexts, while the experimental setup used in (Lachgar et al., 2018) considered only one application connected to a custom server, aiming at analyzing and measuring only specific methods of these APIs. However, in Figure 37, we can see that despite *Volley* obtained a higher median, tests using *Retrofit* obtained less consistent results, where many tests had high-consuming tests above the upper percentile when comparing to the values of the tests that used the *Volley* library.

Other results from our experiment evidence trade-offs in terms of performance, where apps that favor one component might end up being harmed across other performance indicators. Apps that used threading libraries also showed distinguishable results, in which the *Kotlin coroutines* stood out for its efficiency in terms of run-time and energy and *RXJava* exhibited the worst behavior in all the performance indicators evaluated. Finally, regarding JSON libraries, apps using *GSON* seem to be the most energy and run-time efficient, while losing in terms of memory usage in relation to apps using *org.JSON*.

4.3.5 Programming Guidelines and other Findings

This section presents several guidelines that were defined based on the collected results and their analysis. Such guidelines aim to steer software developers and the scientific community in developing greener Android software. This section also presents several findings that were derived from the analysis of the data presented in Section 4.3.2.

The data analyzed in the Section 4.3.2, which allowed us to answer the last two research questions, aims to advise developers to **always test their apps and evaluate their app performance individually** (**G1**), especially on devices and environments similar to where they will be executed. For instance, As can be observed by the Figures 29 and 31, there are many outliers in the indicators measured in the tests that invoke these APIs. This suggests that the classification of these APIs as energy-greedy or energy-efficient hardly depends on the context (Li et al., 2020; Pinto & Castor, 2017). For instance, some of these APIs are Logging APIs, whose energy performance was analyzed in prior publications (Chowdhury et al., 2018; Zeng et al., 2019). For instance, Chowdhury et al. (Chowdhury et al., 2018) demonstrated that factors such as logging rate, disk flush and message size have a significant impact on energy consumption. Nevertheless, the Android ecosystem, its APIs and its development environment are constantly

changing and most of the contributions and performance analyses made by the community relate to specific execution contexts and scenarios that may be out of date and far from more realistic and updated ones. In the case of APIs or patterns identified as inefficient, they can present such behavior only in certain contexts (size and type of arguments, system load, signal quality), sometimes exhibiting behaviors with negligible impact on the app's performance in most of the cases.

[G1]: Developers should always test their apps and evaluate their app performance individually. As observed in our results, many programming practices such as APIs previously identified as inefficient do not always exhibit such behavior and can even benefit other performance indicators.

In addition to developers having to evaluate the resource consumption of their apps individually, they must also **identify the main bottlenecks of their apps and target devices** (**G2**). In the case of the evaluated libraries, they showed different behaviors in terms of performance, with no single one being considered the overall best for all the evaluated components. Thus, there are APIs that, for example, are recommended whenever energy consumption is a concern, while they should be avoided when memory usage is a concern. In this sense, based on the analysis performed to evaluate the performance footprint of the considered Android libraries, we defined Table 23 (**G3**). This table represents a guideline in order to select the most efficient/inefficient libraries for each evaluated programming task, based on the median values of the app tests of our large-scale analysis. Using this information, practitioners can have a better understanding of which libraries they should choose or avoid, considering each performance indicator. For example, for HTTP libraries, *Volley* seems the best choice if both energy consumption and memory usage is of concern, but was the worst performing in terms of run-time efficiency.

[G2]: Developers should identify the main bottlenecks of their apps and target devices. *The results of our study show that similar programming practices in terms of functionality have different performance footprints across the considered performance indicators.*

Table 23: Most efficient/inefficient libraries **[G3]**

	Energy		Memory		Time	
Task	**Best**	**Worst**	**Best**	**Worst**	**Best**	**Worst**
HTTP	Volley	Picasso	Volley	OKhttp	Picasso	Volley
JSON	GSON	org.JSON	org.JSON	GSON	GSON	org.JSON
Threading	Kotlin cor.	RxJava	GreenRobot	RxJava	Kotlin cor.	RxJava
Collections	List	Sparse	Set	Sparse	List	Sparse
Image	Volley	Picasso	Volley	Picasso	Picasso	Volley
I/O	java.io	java.nio	java.io	java.nio	java.io	java.nio
Logging	Timber	util.Log	util.Log	Timber	Timber	util.Log

Another recommendation that we can provide to the community based on our findings is the incentive to **migrate to Kotlin**(**G3**), even partially, since Kotlin is a JVM language fully-interoperable with Java. Kotlin has mechanisms such as coroutines that are a lightweight alternative in terms of energy consumption and run-time to perform background processing. In our analysis, apps that used Kotlin vs non-Kotlin were compared regarding each performance indicator using boxplots (available in the online appendix). By comparing the values of the 2 groups it is possible to observe the benefits of using Kotlin. In the case of memory consumption, apps with Kotlin showed a substantially lower consumption value, when comparing the medians (3.57% less), whiskers, and lower quartile.

[G4]: Developers should migrate to Kotlin in order to improve energy performance. *Kotlin is a JVM language fully-interoperable with Java. Kotlin has mechanisms such as coroutines that are a lightweight alternative in terms of energy consumption and run-time to perform background processing.*

The long process of extracting and executing the applications, as well as the process of analyzing the results, also allowed us to understand what can be improved in our execution process and what we can recommend to researchers who intend to replicate or conduct similar studies. The first recommendation is that they **use the AnaDroid execution pipeline** (**G4**), used by us to conduct this study. Besides the features referred to in Section 4.3.1, such as source code instrumentation, automatic building, and run-time error detection, among others, during the writing and publication of this study, this execution pipeline was being redesigned (Rua & Saraiva, 2023b) and continuously improved and extended to support new energy profilers and testing tools. Furthermore, it is completely open-source and extensible, also allowing the easy replication and validation of later studies to be elaborated with this tool. It can be used to benchmark Android applications' performance, from its source code or even from an already-built APK.

Furthermore, it supports a new energy profiler (Rua & Saraiva, 2022), which can be used on most of the latest Android devices to analyze the energy consumption of your software.

[G5]: Practitioners should use the AnaDroid execution pipeline to conduct benchmarks the performance Android applications. *This tool is an extensive toolset designed to perform empirical analysis of mobile software performance.*

Our application execution analysis process also led us to conclude that it **is important to make measurements in isolated environments, as in customized Android images** (**G5**) without vendor, manufacturer or 3rd party-apps and services that can interfere with the monitoring process. Besides our efforts to reduce external interference in the monitoring process, we still observed several outliers and filtered several executions where we detected errors caused by other system services and apps.

[G6]: Practitioners should make measurements in isolated environments, such as in customized Android images. Despite our efforts, we still observed many outliers that we associated with events related with other system services and apps.

The execution of this software during the monitoring process introduces overhead to the system and also can raise errors which handling can also directly impact the app under test functioning and also in the performance measurements. For instance, the execution of such services can trigger a CPU frequency scaling event, which can impact energy consumption, as CPU frequency is highly correlated with energy consumption (Chowdhury et al., 2019).

The results presented in Section 4.3.2 aiming to present a taxonomy of the considered applications in our study also allowed us to observe other interesting findings that characterize the current Android software development paradigm. One of the observations that the analysis performed allowed us to provide it is that **App's age does not hint expected performance** (**F4**). We analyzed the data in Figure 26, which shows the relation between the energy consumption of the apps and their release year. Our analysis suggests that besides a noticeable rise since 2017, there is no clear tendency for to apps increase energy consumption over time, even when the energy is being measured in the same device. We also observed the same in terms of memory and run-time.

[F4]: App's age does not hint expected performance. Our analysis suggests that there is no clear correlation between the app's age and degradation of performance.

4.3.6 Dataset

This section presents a summary of the data contained in the results dataset. In addition to the analysis of the previous sections and the automatic execution procedure, this work also presents the gathered dataset as a contribution. This dataset is openly available to the community in the online appendix and the Greensource (Rua et al., 2019) infrastructure. Since the execution procedure is open source and has been designed to be independent of the device and testing framework, the dataset can also be expanded to consider new execution scenarios, such as executions with different interaction and testing frameworks.

The data contained in this dataset presents results from more than one month of continuous test execution. Given the extent of the work presented, from the process of extracting and filtering applications to the agglomeration and obtaining of its results, it is pertinent to share it with the community so it can be used in further studies aimed at analyzing the performance of applications or devices, comparisons of Android testing frameworks, evaluation of software evaluation aspects, between others. In this way, it is intended to contribute to the increase of knowledge regarding the factors that influence the performance of Android applications and their respective weight. Some examples of other research questions that are examples of further research directions (**RD**) that might be answered by analyzing the data contained in this dataset or possible expansions of it are:

- **RD1:** *How do different testing frameworks explore application performance?*
- **RD2:** *What is the most suitable testing framework to detect functional errors in applications?*
- **RD3:** *How does applications' performance evolve throughout their versions?*
- **RD4:** *What is the relation between semantic version changes and performance?*
- **RD5:** *Which types of functionality and bugfixes have more impact on Androids applications energy consumption?*

Given the extent of this work, we believe that the answer to these questions is worthy of work dedicated to exploring these questions. **RQ4.3.1** and **RQ4.3.2** could be answered by considering the data regarding the execution of the 2 testing frameworks used in our study or by adding additional testing frameworks or test cases to the dataset. The third could be answered with the already gathered data, by sorting the applications according to the semantic version or chronologically and exploring conclusions from the data. Finally, the fourth question might be answered by relating the different types of semantic version changes with the performance indicators obtained for the tests executed over the applications.

These data contain a large set of information regarding open-source applications and respective code, as well as static and dynamic metrics regarding their execution and execution environment. These metrics were collected at different levels of granularity, and their value/result can be attributed to an application, a test performed on an application, or a block of code belonging to the application. In table 24 are the metrics collected during the entire application execution process. Each metric collected is presented in the table accompanied by a description and the level at which it was obtained:

- **Application Level**: Metric whose value is inherent and dependent on the executed application. Example: Number of files for an application
- **Test Level**: Metric whose value is inherent and dependent on the test performed on an application executed in a given context. In addition to the application and the type of test/interaction performed, it also depends on factors related to the device and the respective system. Example: memory consumed during the execution of a test.
- **Method/Class Level**: Metric whose value is associated with a block of Kotlin or Java code. Example: Number of methods.

Table 24: Metrics gathered during tests/apps execution

Metric	Description	Level	Metric	Description	Level
Android APIs	APIs used from the Android SDK	M/T/A	Total Energy	Total energy consumed	T/A
Java APIs	APIs used from the Java SDK	M/T/A	Battery Charging	If Battery was charging	T/A
External APIs	Other APIs	M/T/A	Memory	main memory consumed	T/A
Wi-Fi State	If Wi-Fi was used	T/A	# running processes	Nr. of other processes running simultaneously	T/A
Screen State	If Mobile data was used	T/A	LoC	Lines of code	A/C/M
Battery Status	Percentage of battery	T/A	Elapsed Time	Elapsed Time	T/A
Battery voltage	Battery Voltage	T/A	CC	Cyclomatic complexity	A/M
Battery Temperature	Battery temperature in degrees	T/A	#Args	Nr of arguments	M
Battery Charging	If Battery was charging	T/A	#Declared Vars	Nr. of declared variables	M
Wifi Received Signal Strength Indicator (RSSI) Level	Level of RSSI	T/A	#Methods	Number of Methods	A/C
Bluetooth State	If Bluetooth was used	T/A	#Class Vars	Number of Class Vars	C
GPU Frequency	GPU frequency	T/A	#Classes	Number of classes	A
CPU Load Frequency	CPU load frequency (per core)	T/A	#Files	Number of files	A
GPS State	If GPS was used	T/A	Languages	Languages present in App sources	A

Static and dynamic metrics about the device were also collected before and after the execution of the tests, to verify that the execution of the framework and the application did not change the state of the device, which could make comparisons with other applications impossible or bias the results of the next tests. The online appendix contains Tables presenting examples of such metrics, such as the Used CPU, battery level, Proportional Set Size (PSS) Memory Usage, among others.

The collection of all these metrics has led to it being possible to collect more than $1,200,000$ measurements of dynamic metrics related to executions on real devices and more than $420,000$ measurements of static metrics related to applications and their source code.

Table 25: Resume of the different metrics obtained

	Total	Description	Example
Tests	26289	Tests performed over apps	monkey test
Apps	214	Different apps executed	com.niesens.morsetrainer
Versions	1280	Different versions executed	com.niesens.morsetrainer version 1.1
Dynamic Metrics	47	dynamic metric obtained for tests performed	average CPU frequency of during a test
App metrics	14	Static metrics globally related to apps	number of files, languages
Method Metrics	7	Static metrics related to app methods	number of arguments
Class Metrics	5	Static metrics related to app classes	number of methods

All these metrics result from the analysis of the files obtained from the execution of tests on applications with instrumented source code. These were derived from the analysis of the Trepn Profiler log files collected during the monitoring process and from the method traces collected during the application's execution. In addition to these files also being available in the dataset, the system logs (through the *logcat* tool) generated during the monitoring process were also collected. These logs contain dumps of system and app messages, including stack traces of exceptions that occurred, which may also be useful for further studies aimed at evaluating the relationship between system events and application or device performance.

In addition to all this information that was collected for all running applications, data relating to the manual analysis of versions with anomalous energy performance (as described in section 4.3.2.1) is also available in the online appendix. The analysis presents tables with the changes registered between each version change, which can be reused by the community to deepen the impact of the evaluated changes, either on performance or on other functional and non-functional aspects of the application's operation.

4.3.7 Threats to Validity

Profiling the impact of specific coding practices such as patterns, APIs, and refactorings, has been already measured in previous works. In order to quantify the impact of such practices, it is a common procedure to isolate the aspects under analysis to avoid invalid conclusions. For instance, in (Sahin et al., 2014) coding practices were implemented in Android apps, and the energy consumption was profiled before and after individual changes. However, this article aims to identify the impact of these practices in real use cases (real applications) and on the overall app performance and does not aim to focus on specific practices, but rather on some that have already been studied and were invoked by the execution of the tests. The events generated by the selected testing tools generate a lot of entropy in the test results, as they are not intended to focus on the specific points of the applications that contain the practices under

analysis, generating events with a redundant effect for this purpose. However, given the magnitude of the set of applications considered, using white-box testing techniques under this dataset size was impractical.

Using test frameworks to test and estimate app consumption is a typical procedure used by the scientific community (Chowdhury et al., 2019; Couto et al., 2015a; Hu et al., 2017; Li et al., 2020; Linares-Vásquez et al., 2014; Nucci et al., 2017a; Rua et al., 2019, 2020). When it comes to running a large number of tests on a set of variable size apps, using user input is impractical. In addition, collecting or replicating real user inputs often means providing private personal information.

The testing frameworks used, Monkey and App Crawler, can be used independently of the app they intend to test, using a black-box testing approach and performing tests on the app's UI. The first is typically used to evaluate performance(Hu et al., 2017; Li et al., 2020; Linares-Vásquez et al., 2014; Nucci et al., 2017a), and it can be similar to several system events (touches, swipes, clicks) on the test device and adjust the event rate. The second is used by Google to validate the apps submitted for evaluation in the Play Store, simulating events on the app's UI elements to explore the app's code execution. Both frameworks can be configured, and a specific set of instructions can be filtered or carried out to help frameworks increase the coverage of the tested UI elements and, consequently, the app code.

The set of collected apps has software pieces developed in a relatively diverse range, which varies between 2015 and 2020 (between 2010 and 2020 if we consider the first release year of each application). The fact that many of them presented APIs and recent languages like Kotlin, shows that they may be apps that represent the state of the art of Android development and can help to characterize the state in which the development is. The apps are of different categories and the versions collected are diversified and from different development stages. However, the data we collect regarding the downloads of these in the Play Store reveals that, except for some executions, the open-source apps collected are not very popular with Play Store users, according to the download count reported by them. Nevertheless, many of these apps are not downloaded from the Play Store but rather downloaded from 3rd party stores such as F-Droid or Aptoid, or their official page or repository, encouraging the movement of open-source software.

Our corpus of applications contained only open-source applications extracted from open-source repositories. Despite studying open-source apps being a widely-used procedure by the scientific community (Couto et al., 2015a; Das et al., 2020; Li et al., 2020; Ribeiro et al., 2021; Rua et al., 2019), considering closed-source apps could help to obtain more universal and representative conclusions from the Android platform. However, analyzing source code from closed-open-source apps is a technically more elaborate process, since it involves the tasks of decompile or unobfuscate code. The automatic

execution process used in this work already involved an application processing pipeline that caused a significant number of applications to be excluded from the study due to problems encountered during the building process, instrumentation or automatic execution. Adding even more complex steps to the process to consider (eventually more popular or even robust) closed-source apps would likely result in a reduced set of applications to analyze.

All applications of our dataset have Java and/or Kotlin code present in their source code. Some applications also have HyperText Markup Language (HTML) and/or JavaScript (JS) source code, and it was not possible to automatically infer whether these files were actual source code files used by the application or just auxiliary project scripts/files. Therefore, we chose not to reject possible hybrid applications or applications implemented using WebView. Considering this fact, in the author's opinion, the performance metrics obtained at the test level can still be attributed to the consumption of the application, although the metrics of coverage and instrumentation process might not cover all the executed application code, since they only cover Java/Kotlin source.

Vilkomir et al. (Vilkomir & Amstutz, 2014) evidenced that studies from the state-of-the-art have an evident limitation: these are validated in one specific combination of hardware/software that can limit the generality of the reported results. With this study, we do not intend to evaluate performance on different hardware combinations. We aim to evaluate and validate performance-related programming practices in a different execution context and evaluate if we can draw the same conclusions in our own distinct context. The executed apps were tested on 2 different Nexus 5 devices, running the same customized version of the platform. These devices have already been used in several energy-related studies (Cañete et al., 2020a; Couto et al., 2015a; Oliveira et al., 2017b; Rua et al., 2019) The selected platform version is the last official version supported by the device and was modified using root privileges in order to remove services and apps that ran in the background and were not necessary for the apps to run. Among these apps are essentially Google services and apps that periodically perform background work and use sensors such as Wi-Fi and GPS to update system apps and widgets. We considered 2 devices of the same model in order to establish fair comparisons between results obtained in both devices, since the performance measurements that Trepn Profiler estimates are obtained at the system level. Otherwise, it would not be possible to establish fair comparisons between results obtained on the 2 devices, since the magnitude of the performance measures and factors that influence them would not be similar, as the 2 devices would have different hardware and execution contexts. However, such a setup would bring other advantages, since considering more and different execution contexts increases the possibility of obtaining

more universal and representative conclusions from the Android platform.

Finding an adequate tool for energy profiling for the Android environment was also a challenge. Trepn is an accurate tool (Hoque et al., 2015), capable of profiling hardware and resources usage, as well as power consumption of the system or even standalone Android apps, being able to provide fine-grained sub-component specific power consumption. For obtaining data for measuring power, it depends on a special Fuel Gauge chip present in the PMIC which controls the distribution of power from the battery. For the usage statistics of different hardware components, Trepn depends on the */proc* pseudo-filesystem and other system files. The main limitations of this profiler are that only gets accurate battery power readings from Qualcomm System On Chip (SoC)'s, the sampling rate, which cannnot be adjusted to less than 100 milliseconds and its availability on new devices and platform versions. However, the selected Trepn version (6.2) reports accurate system power consumption for the Nexus 5 (Qualcomm, 2014). There is a set of configurations to enable/disable profiling data points to ensure that the profiler records only the information required for this study and reduces its impact on system power consumption and causes noise in the system and the monitoring process. Furthermore, for this study, we configured Trepn to only profile the data points under analysis in this document.

The results presented in this study do not address measurements of the impact of the instrumentation on the consumption of the tests performed over the applications. Obtaining a reliable estimate for its overhead is complex since it depends on many factors intrinsic to the source code of the measured app, the type of interactions performed on the application and also the OS itself. However, taking into account that the comparative results presented always involve a significant and diverse set of applications and executions, the authors consider that the impact of this overhead will eventually have a negligible impact on the conclusions obtained.

