DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
**30**

HINA ANWAR

# Towards Greener Software Engineering Using Software Analytics

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

**30**

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS
**30**

**HINA ANWAR**

# Towards Greener Software Engineering Using Software Analytics

UNIVERSITY OF TARTU
Press

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in Computer Science on 4th November, 2021 by the Council of the Institute of Computer Science, University of Tartu.

*Supervisors*

Prof. Dr. Dietmar Alfred Paul Kurt Pfahl
University of Tartu
Estonia

Prof. Dr. Satish Narayana Srirama
Visiting Professor, University of Tartu, Estonia
Associate Professor, University of Hyderabad, India

*Opponents*

Assist. Prof. Dr. Luis Miranda da Cruz
Delft University of Technology (TU Delft)
Netherlands

Prof. Dr. Coral Calero
University of Castilla-La