4.4 Summary

In this chapter, we have presented two distinct research efforts that empirically analyzed the impact of programming patterns and practices on the performance of Android applications. Both of these analyses considered executions in physical Android devices, with the measurements being obtained in controlled conditions.

In the first research effort, presented in Section 4.2, we delved into the application of memoization in Android applications, specifically examining its impact on energy consumption. We selected 18 methods

from three different applications and designed an experiment to assess the energy consumption of these methods, comparing their performance with their memoized versions. The results of this experiment indicate that the use of memoization indeed leads to energy savings in the majority of cases.

The initial experimental findings obtained in the first research effort serve as a foundation for future investigations into the broader implications of these energy savings on the overall consumption of the applications under consideration. While the primary focus of this preliminary evaluation was to highlight the impact of memoization on energy consumption, a promising avenue for future research involves exploring the relationship between these results and the execution time and energy consumption achieved through memoization refactoring.

In the second research effort, presented in Section 4.3, we presented an extensive static and dynamic performance analysis of Android applications. Our execution process involved the extraction of real apps and the respective code from open-source repositories, their compilation and execution on physical devices to confirm their full execution and the execution of a set of tests with 2 frameworks that totaled an execution time of over 28 days. The collected and analyzed data in this study allowed both a macro and microscopic analysis in terms of mobile apps' performance and their respective source code. The results analyzed in this article (fully available in the online appendix) validated results from several previous studies but also presented some divergent results. This observation serves to prove once again that it is difficult to obtain universal knowledge regarding the expected performance of code, APIs or software patterns/smells. The behavior of code and software depends directly on the context and the execution environment and can be influenced by many external factors.

In our execution context, we manually analyzed a set of 31 apps and 44 versions with anomalous energy variations to identify software changes with a significant impact on app performance. From our results, we concluded that from the 15 software changes considered in our analysis, only the addition of functionalities, bug fixes, refactors, adding layouts, and strings have a noticeable impact on app performance. By re-evaluating the applicability of conclusions gathered in previous performance studies, such as Linares-Vásquez et al.and the Android Lint Issues, we concluded that there are significant differences in the performance of practices already identified as harmful in this sense. Regarding the study of Linares-Vásquez et al., we observed that only 40% of the evaluated energy-greedy APIs had a noticeable impact on app performance, with the remaining having a negligible impact. Furthermore, regarding Lint Issues, we observed that many of these issues did not reflect their supposed negative impact on app performance. Additionally, we concluded that the severity level that accompanied issues, labeled as performance issues,

has very little correlation with the correspondent performance.

Finally, it has been shown that alternative libraries used to perform common programming tasks in Android apps had different impacts on the apps' energy consumption and that there might be significant gains in performance just by migrating to a more efficient library. This contribution was made in a macroscopic way, being able to distinguish versions of these libraries in future works, as well as the respective methods they provide. In this sense, in Section 4.3.3 we present a table containing the most efficient and inefficient libraries for each task and performance indicator evaluated in our study. Also in Section 4.3.3 we present other guidelines that result from the analysis of our data.

5

Greenspecting Android Virtual Keyboards

Abstract: In a mobile setting, energy consumption is a key concern for hardware manufacturers, developers and end users. Virtual keyboards are widely used applications in a mobile setup. Furthermore, inefficient applications with significant usage can have a significant negative impact on the overall energy consumption of the underlying device. In this chapter, we present a study on the energy performance of five of the most used virtual keyboards in the Android ecosystem. We measured and analyzed the energy consumption in different keyboard scenarios, namely with or without using word prediction. We provide results that consider two different experiments: one where we instructed the keyboards to simulate the writing of a predefined input text, and another where we performed an empirical study with real users writing the same text. Our results demonstrate that there are relevant performance differences among the most used keyboards of the considered ecosystem, and it is possible to save nearly 18% of energy by replacing the most used keyboard in Android with the most efficient one. We also showed that is possible to save both energy and time by disabling keyboard intrinsic features and results that suggest that the use of word suggestions does not always compensate for energy and runtime.



"Greenspecting Android Virtual Keyboards"
Rui Rua, Tiago Fraga, Marco Couto, João Saraiva
In: 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft 2020)
pages: ***98 –- 108*** *| DOI:* ***10.1145/3387905.3388600***

5.1 Introduction

With regard to mobile platforms, it has become evident that different software development techniques have different impacts on energy consumption. End users of these devices have been showing concerns regarding energy-related issues, rejecting applications they identify as energy-greedy, and looking for ways to increase uptime on their devices. Thus, software developers for these platforms have shown a growing interest in optimizing their software from an energy point of view.

Very much like a computer user relies on a physical keyboard to interact with a wired computer, a mobile device user mostly relies on a virtual keyboard to interact with a mobile device. A virtual keyboard is a software application offering advanced features that overcome regular physical keyboards, such as multilingual word prediction and auto-correction, key animations, among others. Although mobile operating systems offer a predefined keyboard, there are a large number of alternatives offering similar features that users can install on their devices. Being a software application, virtual keyboards may influence the overall energy consumption of the underlying mobile device. Surprisingly, there is no research work on performing a detailed analysis of the energy efficiency of the mainly used keyboards, nor on analyzing the energy impact of their advanced features.

In this chapter, we analyze the energy consumption of some of the most widely used virtual keyboards and their advanced features. We consider the five most used keyboards available and suitable for our empirical setup in the Android ecosystem, which is the widest-used mobile phone operating system worldwide. To analyze its energy consumption we conducted two studies: First, we used an Android testing framework to exercise the virtual keyboards, by automatically writing a predefined input text in the same mobile device, with exactly the same conditions, while monitoring the energy consumption. Afterward, we conducted a second study with 40 real users who also had to write the same/previous input text. Moreover, in both studies, we considered two usage scenarios. In the former, we considered one with all features off and another in default mode (the pre-install setup, where prediction and auto-correction are available). For the latter, we considered similar scenarios but in default mode: One without using the word suggestions and another maximizing the use of such feature.

These studies aim to answer the following research questions:

Research Question 5.1:

Are there any significant differences in energy consumption between the most used Android keyboards?

Research Question 5.2:

Can significant energy gains be achieved by minimizing the work and features that keyboards use?

Research Question 5.3:

Does the eventual runtime gain of using (computationally intensive) word prediction algorithms compensate for energy?

The results of our studies show that there is a statistically supported difference in the energy consumed by the most used keyboards in the Android platform. We also show that the energy gains can go up to 18% just by replacing the most energetically inefficient keyboard with the most efficient one. We also show that there are cases where minimizing the number of features that keyboards offer can help achieve a drop in energy consumption of up to 9.3%. The preliminary results of the second study show that the use of suggestions may not always be beneficial in saving time and energy.

The remaining chapter follows the following structure: Section 5.2 introduces the keyboards selected for this study and describes the procedure followed to automatically test and monitor the keyboards in real physical devices. It also presents the methodology followed to isolate the testing environment and the system image from undesired interference. Section 5.3 presents the results obtained from the studies performed automatically and with real users and Section 5.4 presents the main conclusions that the results allow to extract. In order to discuss some of the potential threats to this work, in 5.5 we discuss some of the decisions followed to elaborate this work. Finally, in Section 5.6 we draw the main conclusions that can be inferred from this work and the respective results.

5.2 Studying the Energy Efficiency of Virtual Keyboards

In this Section, we fully explain all the underlying aspects of the conducted studies to analyze the energy consumption of Android keyboards. First, we explain how the keyboards used for the experiment were selected, and we describe each one of them (Section 5.2.1). In Section 5.2.2, we then explain the full experimental setup: the devices used for the tests, their state, the supplementary tools used and their purpose, also addressing some of the precautions we took to minimize external interference in the procedures followed in both studies. We present in Section 5.2.3 the full details of the procedure followed for the experiment which includes 2 different studies: one performed using an automatic approach and another using real users.

Table 26: GBoard: Languages, number of files and (source code) lines

Language	#Files	#Lines	LoC
Java	664	117095	75181
C/C++	127	1923	14543
HTML/CSS	7	392	77
C/C++/Header	158	15788	9941
XML	1338	241308	164727
Others	13	746	362
Total:	**2307**	**377252**	**265104**

5.2.1 Keyboards Under Test

A virtual keyboard application is an essential software artifact for mobile device users. A keyboard is not a standalone application, but instead is the software used as the input method for other software applications, for example, to write text messages, edit texts, write an e-mail, fill forms, etc. In fact, a virtual keyboard application offers advanced features that greatly overcome classical physical keyboards, such as the use of multilingual auto-correction and word prediction, key pressing animations (both visual and vibration ones), etc. Thus, the keyboard app became a Swiss-army knife that greatly contributes to improving the user experience of a mobile device/ecosystem.

To provide those advanced features, virtual keyboards are complex software applications. For the language-specific auto-correction feature the software needs to efficiently index large dictionaries. Moreover, to give helpful word predictions, keyboards rely on learning algorithms and artificial intelligence techniques. We can look at nowadays keyboards as large and complex software systems. For example, *GBoard* keyboard (Google keyboard) application is written in several programming languages and in its version 4.1.2.3 it has about $265,104$ lines of code as shown in Table 26.

Being virtual keyboards an important aspect of any mobile device ecosystem, a predefined one is pre-installed by all device manufacturers. The keyboard that typically comes with the Android platform is the GBoard keyboard. However, some manufacturers tend to offer alternative keyboards (for example, Samsung pre-installs its own keyboard, while Huawei re-uses Microsoft *Swiftkey* keyboard [1]). Not only do these large companies provide such advanced virtual keyboards, there is a huge number of similar keyboard applications available from the official Android Play Store.

[1] https://www.microsoft.com/en-us/swiftkey

Table 27: KUT features list. E - Enabled by default, D - Disables by default.

	Cheetah	**Fancykey**	**Go**	**Google**	**Swiftkey**
Auto-Capitalization	E	E	E	E	E
Suggestions display	E	E	E*	E	E
Next word prediction	E	E	E	E	E
Auto-correction	E	E	E	E	E
Emoji prediction	✗	E	E	✗	D
Gestures (swipe input)	E	E	E	E	E
Multi-dictionary	E	E	E	E	E**
Search engine integration	E	E	E	✗	E
Keypress animations	E	E	E	E	E
Voice input	E	E	E	E	E
Keypress Vibration	E	E	D	E	E
Keypress Sound	E	E	E	D	D

Given the wide variety of keyboard applications, and the unfeasibility of analyzing the energy consumption of each on individually, we selected a representative subset of the keyboard apps available on the Play Store. First, we used the provided number of downloads to select the 10 most downloaded keyboards. Since the number of downloads does not necessarily mean the keyboard is currently being used, we combined this information with the data contained in the GreenHub (Matalonga et al., 2019) dataset: an ongoing green software crowd-sourcing initiative, that contains data collected from more than $60,000$ currently running devices, which included the app packages installed in each of them. Using the GrenHub dataset we select the five most used keyboards from the previous set of 10. Samsung keyboard was excluded from this set because it is only available for Samsung devices, which do not support our energy monitoring framework (see Section 5.2.2), and its applicability is limited compared to the other alternatives. Table 27 shows the main features of each KUT considered in our empirical studies (Cheetah, FancyKey, Go, Google and Swiftkey), as well as a comparison among these in terms of such features.

5.2.2 Experimental Setup

Having defined the five representative virtual keyboards we will consider in our study, we now present the full setup designed for our experiments: the devices used for the tests, the tools used to *(i)* precisely monitor the energy consumed by the keyboards and *(ii)* to define the same tasks being executed over different keyboards, and how we guarantee that we minimize, as much as possible, all possible external interference to the keyboard application usage.

Mobile Device and Tailored Android: This study was conducted on two rooted, fully-charged, LG Nexus 5 devices. This model is usually used by the research community to conduct performance studies (Cañete et al., 2020b; Rua et al., 2019; Rua et al., 2019, 2020) due to the fact that does not have many pre-installed 3rd party-apps. Google supported upgrades from the platform version on this model up to version 6.0.1, and it was this version that was chosen to perform the study. Version 6 of the platform is still the most widely used version worldwide ("Distribution dashboard", 2020), and it uses the ART (Android RunTime) virtual machine, that compiles applications at install time, using Ahead Of Time (AOT) (Ahead Of Time) compilation.

To have accurate energy measurements that reflected only the computational effort of keyboards, we reduced Android to the minimal number of processes possible, so that there are no non-necessary processes consuming energy during the studies. In fact, the considered Nexus 5 devices both run a customized system image, obtained by performing root-access changes in the 6.0.1 official version, which was reduced to the minimal set of apps/processes needed to conduct our study, being able to reduce the minimum number of processes running from 8 to 3. Moreover, we restored the boot image between different keyboard executions.

At the system level, precautions were taken to ensure that all external communication interfaces were disconnected. Thus, no Subscriber Identity Module (SIM) cards were inserted in the devices, and all network and sensor interfaces (Wi-Fi, Bluetooth, GPS, NFC, etc.) were turned off, as well as the speakers. Furthermore, we also executed the App Under Test (AUT) in immersive mode, in order to hide the system bar and minimize the impact of eventual events animations associated with it. All tests were also performed with the screen at the same brightness level.

Energy Measurement: Measuring software power consumption is a nontrivial process (**PintoAcm**), also because of the difficulty in reducing external interference (commonly called the Hawthorne effect) to the process to be monitored. Since this work aims to estimate the consumption in order to minimize this interference, some precautions had to be taken, from the system image where the tests were performed to the Profiler configuration and the testing procedure itself. There are many factors that might have impacted the performance of the KUT and the system itself, which can be internal or external to the keyboard application. Table 28 reports some of these factors that we identified and tried to minimize.

Table 28: External and internal factors that can impact keyboard performance

External	**Internal**	
Hardware	Auto-correction	Suggestions
OS version	Sound and Vibration	Emojis
Typing rate	Theme/Layout	Multi-Language support
System Load	Animations	Auto-capitalization
System Language	Gestures	Other user preferences

To measure the energy consumption of the virtual keyboards we used the Trepn energy profiler (Qualcomm Innovation Center, 2016). Although there are currently several alternatives to estimate and measure the energy consumption of applications on the Android (Chowdhury et al., 2019; Couto et al., 2014; Nucci et al., 2017a) platform, we have selected Trepn since it adapts perfectly to the device and the procedure we intend to implement. This tool is a software-based solution developed by Qualcomm, that works on devices with Snapdragon chipset-based running Android, as is the case for our two Nexus 5. Trepn can be used to profile hardware usage (like GPS, WiFi and others), resource usage (memory, CPU) and power consumption of both the system or standalone Android applications. This system does not need external hardware devices, as it gets its power readings from the power management Integrated Circuit (PMIC) and the battery fuel gauge software. Moreover, Trepn reportedly produces accurate estimation (Bakker, 2014), virtually identical to those obtained by hardware-based solutions, such as Monsoon (Monsoon, 2021).

Input Text and Test Application: Because keyboards are not standalone applications, we needed a widely used application where the keyboard is the main input textual method. To reduce any possible noise due to background operations, this application should, however, have a limited number of features, and not rely on (for instance) network interactions or animations. Therefore, as AUT we chose the *Wordpad* application [2], which has a simple and clean interface, and can be used offline without any background task being constantly performing uncontrolled/unpredictable work, like searching for an internet connection.

As input text, we randomly sampled 100 words from a paragraph from the book "Harry Potter and the Goblet of Fire" (Rowling, 2000). This text consists in words with an average length of 6 characters and contains no punctuation. We removed the punctuation from the text since many keyboards have different behaviors when a dot or comma is pressed. Also, in a preliminary evaluation of the real users' study, they complained that the fact of selected keyboards had different places and layouts to show and select

[2]Wordpad application: https://wordpad.br.aptoide.com/app *(visited on June 25, 2024)*

the required punctuation and that affected their writing performance.

Automate Keyboard Inputs: In order to exercise the five keyboards, we needed to define a strategy for automatically writing a pre-defined text input using each keyboard. Therefore, we stored the input text in an auxiliary file, and searched for a tool that could read its content and simulate writing it on *Wordpad*. For this purpose, we used the Android View Client (AVC) framework[3]: a Python-based tool that evolved from *monkeyrunner* ("Monkeyrunner", 2020), and which allows to manage and interact with multiple devices connected to the workstation. This tool has been used in similar works before (Cruz & Abreu, 2017), and uses ADB and the *input* command to simulating touches, taps, swipes and text input, among other events.

Running Keyboard and Collecting Energy Metrics: Finally, for developing the testing workflow, we reused the AnaDroid framework (Rua et al., 2019). We reused its tools and workflow to define the execution procedure, gather relevant metrics during the execution of the tests and generate reports from the profiled data obtained by Trepn.

5.2.3 Experimental Procedure

The energy consumption of each of the considered keyboards was studied using two approaches: *(i)* using a fully automated approach based on the tools described before, and *(ii)* by performing an empirical study, with real users that tested the keyboards under controlled conditions. We chose these two approaches since we aimed at determining the potential gains that could arise from choosing the most energy efficient keyboard, but also to study the impact that the human factor has on such gain.

Automated Approach

As stated in Section 5.2.1, virtual keyboards offer a wide range of features, but the core component is typically the same, as is the possibility to turn on/off more advanced features. From animations, sounds, emojis, and gestures, to complex natural language processing mechanisms (with word suggestions and correction features), all the considered keyboards offer these features with more or less complexity.

Given the high level of customization that this type of application offers, it was necessary to determine real usage scenarios that could be replicated on all keyboards, and in which it was fair to establish comparisons between them. Thus, two different usage modes have been defined:

[3]AVC: https://github.com/dtmilano/AndroidViewClient *(visited on June 25, 2024)*

- **Minimal Mode:** In this mode all possible features are turned off. This usage scenario intends to represent the most minimalist execution scenario possible, which resembles the functionality offered by a physical keyboard, and reduces to the minimum the computational effort of the keyboard application. It also assures that all keyboards are tested under the same conditions.

- **Default Mode:** This mode represents a usage scenario where the enabled features are the ones provided by each keyboard at the moment of its installation. It contains the features that developers defined for that version, which is the ideal keyboard execution scenario. The features activated by default, for all of our five keyboards, include word prediction and suggestion, auto-correction, among others.

In order to configure keyboards in the minimal mode, it is necessary to manually disable each of the features, using the KUT settings. Table 27 shows the features that are automatically activated in default mode for each of our five keyboards: they are marked with $'E'$. The single exception is the Swiftkey keyboard, which needs an internet connection to download the dictionary of the selected language and manually activate it. For this particular case, we used a Wi-Fi connection to download the dictionary and then we rebooted the image and wiped the device caches. Since the customized image doesn't contain other services (like Google Play Services or a browser) in the background waiting for network connections, we ensured that no other resources were downloaded.

To perform our first study, where the five keyboards are exercised both in minimal and default mode within the Wordpad app, we use the methodology specified in Algorithm 2.

As discussed in Section 5.2.2, we use a simpler Android boot image, which reduces to the minimum the number of background processes. Thus, we start by loading that boot image to the Nexus device. Then, for each of the (five) keyboards, a keyboard is installed in the considered mode (minimal or default). If in minimal mode, it is requested an additional manual configuration to turn off advanced features. After having the KUT installed and configured, the input text is loaded, along with the mapping between each letter and the corresponding coordinates in the KUT. The use of these pre-calculated key coordinates for each keyboard is required, although the Input program offers the possibility to write text without explicitly sending the coordinates of each key. However, this option doesn't correspond to the option of pressing individually each key with a touch, leading to different keyboard behaviors (for instance, not showing keypress animation or not performing auto-spacing when touching the suggestion bar).

Once everything is settled, we repeat the text writing 25 times using the KUT within *Wordpad*. Each

Algorithm 2: Text input simulation algorithm

```
loadImage()
for k : keyboards do
    installKeyboard(k)
    if mode == "minimal" then
        manuallyDisableFeatures()
    for i=0;i<25;i++ do
        text = loadTextToWrite(textfile)
        wordsCoords = loadKeyboardKeyCoordinates(text)
        wipeWordpadCache()
        initprofiler()
        openWordpad()
        openKeyboard(k)
        logDeviceState()
        startProfiler()
        writeTextInWordpad(k, wordsCoords )
        stopProfiler()
        logDeviceState()
        closeKeyboard()
        closeWordpad()
        exportProfiledData()
        shutdownProfiler()
    restoreImage()
cleanDalvikArtCache()
generateResults()
```

of these executions consists of the following steps: the *Wordpad* cache is cleaned, to ensure that there is no text related to previous tests that may cause biased data. The Trepn service is then started, and the Wordpad is initialized. The keyboard app opens and the device resource status is collected (number of processes running, CPU and RAM usage, among others) just before starting the monitoring process ('*startProfiler ()*'). This step consists of using the Trepn states to temporarily limit the beginning of the process. The next step consists of automatically writing the text. For this step, the *Input* program is used through ADB, so that each key is pressed to form the words of the input text. Finally, the monitoring process is finished and the device status is recorded again. The AUT and KUT are closed, the data from the monitoring session is exported to a Comma-Separated Values (CSV) file, the profiler is turned off, and the data is exported to the workstation. Finally, after the 25 executions of a keyboard, the boot image is restored and the device's caches are cleaned.

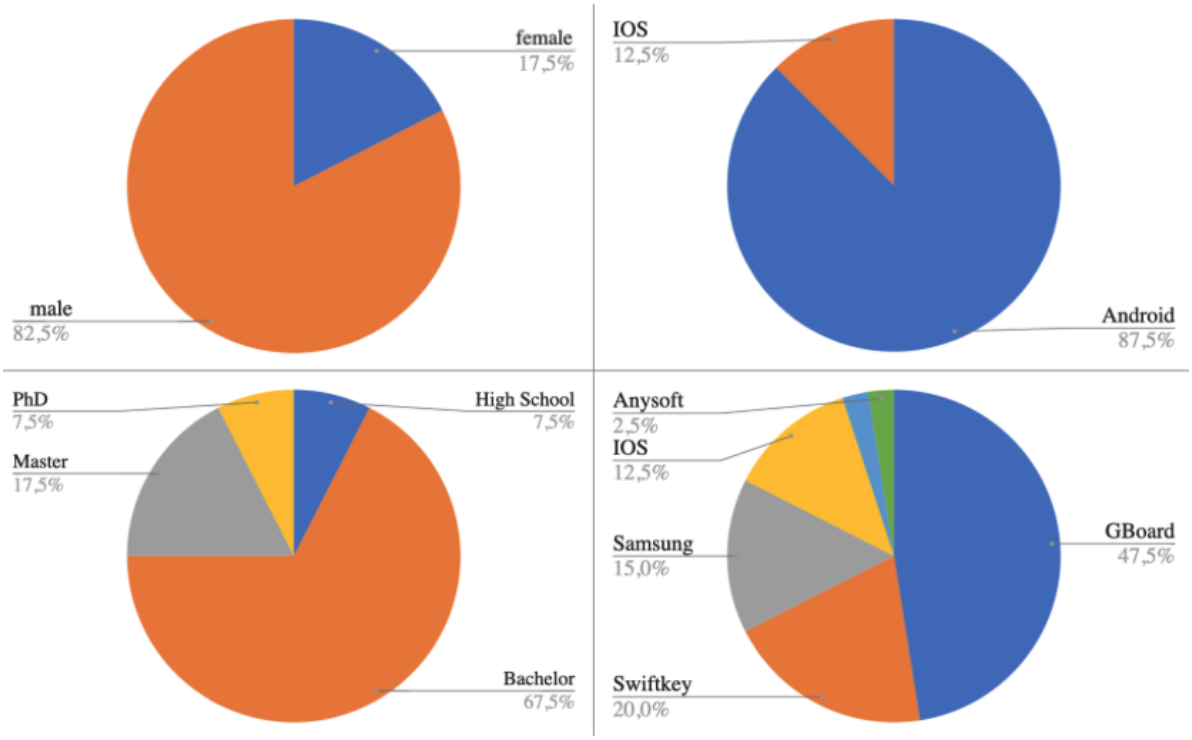


Figure 42: Statistics of the volunteers used in our study

5.2.3.1 Empirical User Study

As described in the previous section, it is possible to instruct a virtual keyboard to write a pre-defined input text. When exploring the energy impact of keyboards' advanced features, however, a similar automated process would be infeasible to apply in all five keyboards. For example, automating the word prediction feature is a complex task: keyboards use different strategies for displaying word suggestions: some only show three predictions, while others display a pull-down list, sometimes in different positions within the keyboard. The position of the best suggestion within the list also varies between keyboards, thus increasing even the complexity and computation overhead of an automated approach.

Thus, to rigorously study the energy impact of these features, we performed an empirical study with 40 real users. The volunteers that participated in our study were aged between 21 and 36, with 17.5% being females and 82.5% males. Only 1 volunteer has the same mobile device and 20% of them used one of the keyboards evaluated in our study, the Swiftkey keyboard. 75% of users run their device predefined keyboard, and 87.5 use the Android operating system. The wide variety of keyboards used by volunteers is valuable for our study. This ensures that they are unconsciously not privileging a specific keyboard, either because they are used to that keyboard or because of personal preference. Statistics characterizing the volunteers of this study are presented in figure 42.

For our study with real users, we considered the two best keyboards in terms of energy performance from the previously described study. Moreover, we divided the users into 4 groups of ten volunteers, so that each group evaluates 1 keyboard in a specific execution scenario. The two scenarios are:

- **Minimizing Suggestions:** For each of the keyboards, 2 distinct groups of users are asked to transcribe a given text, without resorting to the use of suggestions. With keyboards in the default state, users should write all words. This mode aims to represent the scenario in which the keyboard user prefers to write the whole word instead of using the suggestions.

- **Maximizing suggestions:** For both keyboards, groups of 10 users have to transcribe a given text, while maximizing the use of keyboard suggestions, which is in default mode. This mode aims to represent the scenario in which the user prefers to always look for the suggestion instead of having to write the full word.

The process was replicated in the same Nexus device used in Section 5.2, and every user transcribed the exact same text. In order to have more strict control over what users were doing before and during the process (like starting to type before the start of the profiler) the same setup as in the procedure referred to in 2 was used, with the necessary changes. The resultant procedure is described with algorithm 3. The main difference is the fact that 2 boot images were used, almost the same used in the automatic study, but each one containing one keyboard pre-installed and ready to use. The image and caches were restored between each test, to prevent the keyboard's prediction algorithm from adapting and possibly overfitting the input text. The start and end of the profiler is now delimited directly by the user and the step '*writeTextInWordpad()*' has been replaced by the process of the user writing the required words.

Finally, the naive automatic procedure developed to simulate typing text and using suggestions is also very similar to the one developed described in algorithm 2. Since most of these keyboards (with the exception of GBoard) are not open-source, without instrumenting the source code or having access to the view hierarchy, it is not possible to obtain the suggested words in real-time and with negligible computational cost. Thus, we decided to manually measure the effectiveness of each keyboard prediction mechanism in one single test and store this information in a map data structure. For each word of a given text, we associated the target word with the number of characters it was necessary to type in order to appear the word in the suggestions bar. We also evaluated for each keyboard the approximate coordinates of the place where typically the most probable suggestion appears. Using this pre-calculated information,

Algorithm 3: Real User Text input algorithm

```
for k : {GO, Swiftkey} do
    loadImage(k)
    for u : users do
        wipeWordpadCache()
        initprofiler()
        openWordpad()
        openKeyboard(k)
        logDeviceState()
        startProfiler()
        realUserTextInput()
        stopProfiler()
        logDeviceState()
        closeKeyboard()
        closeWordpad()
        shutdownProfiler()
        exportProfiledData()
        restoreImage(k)
        cleanDalvikArtCache()
    generateResults()
```

we performed 25 tests using the same procedure of algorithm 2, but with a different text input procedure, that can be resumed in the algorithm 4.

Algorithm 4: Naive text input algorithm

```
for word, nCharsToWrite : wordMap do
    for i=0;i<nCharsToWrite; i++ do
        typeChar( word[i])
    touchSuggestionBar(suggestionCoords)
```

5.3 Analysing the Energy Consumption

In this section, we present in detail the results obtained in the studies described in the previous sections. Section 5.3.1 contains the results obtained with the automated study, while Section 5.3.2 presents the results obtained with the empirical study with real users.

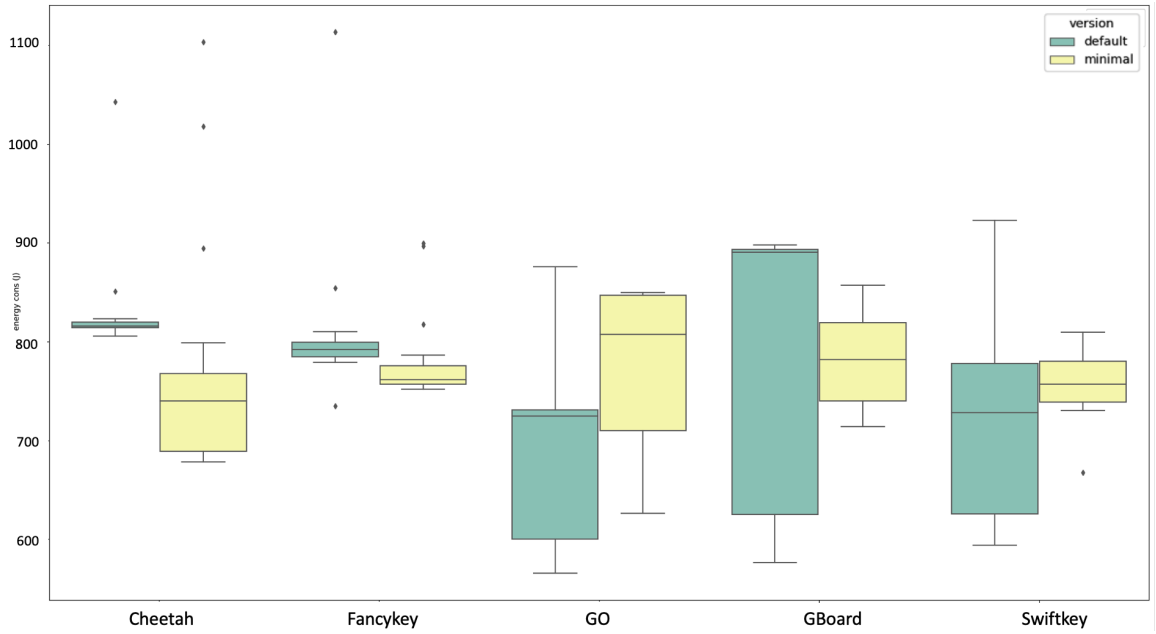


Figure 43: Energy consumption box plots of all keyboards

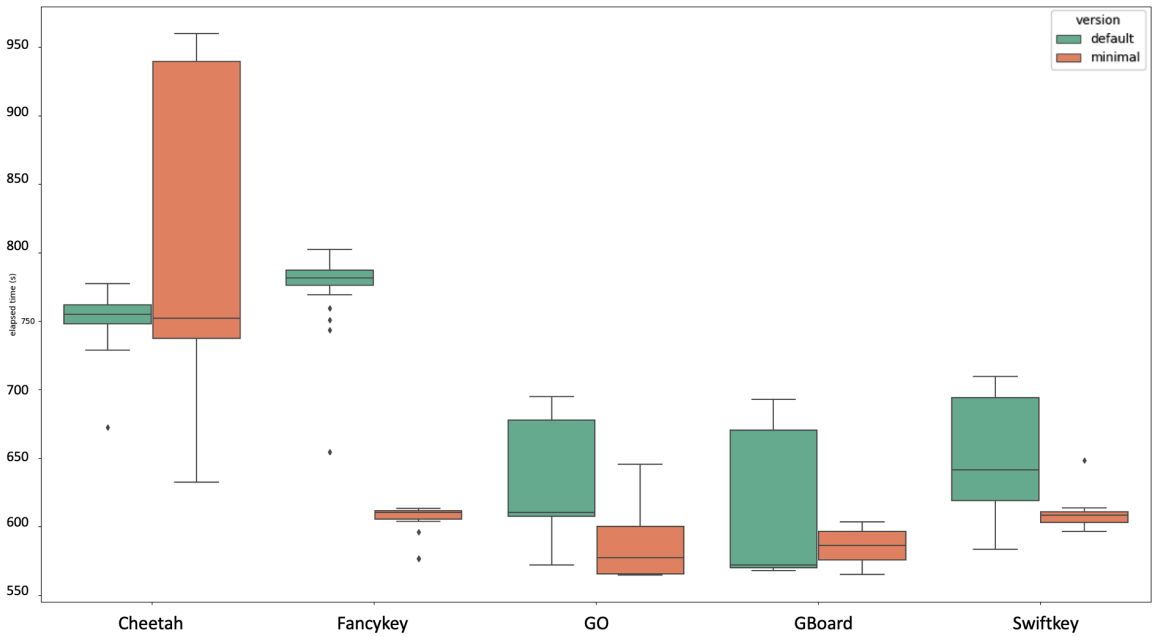


Figure 44: Elapsed time box plots of all keyboards

5.3.1 Default and Minimal mode

The overall results for the automated experiment are depicted in Figures 43 and 44. The former illustrates the energy consumption for all 5 keyboards, while the cllatter contains the values for the elapsed time. Each pair of box plots serves as a comparison between the 25 test executions of both testing modes: default (with only the default features of each keyboard enabled) and minimal (with all features disabled).

These plots show statistical differences among KUT energy consumption. The keyboards with lower median and average values of energy consumed in default mode were Go (median of $724.6J$ and average of $687.5J$) and Swiftkey (median of $727.9J$ and average of $715.98J$). Gboard and Cheetah are the most energy-greedy keyboards, with median values of $890.7J$ and $816.1J$.

When looking at the results for the minimal mode, the most energy-efficient keyboards were Cheetah and Swiftkey, with median values of $740.2J$ and $756.8J$, and average values of $757.2J$ and $757.6J$, respectively. Swiftkey appears as one of the most energy-efficient keyboards, independently of the testing mode. On the other hand, the Cheetah keyboard was by far the one that benefited more with feature disabling, while the Go keyboard actually increased energy consumption with it, being the keyboard with the highest energy consumption in the minimal mode.

Examining the elapsed time values, illustrated in Figure 44, together with the energy consumption values, allow us to verify how much related is energy efficiency to performance. We can confirm that, for the minimal mode, every KUT obtained a lower median value than the ones obtained in the default mode. The most performance-efficient keyboards were GBoard in the default mode and GO in minimal mode, which were also the most energy-greedy ones in the same modes. The opposite occurs, for instance, for Cheetah in minimalist mode or Swiftkey in default mode, being that in these modes they were the most energy efficient, but in terms of performance, they were the most inefficient.

As mentioned in Section 5.2.3, a wide range of metrics were collected during the execution of the tests. In order to assess the potential impact that each of them may have on energy consumption and even on the consumption of other results, Figure 45 presents a correlation matrix that aims to illustrate these relationships. The information from the tests performed in a single dataset was aggregated and the metrics that had no or little (cor)relation between the others were filtered. In this Figure, it can be observed that the features more directly correlated (marked with a deeper blue tone) with the energy consumed are CPU 1 frequency, CPU 2 frequency, CPU Load Normalized (load normalized across all cores, according to the maximum CPU frequency) and battery power. The least correlated features are CPU 2 and 4 load, CPU load (across all cores) and elapsed time (coinciding with the data observed in Figures 43 and 44).

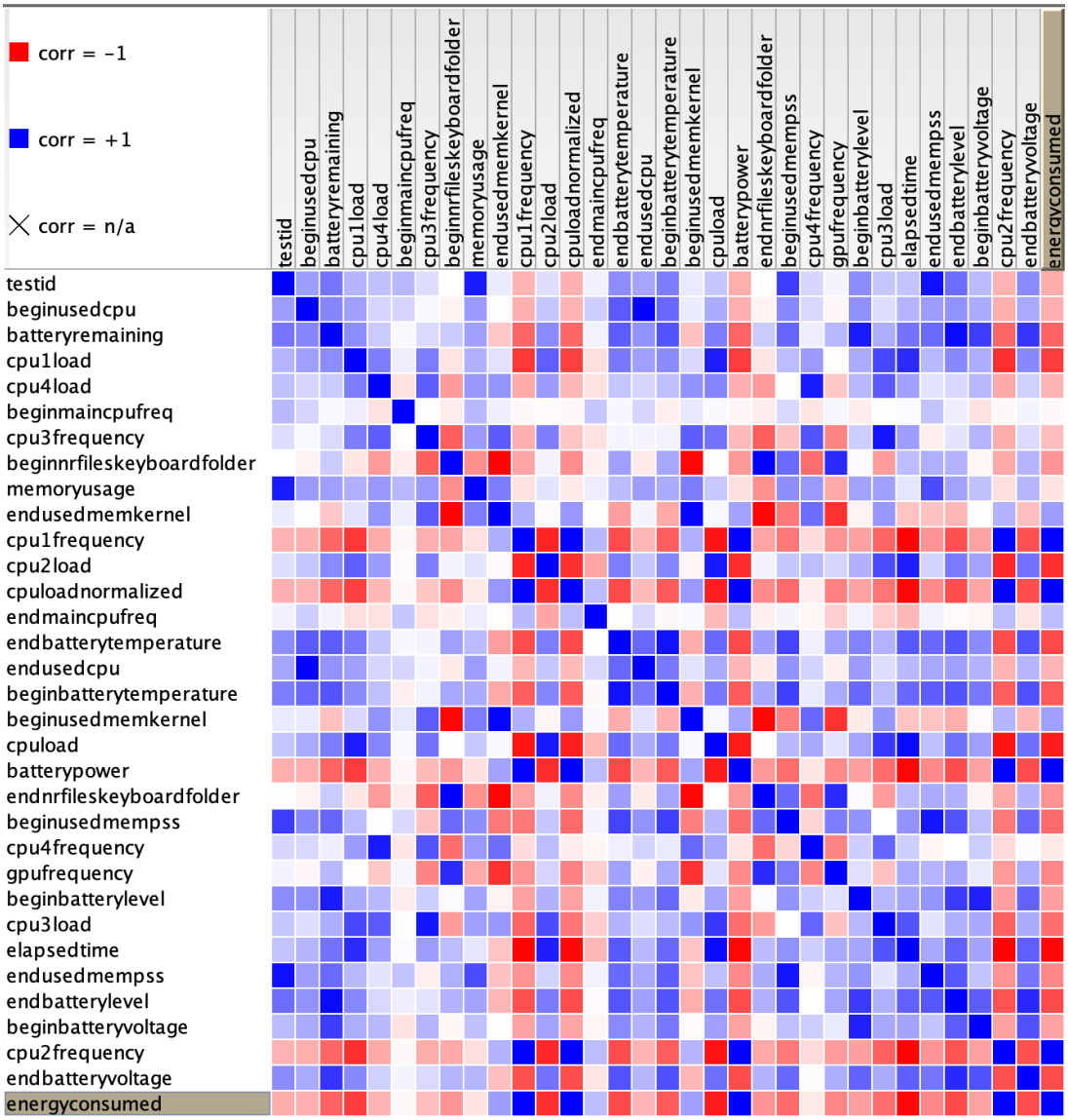


Figure 45: Correlation matrix of the sampled data during tests execution

5.3.2 Real User Interaction and Simulation

Following the same structure of the previous Section 5.3.1, for this study we also present the main results with box plots. However, we only show the results for the 2 keyboards that are cross-platform and were the most energy efficient in the default mode, according to the data contained in Figure 43: Swiftkey and GO keyboard. Since we wanted to test each mode several times, if we considered all of the 5 keyboards, we would end up with only 4 tests for each mode. Furthermore, we decided to select the two most efficient in default mode. The results of the energy consumed and execution time in each of the 10 executions of each configuration can be seen in Figures 46 and 47.

Regarding energy consumption, we can see that using more features (in this case only word prediction)

produces different results on these 2 chosen keyboards, similar to what happened in the automated study (depicted in Figure 43). According to the results, looking at the median, the energy consumption slightly increases when using suggestions in the GO keyboard, but not according to the 1st and 3rd quartile. It also led to users taking more time to complete their tasks, as the values in Figure 47 suggest. As for the Swiftkey keyboard, the suggestions mode was the most energy efficient looking at the median value, but also the most time-consuming.

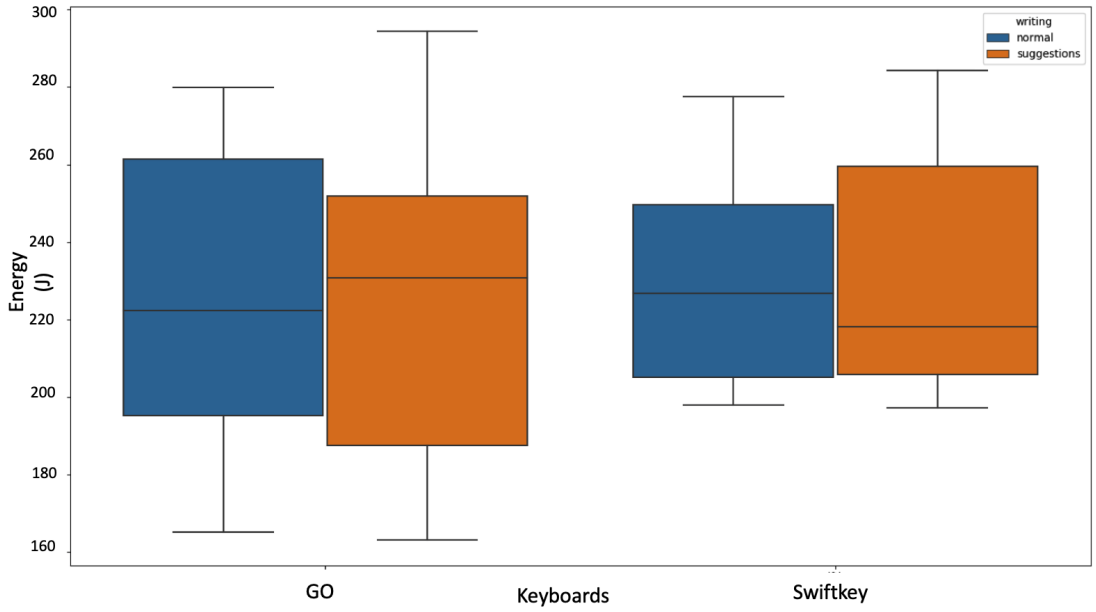


Figure 46: Energy consumption box plots of the 2 keyboards tested with real users

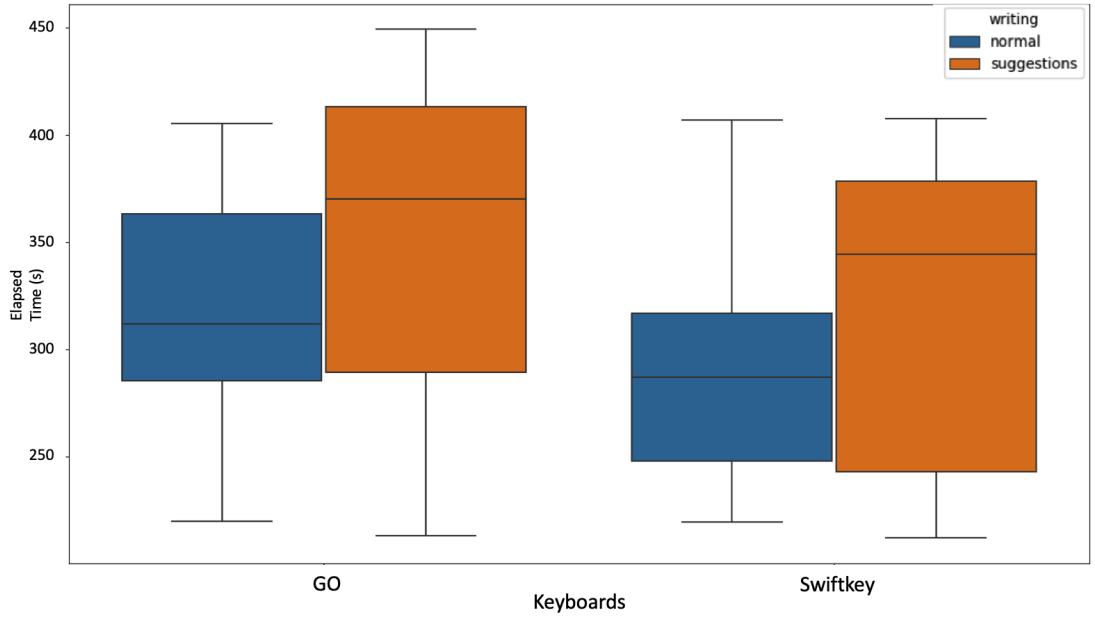


Figure 47: Tests Elapsed time of both keyboards tested with real users

The automatic naive procedure was also performed, in order to simulate the use of word suggestions by real users. This experiment, which was performed with the default features enabled, had quite different results from those obtained with real users, as can be seen in Figures 48 and 49. Using this automatic procedure, looking at the median values obtained for each of the keyboards, there was an average gain of $311.74J$ (a gain of 43%) per test for the GO and $297.9J$ (a gain of 40.9%) for the Swiftkey. Regarding execution time, the tests took an average of 243 seconds less for the GO and 222.2 seconds less for the Swiftkey.

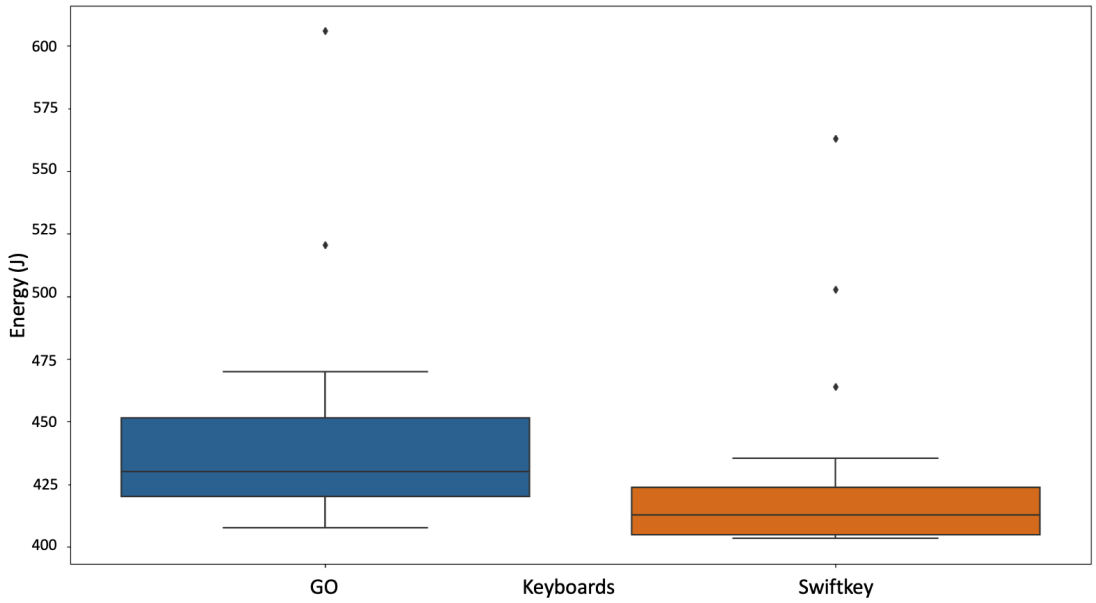


Figure 48: Energy consumption box plots of the 2 keyboards for the naive automatic procedure

5.4 Discussion

In this section, we present the analysis and discussion for each of the studies described previously. For the automated experiment, we analyze the results obtained for each one of the testing modes (with *minimal* and *default* features). We describe and provide details of the statistical evidence used to retrieve meaningful conclusions regarding the testing modes and individual keyboard's energy consumption. Considering the real user experiment, we evaluate the gains associated with word prediction features and the impact of some aspects intrinsic to each user on energy consumption. We also compare the procedure executed with real users with the naive automatic version, described in Section 5.2.3.1.

By examining the data illustrated in Figure 43, it is possible to observe that there are significant differences among the energy consumption of the different KUTs. For example, looking at the median

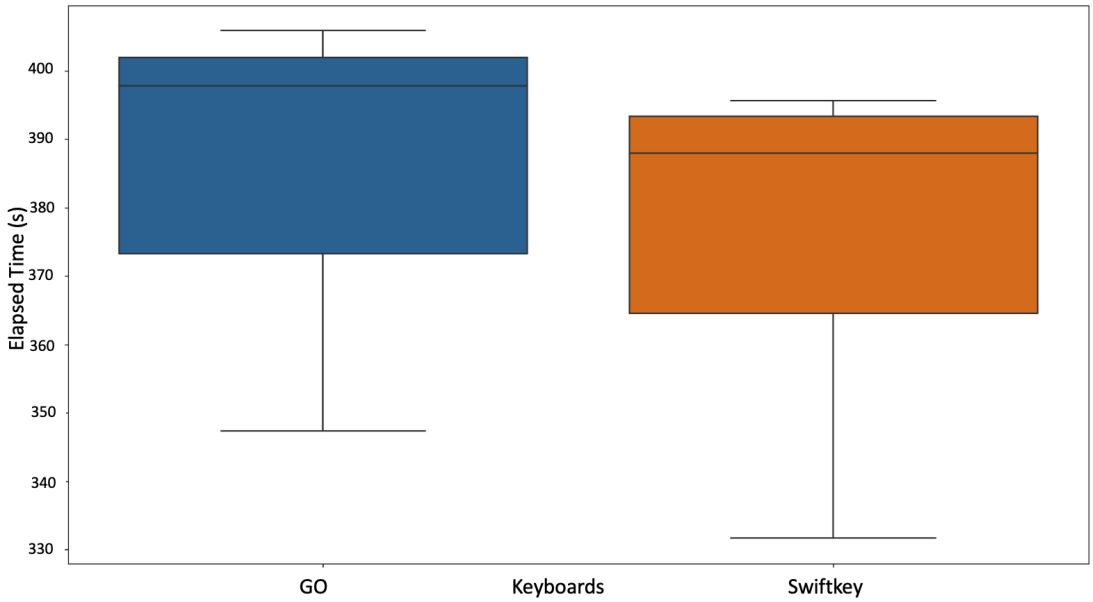


Figure 49: Tests Elapsed time of both keyboards for the naive automatic procedure

values in both testing modes, and comparing the most efficient keyboard with the least efficient, it is safe to conclude that there are significant gains. For instance, in the default mode, the GO keyboard has an efficiency gain of more than 18% compared to the Google keyboard. In the minimal mode, the gain of the more energy-efficient keyboard (Cheetah) compared to the more energy-greedy (GO) is lower (8.3%).

The illustrated data gives us a clear indication of what keyboard behaved as the most energy-friendly for each scenario. Nevertheless, it is important to prove if there is statistical evidence behind these observations, that is, if the energy consumed by each keyboard is statistically different from the consumption of the others. Thus, we tested the following hypothesis, for both default and minimal modes:

$\boldsymbol{H}_0$: The energy values obtained for each KUT follow the same distribution.

$\boldsymbol{H}_1$: The energy values obtained for each KUT do not follow the same distribution.

The evaluation of this hypothesis and consequently rejection of the null hypothesis H_0 (allows us to answer to **RQ5.1**. If a statistical test indicates that two datasets follow different distributions, and we can see a clear tendency favoring one of them, we can conclude that indeed there is evidence that there are differences in energy consumption between the KUT's. To understand if there is an overall significant relevance between the distributions of the KUT's energy values, first, we analyzed each one of the datasets of each keyboard, to evaluate whether these groups are parametric or not. Since only one

keyboard presented values followed a Gaussian distribution, we assumed a non-parametric behavior of the groups and ran the Kruskal Wallis H test Kruskal & Wallis, 1952.

The Kruskal-Wallis test can be used to determine whether more than two independent non-parametric samples have a different distribution. We ran this test twice, For each keyboard mode, we evaluated the p-value considering $\alpha = 0.05$. The p-values of 0 and 0.026 for the KUT's samples in default and minimal mode, respectively, obtained for this test, allow us to reject $\boldsymbol{H}_0$, and respond to **RQ5.1**, concluding that indeed there are statistical differences in energy consumption between the selected keyboards, in both modes. This result can be explained by the fact that these keyboards are complex applications and differ in many aspects, from layout aspects, animations or word suggestion algorithms. Consequently, different characteristics and feature implementations might have different impacts on energy consumption.

As described in the previous Section 5.3, in the case of GO and Swiftkey, these keyboards consumed more energy in the minimal mode than in the default mode. GO is the most evident case, with a median value of 10.2% higher. Swiftkey, however, registered a less significant difference, of only 3.8%. The other keyboards decreased their energy consumption in the minimal modes. The most significant case is the Cheetah, with a decrease of 9.3%. This may be due to the fact that this keyboard uses a greater number of animations by default, such as key-pressing animations or an animated background. Since these animations demand additional computation to be rendered, this sharp descent can be justified by the absence of these animations in minimal mode.

In order to assess whether or not there is a significant and statistically supported difference between using the different modes, we need to perform another statistical test. A suitable test for this purpose is the Mann-Whitney U test (Mann & Whitney, 1947), since is capable of evaluating whether two independent samples were selected from populations having the same distribution. This test was performed for each keyboard, in which the following hypotheses were evaluated:

$\boldsymbol{H}_0$: The energy values obtained for both modes follow the same distribution.

$\boldsymbol{H}_1$: The energy values obtained for both modes don't follow the same distribution.

For the non-parametric groups (all except GO), we used the Mann-Whitney U test (Mann & Whitney, 1947), considering $\alpha = 0.05$. The results obtained are detailed in table 29. The only keyboard that doesn't have significant statistical differences is Google, with Cheetah being the strongest null hypothesis rejected. For the Go keyboard, a different statistical test was performed, as its samples follow a normal distribution, according to 2 different tests performed (Shapiro Shapiro & Wilk, 1965 and D'Agostino and

Pearson's D'Agostino, 1971). In order to evaluate the same hypothesis, for parametric data, the alternative is the T-test (Welch, 1947), a two-sided test for the null hypothesis that 2 independent samples have identical expected values. For the same value of α, we obtained a p-value of 0.0, which also allows us to reject H0.

Table 29: p-values and hypothesis test decision for each of the KUT with non-parametric samples

Keyboard	**p-value**	**Decision**
Cheetah	0	Reject H0
Google	0.076	**Failed to reject H0**
Fancykey	0	Reject H0
Swiftkey	0.036	Reject H0

The results obtained for both parametric and non-parametric samples, combined with the comparison of the values presented in the plots from Figure 43, allow us to answer **RQ5.2**: there are in fact cases where it is possible to save energy by disabling keyboard features. We observed that for Cheetah, Google and Fancykey. Go and Swiftkey surprisingly do not reflect this behavior. A possible justification for this may be that the developers were more concerned with optimizing the default activated features, or even that such features contain performance improvements that are removed with a feature being disabled.

The data from the correlation matrix in Figure 45 help to understand the correlation of the metrics surveyed during the test execution process. With regard to energy consumption, there are expected correlation values, such as CPU frequency, and power, since Android multiplies the CPU time for each application by the voltage required to run the CPU at a specific frequency (Google, 2022c). We also can observe that energy consumption of the keyboards decreased along the 25 tests (each test was identified by its index, column 'testid'), given the slightly red value present in the figure, as well as the elapsed time of the tests. This observation is consistent with a previous study that shows that less execution time does not necessarily indicate less energy consumption (Hao et al., 2013).

As for the results obtained with the use of suggestions, the results were somewhat unexpected in the study with real users. The word prediction and suggestion mechanisms were developed in order to save keyboard users time and effort when writing text. However, the results obtained with the 20 users who used the keyboards trying to maximize the use of suggestions ended up going against this assumption, as they took longer to write the requested text (figure 47. for the median value of each group, the Go keyboard had an increase of 16.6% and the Swiftkey of 15.7%. In the case of GO, it ended up consuming more energy, but the opposite happened for the Swiftkey (which goes against the results illustrated in the table

43, in which it consumed more energy when fewer features were used (figure 46. Taking into account these data, the answer to the question **RQ5.3** would be negative, however, during the execution of the tests with the real users, it was observed that they always showed great concern in actually observing each letter inserted if the suggestion appeared in the respective bar. Also, they didn't recognize the text that they were typing and this study would probably have different results if they were writing a text they knew or had previously thought about.

Nevertheless, the automatic naive approach performed to simulate the same procedure, but without resorting to real users, obtained concordant results with the assumption that the use of suggestions can increase writing efficiency, both in terms of energy and time. Since the default mode of the first study corresponds to the automatic version of the mode that minimized the suggestions in the studies with real users, we can compare those results with the ones with this naive procedure. By analyzing the results of this procedure, described in figures 48 and 49, we can observe that the median of the values obtained for both keyboards using suggestions is significantly lower in terms of energy and time than the ones obtained in the first study in default mode (figure 43 and 44. More specifically, for the median values obtained for each of the keyboards, a gain of 311.74 Joules(43%) per test for the GO and 297.91 (40.9%) for the Swiftkey.

Given the results obtained with the real users and the naive approach, the answer to **RQ5.3** is: Sometimes. The study with real users has controversial results, where the use of word suggestions had a negative impact on energy consumption and elapsed time. However, the same study realized under a different scenario, like writing 100 random words of user choice, could have different results than the ones obtained with the transcription of the selected text. Nevertheless, the results of the naive automatic approach satisfy the assumption that in fact, the use of this kind of feature can help users to save energy and time while typing text in virtual keyboards.

5.5 Threats to validity

Android is a multi-process operating system, where multiple processes and applications run "simultaneously". To ensure that during the execution of each test, only the intended application is running on the device, affecting its consumption, we registered its state before and after each test. Moreover, we

executed the automatic tests in customized images (using root privileges) where such processes were disabled. Testing in customized rooted images is a usual approach followed by research community (Banerjee & Roychoudhury, 2017; Linares-Vásquez et al., 2014; Morales et al., 2018; Nucci et al., 2017a). The device selected for this work is usually used by the research community to conduct performance studies (Cañete et al., 2020b; Rua et al., 2019) due to the fact that does not have many pre-installed 3rd party-apps. Moreover, we run Android version 6.0.1, which uses the ART virtual machine. However, there are several compiling options available for this version for applications and system source code, and for 3rd-party apps, by default, this version uses the quicken option, which only optimizes some Android bytecode instructions to achieve better interpreter performance. Furthermore, to minimize eventual optimizations that the system might perform for both AUT and profiler code, we restore the boot image between sets of tests for each keyboard. Thus, in the second study, we also wipe the Dalvik/ART and system cache, in order to delete eventual optimizations and execution profiles provided by Just In Time (JIT) compilation. To minimize external interference, the automated tests for the default and minimal versions were performed with the minimum value of brightness in order to minimize the impact on power consumption. For the user study, we used a level of 50% of brightness so that users could see the screen.

As for factors intrinsic to the KUT, in order to establish fair and adequate comparisons among keyboards, some precautions had to be taken. All keyboards used the same language (EN-US), the language of the system in which they were tested and the text to be inserted since the chosen language has implications for the keyboard layout and dictionaries. Some keyboards also have some peculiarities with which we had to be cautious and ascertain their impact. For instance, Go Keyboard offers a memory boost option, enabled by default. When tested in minimal mode, we evaluated the eventual gains in using this option, but the energy values obtained were slightly higher than the ones obtained with this option turned off. So we ended up not using it in the minimal testing mode.

As for the results obtained, these may differ between devices, Android versions or the keyboard application. Regarding the automatic procedure performed, there was a noticeable delay in the communication between the workstation and the Android device, caused by the type of interface (ADB), communication protocol and system version. In a test carried out on newer devices and emulators, we found that the typing rate using the same automatic procedure was substantially higher, which can eventually lead to different results. However, we wanted to carry out the study on the currently most used version of the platform.

Regarding the study performed with real users, the chosen case studies (maximizing and minimizing

the use of suggestions in the transcription of a text) may not have been the most suitable to assess the advantage of using predictions. The language used for keyboard and text dictionaries to be entered may also have implications. Although all participant who wrote the text learned the language since elementary school, the results could possibly be more satisfactory if the language chosen was their native language. The process of transcribing words and the indication that they should not make mistakes in writing meant that there was a delay in reading the text to be transcribed and verifying that the word they had just written was right. This delay would probably be inferior if they knew the text beforehand and/or did not have to transcribe it.

5.6 Summary

In this work we have explored the performance of virtual keyboards in the Android platform, focusing on their impact on energy consumption. We considered 5 widely used keyboard applications and designed an experiment to evaluate their energy consumption, in different execution scenarios. We performed different studies using both real users and automatic procedures to simulate real user interaction. The results from this experiment show that indeed there are differences in the energy consumed by the selected keyboards, and switching keyboards or enabling/disabling settings promotes energy saving in most cases. In fact, replacing the most energy greedy keyboard with the greenest one has reduced energy consumption by 18%, and when advanced features of those keyboards (such as word prediction or animations) are turned off, the energy consumption is also reduced, in this case up to 9.3%.

The presented work opens the path for posterior works by demonstrating that there are significant differences in terms of energy footprint between widely used competing applications. For instance, a more extensive study can allow us to determine the best keyboard and the best combination of features that optimize energy consumption and, if possible, according to users' usability preferences.

6

Conclusions

Abstract: In this chapter, we summarize the research works presented and discussed in the previous chapters and we share our conclusions on the developed work. We look back at the research goals introduced at the beginning of this dissertation, providing an overview of how we addressed them. Finally, we present a few research paths that can be explored in the future, both by ourselves and by practitioners working in the field. The proposed research paths are extensions and/or improvements of the work presented in this thesis and other related research that can be conducted based on this thesis' findings, observations or tools.

Throughout this thesis, we have studied and extended the state-of-the-art approaches to creating and maintaining energy-efficient mobile software. We have detailed existing energy measurement techniques, defined energy performance guidelines, developed tools to help practitioners and researchers, and studied potential shortcomings of delivering energy-efficient software.

As discussed before, the main goal of the core contributions incorporated in this thesis was to address a lead research question, introduced in Section 1.1:

Main Research Question

What are the current limitations of the state-of-the-art approaches to improving the energy performance of the Android ecosystem, and how can these limitations be overcome by the development and scientific community?

Each one of the previous 3 chapters consists of an important step in order to answer this main research question. In order to define concrete objectives that allow us to address the main research question, in Section 1.1 we also presented 3 different research goals.

Research Goal 1

Develop novel tools that provide and facilitate integrated energy and performance analysis for Android software

In order to meet the first research goal, Chapter 3 presents two important toolsets that were created in order to address some of the limitations identified in the state-of-the-art. First, we presented *E-MANAFA*, a device-independent, model-based energy profiler that can be used by researchers and developers to analyze energy consumption on Android. Furthermore, it was designed to help developers diagnose energy hotspots of their applications and for researchers aiming to carry out empirical studies regarding energy performance. Secondly, in Chapter 3 we presented *PyAnaDroid*, an execution pipeline tailored for analyzing and benchmarking Android applications. It consists of a generic pipeline that can transform and analyze source code, build applications and execute them automatically, with a high degree of customizability. This tool was developed to be the *de facto* tool to perform empirical analysis on mobile software and can help the scientific community in the tasks of conducting, replicating, and validating empirical analysis experiments. Both *E-MANAFA* and *PyAnaDroid* are open-source, under continuous development and open for contribution. The tools were designed to be easily extendable to support other features and evolve the frameworks. *PyAnaDroid* also consists of a major research effort presented in this thesis that aims to meet the second research goal:

Research Goal 2

Propose procedures and guidelines to facilitate the replication and validation of performance analysis

In Chapters 4 and 5 we provide more steps in the direction of accomplishing the second research goal. In Chapter 4, we present the largest empirical study in the literature that considers dynamic execution and performance monitoring of Android applications. This study produced, among other contributions, a set of guidelines that can be reused in other studies and when developing or maintaining software. The empirical procedures followed in the studies presented in Chapter 5 and Section 4.2.1 are publicly available and can be applied to replicate automated dynamic analysis of other Android applications and/or software. In this sense, these are also contributions to meet the second research goal.

In order to meet the third research goal, in Chapters 4 and 5 we present several efforts that contributed with more knowledge regarding the impact on the performance of certain programming practices/patterns. We started by presenting a preliminary study on the impact of memoization on energy

consumption, which produced evidence that applying this optimization technique can result in significant energy savings. Also in Section 4.3, the presented large-scale empirical study allowed us to determine that out of the 15 software changes examined in our study, only modifications/enhancements in functionalities, bug fixes, refactoring, layout additions, and string incorporations significantly affect app performance. Upon reassessing the relevance of conclusions drawn from previous performance analyses, such as those by Linares-Vásquez et al.and the Android Lint Issues, we identified notable disparities in the performance of practices previously identified as detrimental. In the case of Linares-Vásquez et al.' study, our observations revealed that only 40% of the defined energy-intensive APIs had a discernible impact on app performance, while the rest had negligible effects. Additionally, our examination of Lint Issues disclosed that many of these problems did not manifest the expected negative influence on app performance. Furthermore, we determined that the severity levels assigned to issues labeled as performance-related exhibited minimal correlation with their actual impact on performance.

Research Goal 3

Study and document best practices and recurrent solutions that can be applied to enhance the efficiency of mobile applications and devices.

Finally, Chapter 5 also presents contributions that aim to contribute to meet the third research goal. This chapter presents a study that delves into the performance of virtual keyboards within the Android platform, with a specific emphasis on their influence on energy consumption. Five of the most utilized keyboard applications were selected and analyzed in distinct execution scenarios. The results from this experiment, among other conclusions, allowed us to conclude that there are significant differences in the energy consumed by the selected keyboards, and switching keyboards or enabling/disabling settings promotes energy saving in most cases. These results can impact the development and scientific community, by demonstrating that there is a lot of room for performance improvements even among the most used applications in the ecosystem and that assessing the performance of popular applications can help to advise users to save battery and aware developers about their applications performance footprint.

6.1 Future Work

In this Section, we present some recommendations for future research based on the work and research made in the scope of this thesis. We divide these recommendations into two distinct sections: a former

addressing further efforts that can help to improve and extend the contributions presented in this thesis (Section 6.1.1) and a former that suggests further related work that can lead to further improvements in the performance of existing software (Section 6.1.2).

6.1.1 Refining this Thesis

In this Section, we present a list of some possible direct extensions to the work presented in this thesis. For each of the previous 3 chapters, we will provide a set of possible efforts that could potentially provide significant contributions to the developed work and to this thesis field.

The efforts presented in Chapter 3 could be improved by exploring the following paths:

- **By comparing the accuracy of *E-MANAFA* with other energy profilers and monitors.** As referred in Section 3.2.4, the results reported by the tool have not yet been compared with results obtained with other tools. Moreover, it is presently challenging to provide a definitive assessment regarding the precision of the values generated by the tool. Nonetheless, there is a reasonable assumption that the results are accurate, given their reliance on widely-used services such as *batterystats* that are used by various profilers (Bouaffar et al., 2021; Google, 2021b; Nucci et al., 2017a) for estimating device power consumption. Collaborative efforts with existing Android profiling tools, such as those mentioned, could contribute to a more comprehensive understanding and validation of the power consumption estimates. As the tool evolves, continual scrutiny and validation against diverse use cases will enhance confidence in the reliability of the obtained results.

- **By extending *E-MANAFA* to support memory profiling**. Extending our profiler with integrated memory profiling capabilities could play a pivotal role in optimizing Android applications, by allowing its users to consider this key indicator alongside energy and execution time. Given the fact that memory-related issues, such as memory leaks or inefficient memory allocation, can significantly impact an application's responsiveness and stability, by employing a performance profiler tailored for memory profiling on Android, developers gain invaluable insights into the dynamic memory usage patterns and potential bottlenecks within their applications. This diagnostic information allows for the timely identification and resolution of memory-related issues, preventing degraded performance, crashes, or excessive resource consumption. The ability to pinpoint specific areas of code contributing to memory inefficiencies empowers developers to implement targeted optimizations, leading to more resource-efficient and responsive Android applications.

- **By conducting a more extensive study on the suitability/accuracy of state-of-the-art testing frameworks for several testing objectives.** Both studies presented in Chapter 3 provide a comparison between the automated execution of applications with two testing frameworks as illustrative examples for the results that both tools can produce. The results obtained in this chapter regarding the different frameworks considered

- **By integrating other software analysis tools into the *PyAnaDroid* pipeline.** Integrating other static and dynamic tools can extend the functionalities of the pipeline and its domain of applicability. For instance, the development of other Building blocks can bring support for cross-platform development frameworks and applications. This would significantly increase the domain of applicability of *PyAnaDroid*, since currently more than 20% of Android applications are developed using cross-platform development frameworks (Brain, 2023) and the performance analysis of these applications has gathered interest among the research and development community (Corbalan et al., 2018a; Oliveira et al., 2023).

The work presented in Chapter 4 could be improved by exploring the following paths:

- **By evaluating the impact of memoization on a larger scale.** To undertake this research, it is crucial to consider a large and diverse set of Android apps, representing various domains and usage scenarios, to ensure the findings are applicable across a wide spectrum. Rigorous experimentation, including controlled experiments, edge case evaluations, and scalability assessments, should be designed and considered to systematically measure the impact of memoization using key performance indications. The research outcomes can provide valuable insights into when and where memoization proves most effective, guiding developers in adopting this optimization technique on Android. applications.

- **By replicating the empirical procedure followed in Section 4.3 on more execution contexts.** Replicating this empirical procedure in more devices and sets of apps can help to reject, generalize, or particularize assumptions from the state-of-the-art regarding the performance of Android source code.

- **By investigating the research directions proposed in Section 4.3.3.** In this section, we presented some research directions that were hinted at by some of the results obtained during our experiment. The proposed research directions can have as a starting point the dataset produced

in our study. The research directions, as mentioned in Section 4.3.3, formalized as research questions, are the following:

- **RD1:** *How do different testing frameworks explore application performance?*
- **RD2:** *What is the most suitable testing framework to detect functional errors in applications?*
- **RD3:** *How does applications' performance evolve throughout their versions?*
- **RD4:** *What is the relation between semantic version changes and performance?*
- **RD5:** *Which types of functionality and bugfixes have more impact on Androids applications energy consumption?*

Finally, the research effort presented in Chapter 5 could be improved in the following ways:

- **By extending the analysis to more keyboards and execution contexts.** A more extensive analysis considering more keyboards, different execution scenarios, and different devices could help to evaluate the individual impact that each of the keyboard features may have on energy consumption and the reason for such impact. The study could allow us to determine the best keyboard and the best combination of features that optimize energy consumption and, if possible, according to users' usability preferences.

- **By producing a guide or recommendation system to aid in the selection of the most appropriate keyboard according to the results obtained and user preferences.** This effort would result in a contribution more user-centered, which could result in knowledge that could improve the efficiency of the Android OS experience. It could help to enhance factors such as optimized UX and Performance, time savings for users and consequently, also users' satisfaction.

6.1.2 Beyond this Thesis

In this Section, we present several ideas that can lead to further improvements and findings regarding the energy efficiency of existing Android applications and source code. These ideas are not directly related to the contributions presented in this document, but they can reuse the developed techniques, tools and procedures to achieve their results.

6.1.2.1 Evaluating the impact of Android cryptographic algorithms.

Cryptographic algorithms can be computationally heavy depending on the specific algorithm and the size of the data being processed. Some of these algorithms are computationally intensive and may require more powerful devices to perform the computations efficiently. The Android SDK itself provides several security providers, that implement several configurable primitives and algorithms that are available for all state-of-the-art Android devices. We propose a multi-objective performance evaluation of cryptographic algorithms in order to evaluate which configurations are most efficient according to different performance indicators. Previous research efforts considering the performance of different cryptographic algorithms, respective parameters (Montenegro et al., 2018; Salama et al., 2009, 2010), data types (Salama et al., 2010), libraries/providers (Montenegro et al., 2018) and devices or environments (Ramesh & Umarani, 2012; Salama et al., 2009) already exist. However, there is still no extensive study in the literature that analyzes the efficiency of the algorithms and providers available on state-of-the-art Android devices. Nor is there a study that considers possible trade-offs in terms of security or execution time. The proposed study can help developers develop more sustainable software and provide hints for further research in the field.

6.1.2.2 Evaluating the impact of using and running Large Language Model (LLM)s on Android.

With the increasing integration of advanced natural language processing capabilities into mobile applications, understanding the specific challenges and advantages associated with LLMs on Android is crucial. Our proposed research path suggests a systematic evaluation of multiple LLMs, assessing their impact on key performance indicators, such as execution time/responsiveness, memory usage, and energy consumption, and finding trade-offs between performance and accuracy of the models. Similar studies considering cloud or personal computer setups already exist (Dodge et al., 2022). However, for a mobile setup, no study evaluates the performance of using LLMs. The proposed study should consider a diverse set of Android devices and operating system versions to capture the variations in hardware capabilities and software environments.

6.1.2.3 Evaluating the energy efficiency of programming languages of the Android environment.

Currently, it is possible to develop software for Android using a wide range of programming languages, either by directly pre-compiling and executing the generated native code or by using virtual machines. Inspired by previous works (Pereira et al., 2017, 2021a), we propose the design of a large-scale study that evaluates the performance of Android software built with different programming languages. The results of this research effort could be materialized in a ranking of programming languages that could help developers select the most efficient language according to their requirements and contribute to a more sustainable Android platform usage.

6.1.2.4 Evaluating the Impact and Feasibility of Running Blockchain Nodes on Android.

With the increasing ubiquity of mobile devices and the growing interest in decentralized technologies, understanding the practicality and implications of running blockchain nodes on Android is very pertinent. We propose a thorough examination of the impact and feasibility of deploying different types of blockchain nodes on the Android platform. The findings of this study provide valuable insights into the trade-offs involved in running blockchain nodes on mobile platforms, offering guidance to developers, blockchain architects, and researchers. The implications extend to potential use cases, user adoption, and the broader integration of decentralized technologies into the mobile ecosystem. This evaluation should rely on key performance metrics, such as processing speed, memory usage, and energy consumption, and consider other crucial non-functional requirements such as user experience or sustainable impact.

Bibliography

Agosta, G., Bessi, M., Capra, E., & Francalanci, C. (2012). Automatic memoization for energy efficiency in financial applications [IEEE Int. Green Computing Conference (IGCC'11)]. *Sustainable Computing: Informatics and Systems*, *2*(2), 105–115.

Androguard. (2021). *Androguard's documentation*. Retrieved 2022-02-22, from https://androguard.readthedocs.io

Androzooopencrawler. (2022). Retrieved 2022-02-21, from https://github.com/greensoftwarelab/androzoopen-crawler

APKtool. (2022). *Apktool - a tool for reverse engineering apk files*. Retrieved 2022-04-20, from https://apktool.org

AppBrain. (2020a). *App brain android development tool statistics* [Last visit: 2023-12-10]. https://www.appbrain.com/stats/libraries/development-tools

AppBrain. (2020b). *App brain libraries statistics* [Last visit: 2021-02-10]. Retrieved 2021-02-22, from https://www.appbrain.com/stats/libraries

Apple. (2023). *Measure energy impact with instruments* [Last visit: 2023-10-10]. Retrieved 2016-09-13, from https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/MonitorEnergyWithInstruments.html

Bakker, A. R. Comparing energy profilers for android. In: In *Proceedings of 21st twente student conference on it, enschede, the netherlands*. 2014.

Baldwin, C., & Clark, K. (2000, January). *Design rules volume i: The power of modularity* (Vol. 1). https://doi.org/10.2307/259400

Banerjee, A., & Roychoudhury, A. Future of mobile software for smartphones and drones: Energy and performance. In: In *2017 ieee/acm 4th international conference on mobile software engineering and systems (mobilesoft)*. 2017-05, 1–12. https://doi.org/10.1109/MOBILESoft.2017.1

Bangash, A. A., Ali, K., & Hindle, A. A black box technique to reduce energy consumption of android apps. In: In *Proceedings of the acm/ieee 44th international conference on software engineering: New ideas and emerging results*. ICSE-NIER '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, 1–5. isbn: 9781450392242. https://doi.org/10.1145/3510455.3512795

Bangash, A. A., Ali, K., & Hindle, A. A black box technique to reduce energy consumption of android apps. In: In *New ideas and emerging results (icse-nier'22),* ICSE-NIER'22, May 21–29, 2022, Pittsburgh, PA, USA. Pittsburgh, PA, USA: ACM, 2022. https://doi.org/https://doi.org/10.1145/3510455.3512795

Bangash, A. A., Tiganov, D., Ali, K., & Hindle, A. Energy efficient guidelines for ios core location framework. In: In *2021 ieee international conference on software maintenance and evolution (icsme)*. 2021, 320–331. https://doi.org/10.1109/ICSME52107.2021.00035

Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Penzenstadler, B., Seyff, N., & Venters, C. C. Sustainability design and software: The karlskrona manifesto. In: In *2015 ieee/acm 37th ieee international conference on software engineering. 2*. 2015, 467–476. https://doi.org/10.1109/ICSE.2015.179

Biørn-Hansen, A., Rieger, C., Grønli, T.-M., Majchrzak, T. A., & Ghinea, G. (2020). An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering*. https://doi.org/10.1007/s10664-020-09827-6

Bouaffar, F., Le Goaer, O., & Noureddine, A. PowDroid: Energy Profiling of Android Applications. In: In *2nd International Workshop on Sustainable Software Engineering (SUSTAINSE)*. Melbourne, Australia, 2021-11.

Brain, A. (2023). *Android app frameworks*. Retrieved 2024-01-01, from https://www.appbrain.com/stats/libraries/tag/app-framework/android-app-frameworks

Cañete, A., Horcas, J.-M., Ayala, I., & Fuentes, L. (2020a). Energy efficient adaptation engines for android applications. *Information and Software Technology*, *118*, 106220. https://doi.org/https://doi.org/10.1016/j.infsof.2019.106220

Cañete, A., Horcas, J.-M., Ayala, I., & Fuentes, L. (2020b). Energy efficient adaptation engines for android applications. *Information and Software Technology*, *118*, 106220. https://doi.org/https://doi.org/10.1016/j.infsof.2019.106220

Chambers, J. M., Cleveland, W. S., Kleiner, B., & Tukey, P. A. (2017). *Graphical methods for data analysis*. Chapman; Hall/CRC.

Chen, X., & Zong, Z. Android app energy efficiency: The impact of language, runtime, compiler, and implementation. In: In *2016 ieee international conferences on big data and cloud computing (bdcloud), social computing and networking (socialcom), sustainable computing and communications (sustaincom) (bdcloud-socialcom-sustaincom)*. 2016-01, 485–492. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.77

Choudhary, S. R., Gorla, A., & Orso, A. Automated test input generation for android: Are we there yet? (e). In: In *Proceedings of the 2015 30th ieee/acm international conference on automated software engineering (ase)*. ASE '15. Washington, DC, USA: IEEE Computer Society, 2015, 429–440. isbn: 978-1-5090-0025-8. https://doi.org/10.1109/ASE.2015.89

Chowdhury, S., Borle, S., Romansky, S., & Hindle, A. (2019). Greenscaler: Training software energy models with automatic test generation. *Empirical Softw. Engg.*, *24*(4), 1649–1692. https://doi.org/10.1007/s10664-018-9640-7

Chowdhury, S., Di Nardo, S., Hindle, A., & Jiang, Z. M. (2018). An exploratory study on assessing the energy impact of logging on android applications. *Empirical Softw. Engg.*, *23*(3), 1422–1456. https://doi.org/10.1007/s10664-017-9545-x

Composables. (2023). *Android distribution chart*. Retrieved 2023-10-10, from https://www.composables.com/tools/distribution-chart

Corbalan, L., Fernandez, J., Cuitiño, A., Delia, L., Cáseres, G., Thomas, P., & Pesado, P. Development frameworks for mobile devices: A comparative study about energy consumption. In: In *Proceedings of the 5th international conference on mobile software engineering and systems*. MOBILESoft '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, 191–201. isbn: 9781450357128. https://doi.org/10.1145/3197231.3197242

Corbalan, L., Fernandez Sosa, J., Cuitiño, A., Delia, L., Cáseres, G., Thomas, P., & Pesado, P. Development frameworks for mobile devices: A comparative study about energy consumption. In: 2018-05, 191–201. https://doi.org/10.1145/3197231.3197242

Couto, M., Cunha, J., Fernandes, J. P., Pereira, R., & Saraiva, J. Greendroid: A tool for analysing power consumption in the android ecosystem. In: In *2015 ieee 13th international scientific conference on informatics*. 2015-11, 73–78. https://doi.org/10.1109/Informatics.2015.7377811

Couto, M., Saraiva, J., & Fernandes, J. P. Energy refactorings for android in the large and in the wild. In: In *2020 ieee 27th international conference on software analysis, evolution and reengineering (saner)*. 2020, 217–228. https://doi.org/10.1109/SANER48275.2020.9054858

Couto, M., Borba, P., Cunha, J., Fernandes, J. P., Pereira, R., & Saraiva, J. Products go green: Worst-case energy consumption in software product lines. In: In *Proceedings of the 21st international systems and software product line conference-volume a*. 2017, 84–93.

Couto, M., Carção, T., Cunha, J., Fernandes, J. P., & Saraiva, J. Detecting anomalous energy consumption in android applications (F. M. Quintão Pereira, Ed.). In: *Programming languages* (F. M. Quintão Pereira, Ed.). Ed. by Quintão Pereira, F. M. Springer International Publishing, 2014, 77–91. isbn: 978-3-319-11863-5.

Couto, M., Cunha, J., Fernandes, J. P., Pereira, R., & Saraiva, J. Greendroid: A tool for analysing power consumption in the android ecosystem. In: In *2015 ieee 13th international scientific conference on informatics*. IEEE. 2015, 73–78.

Couto, M., Pereira, R., Ribeiro, F., Rua, R., & Saraiva, J. a. Towards a green ranking for programming languages. In: In *Proceedings of the 21st brazilian symposium on programming languages*. SBLP 2017. Fortaleza, CE, Brazil: ACM, 2017, 7:1–7:8. isbn: 978-1-4503-5389-2. https://doi.org/10.1145/3125374.3125382

Cruz, L., & Abreu, R. Performance-based guidelines for energy efficient mobile applications. In: In *2017 ieee/acm 4th international conference on mobile software engineering and systems (mobilesoft)*. 2017-05, 46–57. https://doi.org/10.1109/MOBILESoft.2017.19

Cruz, L., & Abreu, R. Performance-based guidelines for energy efficient mobile applications. In: In *2017 ieee/acm 4th international conference on mobile software engineering and systems (mobilesoft)*. 2017-05, 46–57. https://doi.org/10.1109/MOBILESoft.2017.19

Cruz, L., Abreu, R., & Rouvignac, J. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In: In *2017 ieee/acm 4th international conference on mobile software engineering and systems (mobilesoft)*. 2017-05, 205–206. https://doi.org/10.1109/MOBILESoft.2017.21

Cruz, L. (2022). *Sustainable software engineering: An introduction* [Accessed: June 25, 2024]. https://luiscruz.github.io/2022/01/01/sustainable-se-intro.html

Cruz, L., & Abreu, R. (2018). Using automatic refactoring to improve energy efficiency of android apps. *CoRR*, *abs/1803.05889*. http://arxiv.org/abs/1803.05889

Cruz, L., Abreu, R., & Rouvignac, J.-N. Leafactor: Improving energy efficiency of android apps via automatic refactoring. In: In *2017 ieee/acm 4th international conference on mobile software engineering and systems (mobilesoft)*. 2017, 205–206. https://doi.org/10.1109/MOBILESoft.2017.21

Custom hunter-debug plugin. (2022). Retrieved 2022-02-04, from https://github.com/raphael28/hunter-custom-debug

D'Agostino, R. B. An omnibus test of normality for moderate and large size samples. In: 1971.

Das, T., Penta, M. D., & Malavolta, I. (2020). Characterizing the evolution of statically-detectable performance issues of android apps. *Empir. Softw. Eng.*, *25*(4), 2748–2808. https://doi.org/10.1007/s10664-019-09798-3

Distribution dashboard. (2020). Retrieved 2020-02-20, from {https://developer.android.com/about/dashboards}

Dodge, J., Prewitt, T., Tachet des Combes, R., Odmark, E., Schwartz, R., Strubell, E., Luccioni, A. S., Smith, N. A., DeCario, N., & Buchanan, W. Measuring the carbon intensity of ai in cloud instances. In: In *Proceedings of the 2022 acm conference on fairness, accountability, and transparency*. FAccT '22. Seoul, Republic of Korea: Association for Computing Machinery, 2022, 1877–1894. isbn: 9781450393522. https://doi.org/10.1145/3531146.3533234

Egele, M., Brumley, D., Fratantonio, Y., & Kruegel, C. An empirical study of cryptographic misuse in android applications. In: In *Proceedings of the 2013 acm sigsac conference on computer amp; communications security*. CCS '13. Berlin, Germany: Association for Computing Machinery, 2013, 73–84. isbn: 9781450324779. https://doi.org/10.1145/2508859.2516693

Escobar-Velásquez, C., Mazuera-Rozo, A., Bedoya, C., Osorio-Riaño, M., Linares-Vásquez, M., & Bavota, G. (2022). Studying eventual connectivity issues in android apps. *Empirical Software Engineering*, *27*. https://doi.org/10.1007/s10664-021-10020-6

F-Droid - Free and Open Source Android App Repository [Accessed: June 25, 2024]. (n.d.). https://f-droid.org/

Field, A. (2013). *Discovering statistics using ibm spss statistics* (4th). Sage Publications Ltd.

Goaër, O. L. Enforcing green code with android lint. In: In *2020 35th ieee/acm international conference on automated software engineering workshops (asew)*. 2020, 85–90. https://doi.org/10.1145/3417113.3422188

Goaër, O. L. Enforcing green code with android lint. In: In *2020 35th ieee/acm international conference on automated software engineering workshops (asew)*. 2020, 85–90. https://doi.org/10.1145/3417113.3422188

Gomez, L., Neamtiu, I., Azim, T., & Millstein, T. Reran: Timing- and touch-sensitive record and replay for android. In: In *2013 35th international conference on software engineering (icse)*. 2013, 72–81. https://doi.org/10.1109/ICSE.2013.6606553

Gonçalves, N., Rua, R., Cunha, J., Pereira, R., & de Sousa Saraiva, J. (2022). Energy efficiency of web browsers in the android ecosystem. *ArXiv*, *abs/2205.11399*.

Google. (2021a). *Android lint checks* [Last visit: 2021-02-10]. Retrieved 2021-02-22, from http://tools.android.com/tips/lint-checks

Google. (2021b). *Android profiler* [Last visit: 2021-02-10]. Retrieved 2021-02-22, from https://developer.android.com/studio/profile/android-profiler

GOOGLE. (2021). *Android rust introduction*. https://source.android.com/setup/build/rust/building-rust-modules/overview

Google. (2021). *App crawler* [Last visit: 2021-02-10]. Retrieved 2021-02-22, from https://developer.android.com/training/testing/crawler?hl=en

Google. (2021). *Google sustainability* [Accessed on 2023-10-10]. https://sustainability.google/

Google. (2021a). *Profile battery usage with batterystats*. https://developer.android.com/topic/performance/power/setup-battery-historian

Google. (2021b). *Ui/application exerciser monkey* [Last visit: 2021-02-10]. Retrieved 2021-02-22, from https://developer.android.com/studio/test/monkey

Google. (2022a). *Android anr*. Retrieved 2022-03-27, from https://developer.android.com/topic/performance/vitals/anr

Google. (2022b). *Espresso*. https://developer.android.com/training/testing/espresso

Google. (2022c). *Measuring power values*. Retrieved 2021-12-27, from https://source.android.com/docs/core/power/values

Google. (2022d). *Perfetto: System profiling, app tracing and trace analysis*. Retrieved 2021-12-27, from https://perfetto.dev

Google. (2022e). *Write automated tests with ui automator*. https://developer.android.com/training/testing/other-components/ui-automator

Gradle. (2023). *Gradle*. https://gradle.org

Hao, S., Li, D., Halfond, W. G. J., & Govindan, R. Estimating mobile application energy consumption using program analysis. In: In *Proc. of 35th int. conf. on software engineering (icse)*. 2013-05.

Hao, S., Li, D., Halfond, W. G. J., & Govindan, R. Estimating mobile application energy consumption using program analysis. In: In *2013 35th international conference on software engineering (icse)*. 2013-05, 92–101. https://doi.org/10.1109/ICSE.2013.6606555

Hao, S., Li, D., Halfond, W., & Govindan, R. Estimating Android applications' CPU energy usage via bytecode profiling. In: In *Green and sustainable software (greens), 2012 first int. workshop on*. 2012-06, 1–7.

HARDKERNEL CO., L. (2022). *Odroid*. Retrieved 2022-02-22, from https://www.hardkernel.com

Hegde, N., Melanson, E. L., & Sazonov, E. Development of a realtime activity monitoring android application utilizing smartstep. In: In *Proceedings of the 38th ieee int. conf. of the engineering in medicine and biology society*. 2016, 1886–1889.

Hindle, A. Green mining: A methodology of relating software change to power consumption. In: In *2012 9th ieee working conference on mining software repositories (msr)*. 2012-06, 78–87. https://doi.org/10.1109/MSR.2012.6224303

Hindle, A. (2013). Green mining: A methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, *20*. https://doi.org/10.1007/s10664-013-9276-6

Hoque, M., Siekkinen, M., Khan, K., Xiao, Y., & Tarkoma, S. (2015). Modeling, profiling, and debugging the energy consumption of mobile devices. *ACM Computing Surveys*, *48*, 40. https://doi.org/10.1145/2840723

Hort, M., Kechagia, M., Sarro, F., & Harman, M. (2022). A survey of performance optimization for mobile applications. *IEEE Transactions on Software Engineering*, *48*(8), 2879–2904. https://doi.org/10.1109/TSE.2021.3071193

Hu, Y., Yan, J., Yan, D., Lu, Q., & Yan, J. (2017). Lightweight energy consumption analysis and prediction for android applications. *Science of Computer Programming*, *162*. https://doi.org/10.1016/j.scico.2017.05.002

Jabbarvand, R., & Malek, S. □droid: An energy-aware mutation testing framework for android. In: In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 208–219. isbn: 9781450351058. https://doi.org/10.1145/3106237.3106244

Jung, W., Kang, C., Yoon, C., Kim, D., & Cha, H. Devscope: A nonintrusive and online power analysis tool for smartphone hardware components. In: 2012-10. https://doi.org/10.1145/2380445.2380502

JUnit. (2022). *Junit*. Retrieved 2022-02-22, from https://junit.org/junit5/

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. Aspect-oriented programming (M. Akşit & S. Matsuoka, Eds.). In: *Ecoop'97 — object-oriented programming* (M. Akşit & S. Matsuoka, Eds.). Ed. by Akşit, M., & Matsuoka, S. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, 220–242.

Kin Keong, C., Tieng Wei, K., Abd. Ghani, A. A., & Sharif, K. Y. Toward using software metrics as indicator to measure power consumption of mobile application: A case study. In: In *2015 9th malaysian software engineering conference (mysec)*. 2015, 172–177. https://doi.org/10.1109/MySEC.2015.7475216

Kjærgaard, M., & Blunck, H. Unsupervised power profiling for mobile devices. In: *104*. 2012-01. isbn: 978-3-642-30972-4. https://doi.org/10.1007/978-3-642-30973-1_12

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2019a). Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, *68*(1), 45–66. https://doi.org/10.1109/TR.2018.2865733

Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., & Klein, J. (2019b). Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, *68*(1), 45–66. https://doi.org/10.1109/TR.2018.2865733

Kruskal, W. H., & Wallis, W. A. (1952). Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association*, *47*(260), 583–621. https://doi.org/10.1080/01621459.1952.10483441

Lachgar, M., Benouda, H., & Elfirdoussi, S. Android rest apis: Volley vs retrofit. In: In *2018 international symposium on advanced electrical and communication technologies (isaect)*. 2018, 1–6. https://doi.org/10.1109/ISAECT.2018.8618824

Levin, J. (2015). *Android internals: A confectioner's cookbook - volume i* (1st). Technologeeks.com.

Li, D., Hao, S., Halfond, W. G. J., & Govindan, R. Calculating Source Line Level Energy Information for Android Applications. In: In *Proc. of 2013 int. symposium on software testing and analysis*. ISSTA 2013. ACM, 2013, 78–89.

Li, D., & Halfond, W. G. J. An investigation into energy-saving programming practices for android smartphone app development. In: In *Proceedings of the 3rd international workshop on green and sustainable software*. GREENS 2014. Hyderabad, India: ACM, 2014, 46–53. isbn: 978-1-4503-2844-9. https://doi.org/10.1145/2593743.2593750

Li, D., Jin, Y., Sahin, C., Clause, J., & Halfond, W. G. J. Integrated energy-directed test suite optimization. In: In *Proceedings of the 2014 international symposium on software testing and analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, 339–350. isbn: 9781450326452. https://doi.org/10.1145/2610384.2610414

Li, D., Lyu, Y., Gui, J., & Halfond, W. G. J. Automated energy optimization of http requests for mobile applications. In: In *2016 ieee/acm 38th international conference on software engineering (icse)*. 2016, 249–260. https://doi.org/10.1145/2884781.2884867

Li, X., Chen, J., Liu, Y., Wu, K., & Gallagher, J. (2022). Combatting energy issues for mobile applications. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3527851

Li, X., Chen, J., Liu, Y., Wu, K., & Gallagher, J. P. (2023). Combatting energy issues for mobile applications. *ACM Trans. Softw. Eng. Methodol.*, *32*(1). https://doi.org/10.1145/3527851

Li, X., Yang, Y., Liu, Y., Gallagher, J. P., & Wu, K. Detecting and diagnosing energy issues for mobile applications. In: In *Proceedings of the 29th acm sigsoft international symposium on software testing and analysis*. ISSTA 2020. Virtual Event, USA: Association for Computing Machinery, 2020, 115–127. isbn: 9781450380089. https://doi.org/10.1145/3395363.3397350

Li, Y., Yang, Z., Guo, Y., & Chen, X. Droidbot: A lightweight ui-guided test input generator for android. In: In *2017 ieee/acm 39th international conference on software engineering companion (icse-c)*. 2017, 23–26.

Lima, L. G., Soares-Neto, F., Lieuthier, P., Castor, F., Melfe, G., & Fernandes, J. P. Haskell in green land: Analyzing the energy behavior of a purely functional language. In: In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)*. *1*. IEEE. 2016, 517–528.

Lin, J. W., Salehnamadi, N., & Malek, S. Test automation in open-source android apps: A large-scale empirical study. In: In *2020 35th ieee/acm international conference on automated software engineering (ase)*. 2020, 1078–1089.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., & Poshyvanyk, D. Mining energy-greedy api usage patterns in android apps: An empirical study. In: In *Proceedings of the*

11th working conference on mining software repositories. MSR 2014. Hyderabad, India: ACM, 2014, 2–11. isbn: 978-1-4503-2863-0. https://doi.org/10.1145/2597073.2597085

Linkedin. (2018). *Dexmaker*. Retrieved 2022-02-22, from https://github.com/linkedin/dexmaker

Liu, J., Wu, T., Deng, X., Yan, J., & Zhang, J. Insdal: A safe and extensible instrumentation tool on dalvik byte-code for android applications. In: In *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)*. 2017, 502–506. https://doi.org/10.1109/SANER.2017.7884662

Liu, P., Li, L., Zhao, Y., Sun, X., & Grundy, J. Androzoooopen: Collecting large-scale open source android apps for the research community. In: MSR '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 548–552. isbn: 9781450375177. https://doi.org/10.1145/3379597.3387503

Lourenço, J. M. (2021). *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. https://github.com/joaomlourenco/novathesis/raw/main/template.pdf

Ma, X., Huang, P., Jin, X., Wang, P., Park, S., Shen, D., Zhou, Y., Saul, L. K., & Voelker, G. M. eDoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In: In *10th usenix symposium on networked systems design and implementation (nsdi 13)*. Lombard, IL: USENIX Association, 2013-04, 57–70. isbn: 978-1-931971-00-3. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ma

Machiry, A., Tahiliani, R., & Naik, M. Dynodroid: An input generation system for android apps. In: In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*. ESEC/FSE 2013. Saint Petersburg, Russia: Association for Computing Machinery, 2013, 224–234. isbn: 9781450322379. https://doi.org/10.1145/2491411.2491450

Maia, D., Couto, M., Saraiva, J., & Pereira, R. E-debitum: Managing software energy debt. In: In *2020 35th ieee/acm international conference on automated software engineering workshops (asew)*. 2020, 170–177. https://doi.org/10.1145/3417113.3422999

Malavolta, I., Grua, E. M., Lam, C.-Y., de Vries, R., Tan, F., Zielinski, E., Peters, M., & Kaandorp, L. A framework for the automatic execution of measurement-based experiments on android devices. In: In *2020 35th ieee/acm international conference on automated software engineering workshops (asew)*. 2020, 61–66. https://doi.org/10.1145/3417113.3422184

Mann, H. B., & Whitney, D. R. (1947). On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, *18*(1), 50–60. https://doi.org/10.1214/aoms/1177730491

Manotas, I., Bird, C., Zhang, R., Shepherd, D., Jaspan, C., Sadowski, C., Pollock, L., & Clause, J. An empirical study of practitioners' perspectives on green software engineering. In: In *International conference on software engineering (icse), 2016 ieee/acm 38th*. IEEE. 2016, 237–248.

Martins, P., Achar, R., & Lopes, C. V. 50k-c: A dataset of compilable, and compiled, java projects. In: In *Proceedings of the 15th international conference on mining software repositories*. MSR '18. Gothenburg, Sweden: ACM, 2018, 1–5. isbn: 978-1-4503-5716-6. https://doi.org/10.1145/3196398.3196450

Matalonga, H., Cabral, B., Castor, F., Couto, M., Pereira, R., de Sousa, S. a. M., & Fernandes, J. a. P. Greenhub farmer: Real-world data for android energy mining. In: In *Proceedings of the 16th international conference on mining software repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, 171–175. https://doi.org/10.1109/MSR.2019.00034

Mazuera-Rozo, A., Trubiani, C., Linares-Vásquez, M., & Bavota, G. (2020). Investigating types and survivability of performance bugs in mobile apps. *Empirical Softw. Engg.*, *25*(3), 1644–1686. https://doi.org/10.1007/s10664-019-09795-6

McCabe, T. J. A complexity measure. In: In *Proceedings of the 2nd international conference on software engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, 407–. http://dl.acm.org/citation.cfm?id=800253.807712

Mickle, T. (2018). Your phone is almost out of battery. remain calm. call a doctor. [Last visit: 2021-02-10].

Monkeyrunner. (2020). Retrieved 2019-12-27, from {https://developer.android.com/studio/test/monkeyrunner}

Monsoon. (2021). *Monsoon power monitor* [Last visit: 2021-02-10]. Retrieved 2021-02-22, from https://www.msoon.com

Montenegro, J. A., Pinto, M., & Fuentes, L. (2018). What do software developers need to know to build secure energy-efficient android applications? *IEEE Access*, *6*, 1428–1450. https://doi.org/10.1109/ACCESS.2017.2779131

Morales, R., Saborido, R., Khomh, F., Chicano, F., & Antoniol, G. (2018). Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Transactions on Software Engineering*, *44*(12), 1176–1206. https://doi.org/10.1109/TSE.2017.2757486

Nucci, D. D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., & Lucia, A. D. Petra: A software-based tool for estimating the energy profile of android applications. In: In *2017 ieee/acm 39th international conference on software engineering companion (icse-c)*. 2017-05, 3–6. https://doi.org/10.1109/ICSE-C.2017.18

Nucci, D. D., Palomba, F., Prota, A., Panichella, A., Zaidman, A., & Lucia, A. D. Software-based energy profiling of android apps: Simple, efficient and reliable? In: In *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)*. 2017-02, 103–114. https://doi.org/10.1109/SANER.2017.7884613

Oliveira, W., Matalonga, H., Pinto, G., Castor, F., & Fernandes, J. P. (2020). Small changes, big impacts: Leveraging diversity to improve energy efficiency.

Oliveira, W., Moraes, B., Castor, F., & Fernandes, J. a. P. Analyzing the resource usage overhead of mobile app development frameworks. In: In *Proceedings of the 27th international conference on evaluation and assessment in software engineering*. EASE '23. Oulu, Finland: Association for Computing Machinery, 2023, 152–161. isbn: 9798400700446. https://doi.org/10.1145/3593434.3593487

Oliveira, W., Oliveira, R., & Castor, F. A study on the energy consumption of android app development approaches. In: In *2017 ieee/acm 14th international conference on mining software repositories (msr)*. 2017, 42–52. https://doi.org/10.1109/MSR.2017.66

Oliveira, W., Oliveira, R., & Castor, F. A study on the energy consumption of android app development approaches. In: In *2017 ieee/acm 14th international conference on mining software repositories (msr)*. 2017, 42–52. https://doi.org/10.1109/MSR.2017.66

Oliveira, W., Oliveira, R., Castor, F., Fernandes, B., & Pinto, G. Recommending energy-efficient java collections. In: In *Proceedings of the 16th international conference on mining software repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, 160–170. https://doi.org/10.1109/MSR.2019.00033

Ortiz, G., García-De-Prado, A., Berrocal, J., & Hernández, J. (2019). Improving resource consumption in context- aware mobile applications through alternative architectural styles. *IEEE Access*, *7*, 65228–65250. https://doi.org/10.1109/ACCESS.2019.2918239

Overflow, S. (2021). *Stack overflow developer survey*. Retrieved 2022-02-22, from https://insights.stackoverflow.com/survey/2021#section-most-popular-technologies-programming-scripting-and-markup-languages

Pang, C., Hindle, A., Adams, B., & Hassan, A. E. (2016). What do programmers know about software energy consumption? *IEEE Software*, *33*(3), 83–89.

Park, J., Park, Y. B., & Ham, H. K. Fragmentation problem in android. In: In *2013 international conference on information science and applications (icisa)*. 2013, 1–2. https://doi.org/10.1109/ICISA.2013.6579465

Pathak, A., Hu, Y. C., & Zhang, M. Where is the energy spent inside my app?: Fine grained energy accounting on smartphones with eprof. In: In *European conference on computer systems*. 2012. https://api.semanticscholar.org/CorpusID:15343625

Pathak, A., Hu, Y. C., Zhang, M., Bahl, P., & Wang, Y.-M. Fine-grained power modeling for smartphones using system call tracing. In: In *Proceedings of the sixth conference on computer systems*. EuroSys '11. Salzburg, Austria: ACM, 2011, 153–168. isbn: 978-1-4503-0634-8. https://doi.org/10.1145/1966445.1966460

Pereira, R., Carção, T., Couto, M., Cunha, J., Fernandes, J. P., & Saraiva, J. (2020). Spelling out energy leaks: Aiding developers locate energy inefficient code. *Journal of Systems and Software*, *161*, 110463.

Pereira, R., Couto, M., Cunha, J., Fernandes, J. P., & Saraiva, J. The influence of the java collection framework on overall energy consumption. In: In *2016 ieee/acm 5th international workshop on green and sustainable software (greens)*. IEEE. 2016, 15–21.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. a. P., & Saraiva, J. a. Energy efficiency across programming languages: How do energy, time, and memory relate? In: In *Proceedings of the 10th acm sigplan international conference on software language engineering*. SLE 2017. Vancouver, BC, Canada: ACM, 2017, 256–267. isbn: 978-1-4503-5525-4. https://doi.org/10.1145/3136014.3136031

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2021a). Ranking programming languages by energy efficiency. *Science of Computer Programming*, *205*, 102609. https://doi.org/https://doi.org/10.1016/j.scico.2021.102609

Pereira, R., Matalonga, H., Couto, M., Castor, F., Cabral, B., Carvalho, P., de Sousa, S. a. M., & Fernandes, J. a. P. (2021b). Greenhub: A large-scale collaborative dataset to battery consumption analysis of android devices. *Empirical Softw. Engg.*, *26*(3). https://doi.org/10.1007/s10664-020-09925-5

Pereira, R., Simão, P., Cunha, J., & Saraiva, J. a. Jstanley: Placing a green thumb on java collections. In: In *Proceedings of the 33rd acm/ieee international conference on automated software engineering*.

ASE 2018. Montpellier, France: ACM, 2018, 856–859. isbn: 978-1-4503-5937-5. https://doi.org/10.1145/3238147.3240473

Peters, M., Scoccia, G. L., & Malavolta, I. How does migrating to kotlin impact the run-time efficiency of android apps? In: In *2021 ieee 21st international working conference on source code analysis and manipulation (scam)*. 2021, 36–46. https://doi.org/10.1109/SCAM52516.2021.00014

Pinto, G., & Castor, F. (2017). Energy efficiency: A new concern for application software developers. *Commun. ACM*, *60*(12), 68–75. https://doi.org/10.1145/3154384

Pinto, G., Castor, F., & Liu, Y. D. Mining questions about software energy consumption. In: In *Proceedings of the 11th working conference on mining software repositories*. MSR 2014. Hyderabad, India: ACM, 2014, 22–31. isbn: 978-1-4503-2863-0. https://doi.org/10.1145/2597073.2597110

Pinto, G., Castor, F., & Liu, Y. D. Mining questions about software energy consumption. In: In *Proceedings of the 11th working conference on mining software repositories*. ACM. 2014, 22–31.

Police, A. (2021). *Google's latest android version distribution numbers show 11 in dead heat with 10*. Retrieved 2022-03-27, from https://www.androidpolice.com/googles-latest-android-version-distribution-numbers-show-11-in-dead-heat-with-10/

Qualcomm. (2014). *Qualcomm forum - which mobile devices report accurate system power consumption?* [Last visit: 2021-02-10]. Retrieved 2021-02-22, from https://developer.qualcomm.com/forum/qdn-forums/software/trepn-power-profiler/28349

Qualcomm Innovation Center, I. (2016). *Trepn profiler android app* [Last visit: 2023-02-10]. Retrieved 2023-02-22, from https://play.google.com/store/apps/details?id=com.quicinc.trepn

Quinn. (2018). *Hunter-debug plugin* [Last visit: 2023-02-04].

Ramesh, G., & Umarani, R. (2012). Performance analysis of most common encryption algorithms on different web browsers. *International Journal of Information Technology and Computer Science*, *4*, 60–66. https://doi.org/10.5815/ijitcs.2012.12.06

Rea, L., Ñames, Parker, R. A., & Allen, R. Designing and conducting survey research. In: Jossey-Bass Publishers, 2016.

Ribeiro, A., Ferreira, J. F., & Mendes, A. Ecoandroid: An android studio plugin for developing energy-efficient java mobile applications. In: In *2021 ieee 21st international conference on software quality, reliability and security (qrs)*. 2021, 62–69. https://doi.org/10.1109/QRS54544.2021.00017

Richter, F. (2021). The most wanted smartphone features [Accessed: 2021-02-10].

RobotiumTech. (2022). *Robotium: User scenario testing for android*. Retrieved 2021-12-27, from https://github.com/RobotiumTech/robotium

Rosenthal, R., Cooper, H, & Hedges, L. (1994). Parametric measures of effect size. *The handbook of research synthesis*, 231–244.

Rowling, J. (2000). *Harry potter and the goblet of fire* (1st ed.). Scholastic.

Rua, R., Couto, M., & Saraiva, J. Greensource: A large-scale collection of android code, tests and energy metrics. In: In *2019 ieee/acm 16th international conference on mining software repositories (msr)*. 2019-05, 176–180. https://doi.org/10.1109/MSR.2019.00035

Rua, R. (2022). *Online appendix* [Last visit: 2022-11-22]. Retrieved 2022-11-22, from https://sites.google.com/view/perf-guidelining-appendix

Rua, R., Couto, M., Pinto, A., Cunha, J., & Saraiva, J. Towards using memoization for saving energy in android (B. Marín, I. S. Brito, M. K. Mora, A. Malucelli, E. Serral, G. Giachetti, J. Araújo, M. Goulão, C. P. Ayala, M. Genero, & V. S. Souza, Eds.). In: *Proceedings of the XXII iberoamerican conference on software engineering, cibse 2019, la habana, cuba, april 22-26, 2019* (B. Marín, I. S. Brito, M. K. Mora, A. Malucelli, E. Serral, G. Giachetti, J. Araújo, M. Goulão, C. P. Ayala, M. Genero, & V. S. Souza, Eds.). Ed. by Marín, B., Brito, I. S., Mora, M. K., Malucelli, A., Serral, E., Giachetti, G., Araújo, J., Goulão, M., Ayala, C. P., Genero, M., & Souza, V. S. Curran Associates, 2019, 279–292. isbn: 978-1-5108-8795-4.

Rua, R., Fraga, T., Couto, M., & Saraiva, J. a. Greenspecting android virtual keyboards. In: In *Proceedings of the ieee/acm 7th international conference on mobile software engineering and systems*. MOBILESoft '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, 98–108. isbn: 9781450379595. https://doi.org/10.1145/3387905.3388600

Rua, R., & Saraiva, J. a. E-manafa: Energy monitoring and analysis tool for android. In: ASE22. Rochester, MI, USA: Association for Computing Machinery, 2022. isbn: 9781450394758. https://doi.org/10.1145/3551349.3561342

Rua, R., & Saraiva, J. (2023a). A large-scale empirical study on mobile performance: Energy, run-time and memory. *Empirical Software Engineering*, 67. https://doi.org/10.1007/s10664-023-10391-y

Rua, R., & Saraiva, J. Pyanadroid: A fully-customizable execution pipeline for benchmarking android applications. In: In *2023 ieee international conference on software maintenance and evolution (icsme)*. 2023, 586–591. https://doi.org/10.1109/ICSME58846.2023.00077

Sadeghi, A., Jabbarvand, R., & Malek, S. Patdroid: Permission-aware gui testing of android. In: In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, 220–232. isbn: 9781450351058. https://doi.org/10.1145/3106237.3106250

Sahin, C., Tornquist, P., Mckenna, R., Pearson, Z., & Clause, J. How does code obfuscation impact energy usage? In: In *2014 ieee international conference on software maintenance and evolution*. 2014-09, 131–140. https://doi.org/10.1109/ICSME.2014.35

Salama, D.-D., Abd elkader, H., & Hadhoud, M. M. (2009). Energy efficiency of encryption schemes for wireless devices. *International Journal of Computer Theory and Engineering*, 302–309. https://doi.org/10.7763/IJCTE.2009.V1.49

Salama, D.-D., Abd elkader, H., & Hadhoud, M. M. (2010). Evaluating the effects of symmetric cryptography algorithms on power consumption for different data types. *International Journal of Network Security*, *11*.

Scalabrino, S., Bavota, G., Linares-Vásquez, M., Piantadosi, V., Lanza, M., & Oliveto, R. (2020). Api compatibility issues in android: Causes and effectiveness of data-driven detection techniques. *Empirical Software Engineering*, *25*, 5006–5046. https://doi.org/10.1007/s10664-020-09877-w

Schlachter, F. (2013). No moore's law for batteries. *Proceedings of the National Academy of Sciences*, *110*(14), 5273–5273. https://doi.org/10.1073/pnas.1302988110

Schuler, A., & Anderst-Kotsis, G. Towards a framework for detecting energy drain in mobile applications: An architecture overview. In: In *Companion proceedings for the issta/ecoop 2018 workshops*. ISSTA '18. Amsterdam, Netherlands: ACM, 2018, 144–149. isbn: 978-1-4503-5939-9. https://doi.org/10.1145/3236454.3236504

Shapiro, S. S., & Wilk, M. B. (1965). An analysis of variance test for normality (complete samples)†. *Biometrika*, *52*(3-4), 591–611. https://doi.org/10.1093/biomet/52.3-4.591

Sloc cloc and code (scc). (2022). https://github.com/boyter/scc#sloc-cloc-and-code-scc

Spearman, C. (1904). The proof and measurement of association between two things. *The American Journal of Psychology*, *15*(1), 72–101. http://www.jstor.org/stable/1412159

The Shift Project. (2019). *Lean ICT: Our new report* [Accessed on 2023-10-10]. https://theshiftproject.org/en/article/lean-ict-our-new-report/

Theis, T. N., & Wong, H. P. (2017). The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, *19*(2), 41–50. https://doi.org/10.1109/MCSE.2017.29

Thorwart, A., & O'Neill, D. (2017). Camera and Battery Features Continue to Drive Consumer Satisfaction of Smartphones in US [Last visit: 2021-02-10].

United Nations, D. o. E., & Development, S. A. S. (2015). Transforming our world: The 2030 agenda for sustainable development. https://sdgs.un.org/2030agenda

van Hezewijk, R. (1965). Quasi-experimentation. *The School Review*.

Vilkomir, S., & Amstutz, B. Using combinatorial approaches for testing mobile applications. In: In *2014 ieee seventh international conference on software testing, verification and validation workshops*. 2014, 78–83. https://doi.org/10.1109/ICSTW.2014.9

Welch, B. L. (1947). The generalization of 'student's' problem when several different population variances are involved. *Biometrika*, *34*(1-2), 28–35. https://doi.org/10.1093/biomet/34.1-2.28

Westfield, B., & Gopalan, A. Orka: A new technique to profile the energy usage of android applications. In: In *2016 5th international conference on smart cities and green ict systems (smartgreens)*. 2016, 1–12.

Wilke, C., Piechnick, C., Richly, S., Püschel, G., Götz, S., & Aundefinedmann, U. Comparing mobile applications' energy consumption. In: In *Proceedings of the 28th annual acm symposium on applied computing*. SAC '13. Coimbra, Portugal: Association for Computing Machinery, 2013, 1177–1179. isbn: 9781450316569. https://doi.org/10.1145/2480362.2480583

YAMANE, T. (1973). *Statistics. an introductory analysis. third edition*. Harper & Row. https://books.google.co.jp/books?id=sl75MgEACAAJ

Yang, J., Hotta, K., Higo, Y., & Kusumoto, S. Towards purity-guided refactoring in Java. In: In *Int. conf. on software maintenance and evolution*. 2015, 521–525.

Yang, J., Hotta, K., Higo, Y., & Kusumoto, S. Revealing purity and side effects on functions for reusing Java libraries. In: ICSR 2014. Cham, 2014, 314–329.

Yoon, C., Kim, D., Jung, W., Kang, C., & Cha, H. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. In: 2012-06.

Zeng, Y., Chen, J., Shang, W., & Chen, T.-H. P. (2019). Studying the characteristics of logging practices in mobile apps: A case study on f-droid. *Empirical Software Engineering*, *24*. https://doi.org/10.1007/s10664-019-09687-9

Zhang, L., Tiwana, B., Dick, R. P., Qian, Z., Mao, Z. M., Wang, Z., & Yang, L. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In:

In *2010 ieee/acm/ifip international conference on hardware/software codesign and system synthesis (codes+isss)*. 2010, 105–114.

A

Chapter 4 Appendix

A.1 Libraries

A.1.1 Collections/Data Structures Libraries

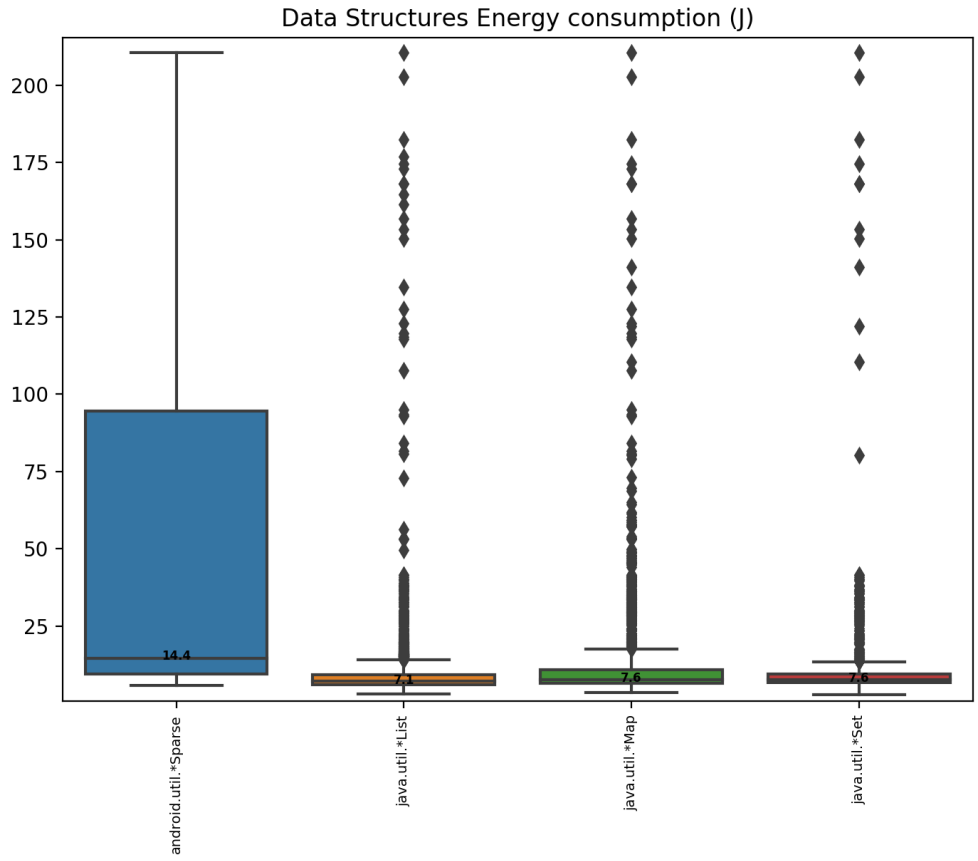


Figure 50: Collections/Data Structures Libraries Energy Consumption

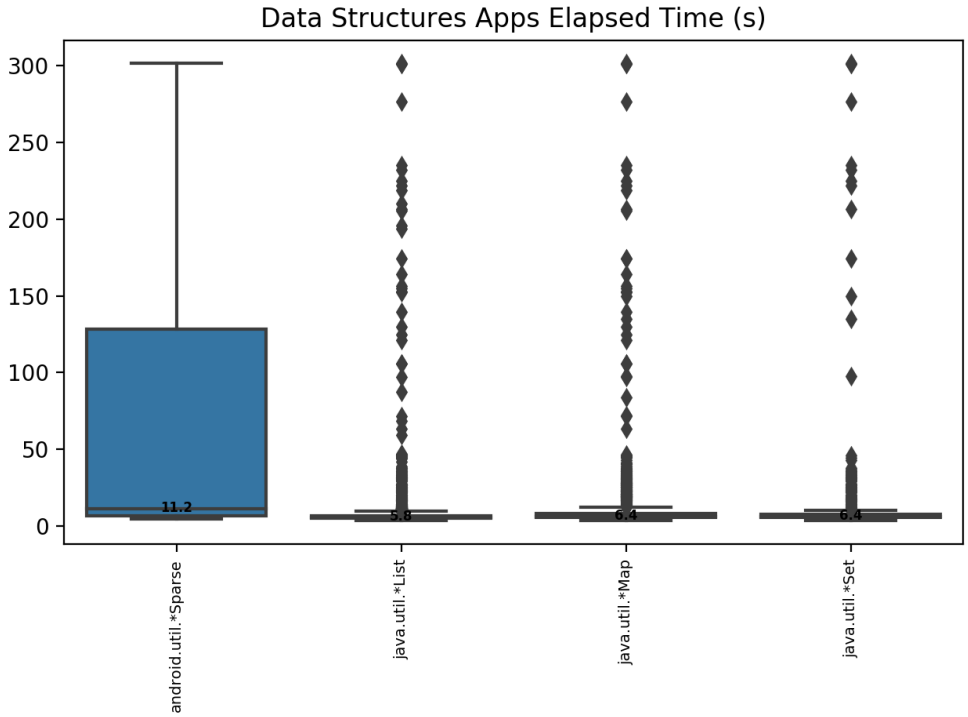


Figure 51: Collections/Data Structures Libraries Elapsed Time

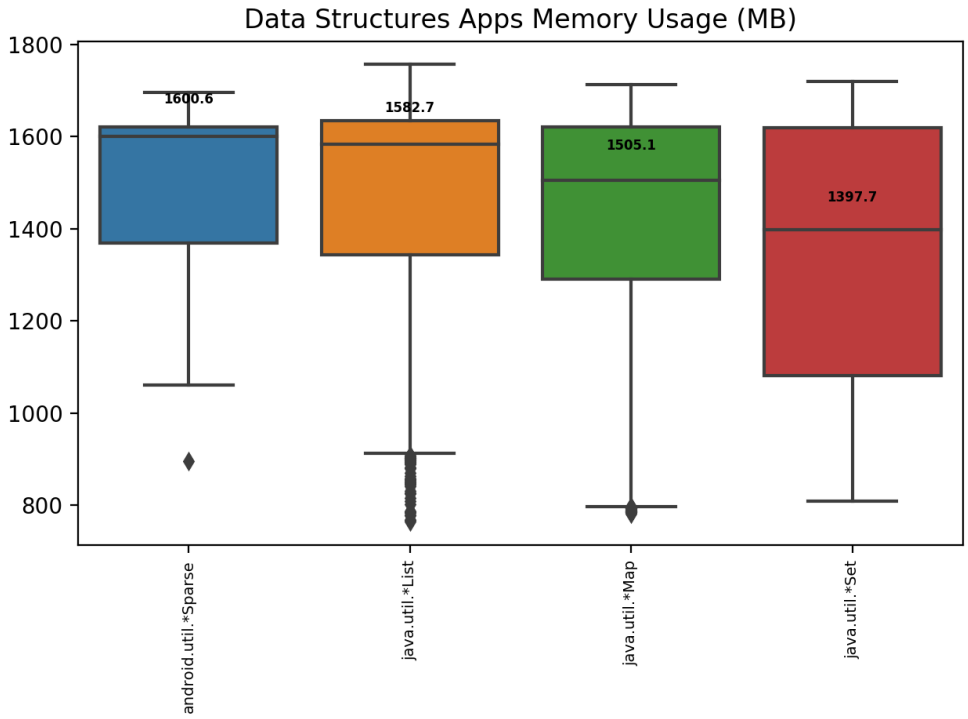


Figure 52: Collections/Data Structures Libraries Memory Usage

A.1.2 Logging Libraries

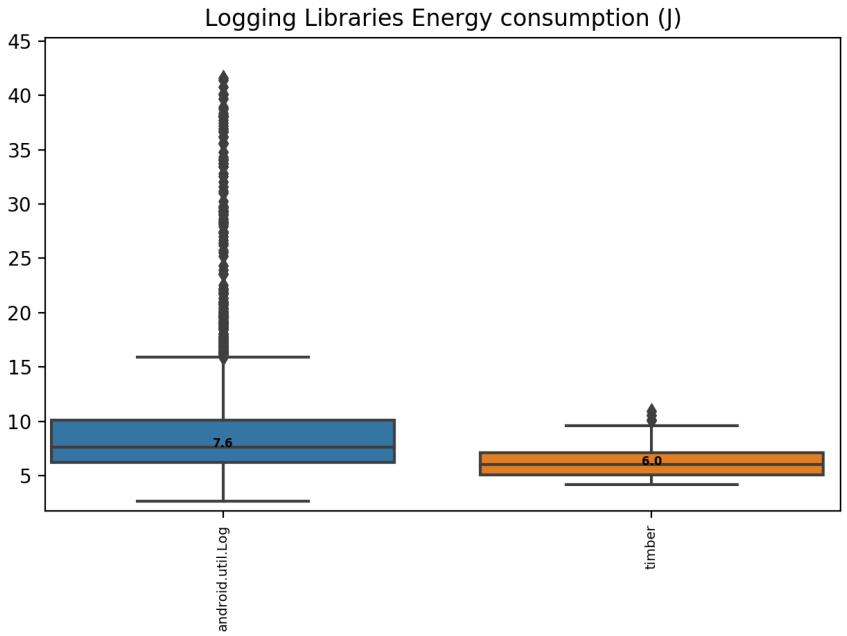


Figure 53: Logging Libraries Energy Consumption

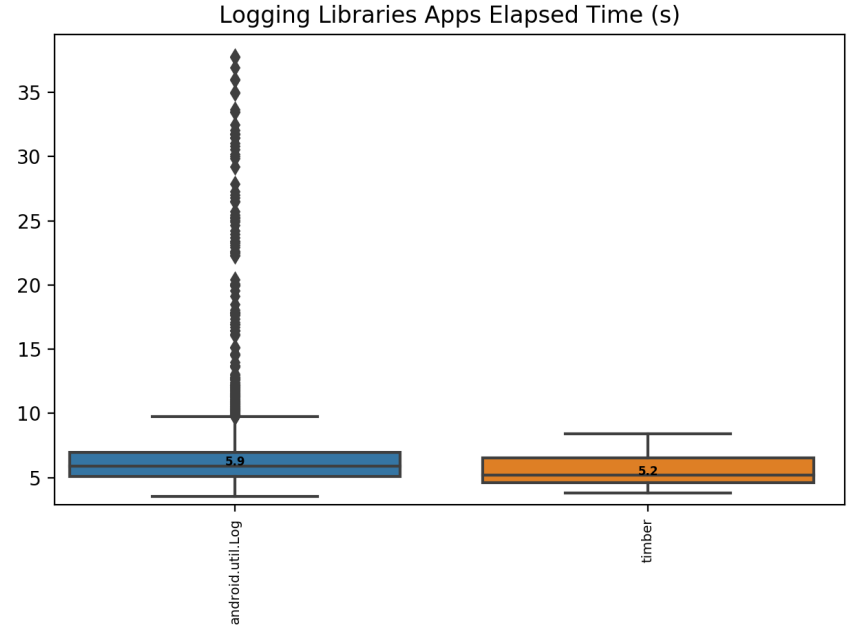


Figure 54: Logging Libraries Elapsed Time

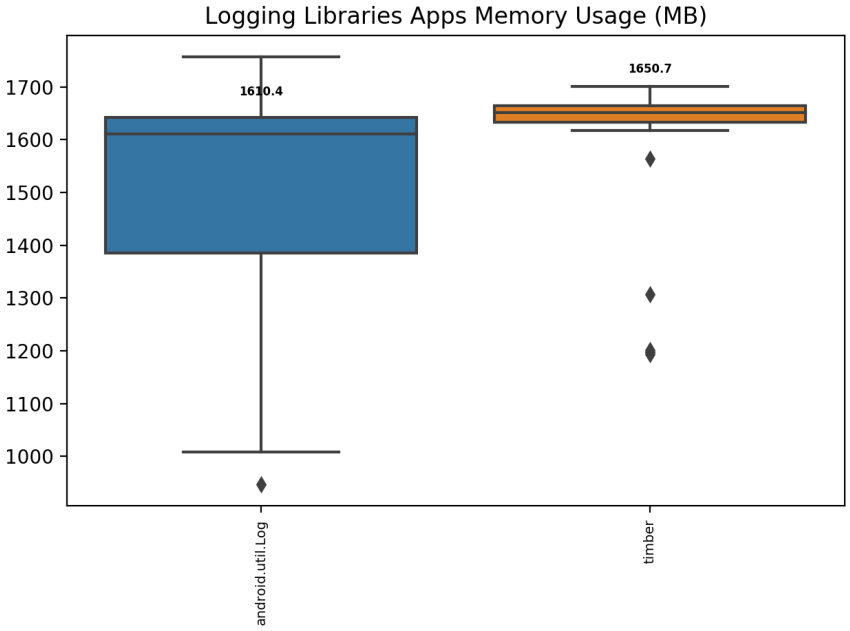


Figure 55: Logging Libraries Memory Usage

A.1.3 Image Loading/Rendering Libraries

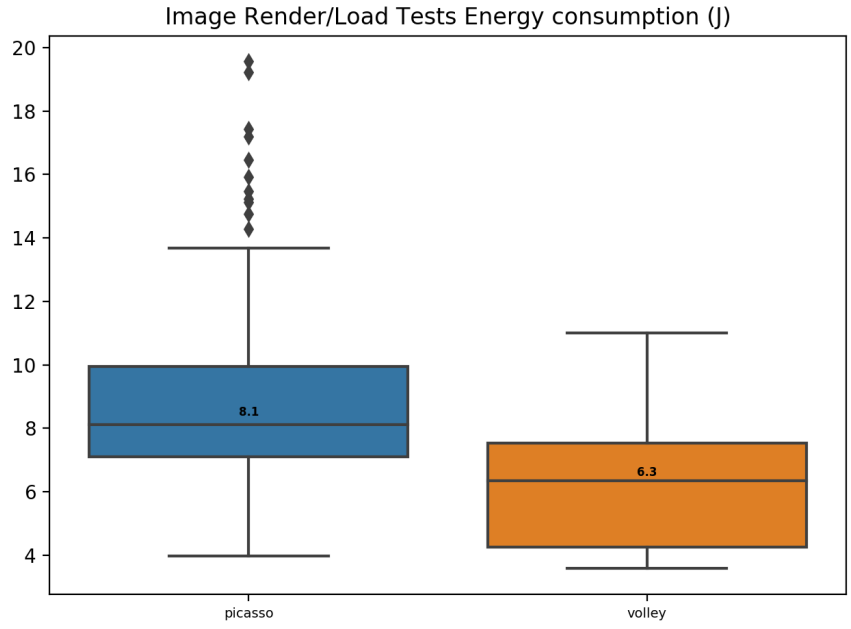


Figure 56: Image Loading/Rendering Energy Consumption

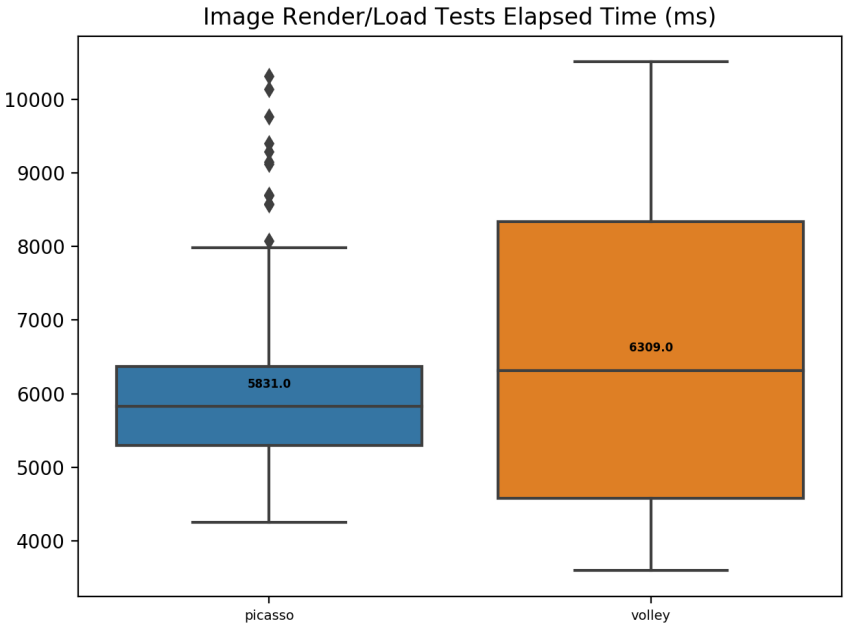


Figure 57: Image Loading/Rendering Libraries Elapsed Time

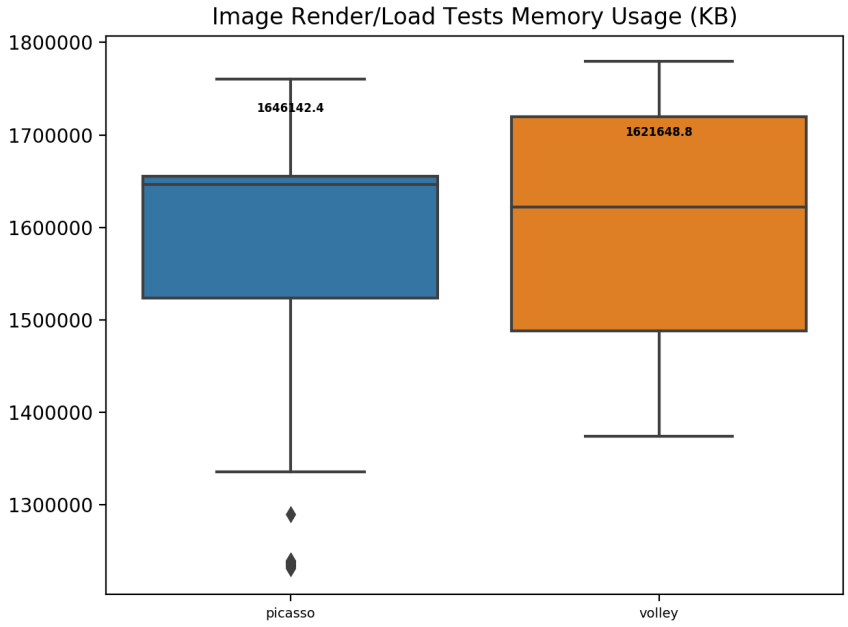


Figure 58: Image Loading/Rendering Libraries Memory Usage

A.1.4 I/O Libraries

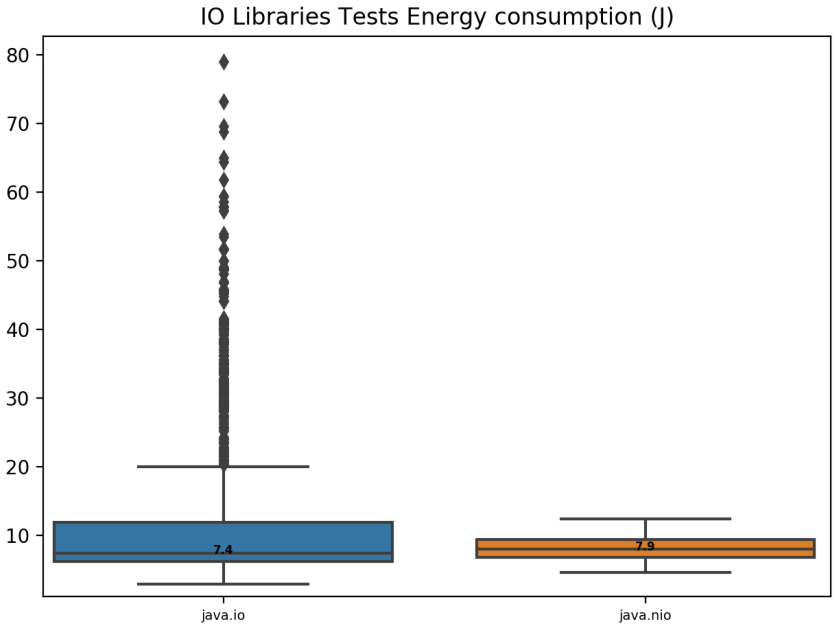


Figure 59: I/O Libraries Energy Consumption

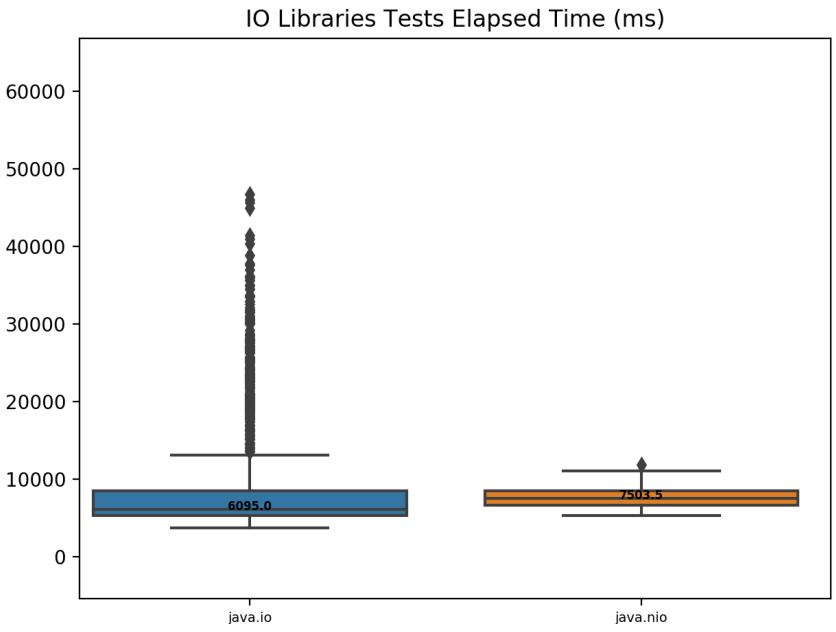


Figure 60: I/O Libraries Elapsed Time

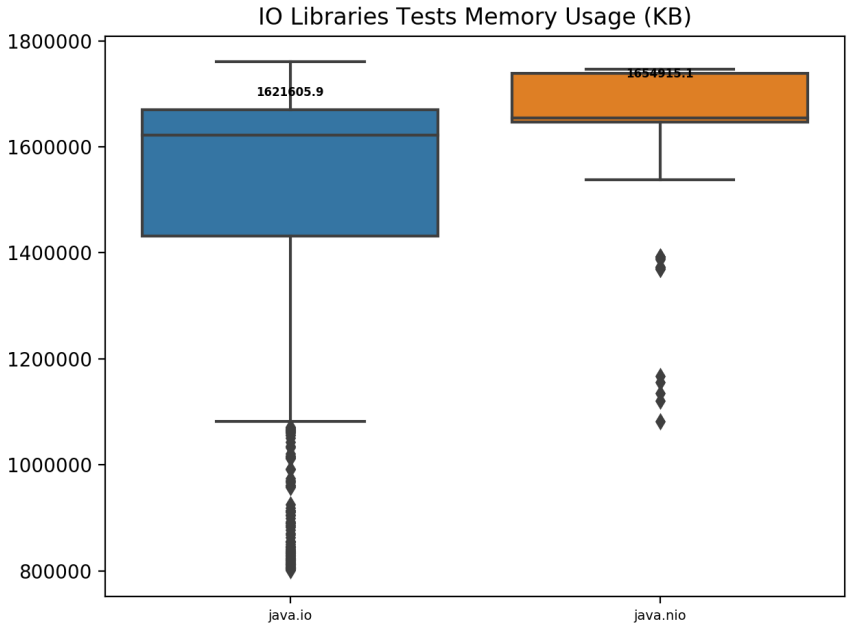


Figure 61: I/O Libraries Memory Usage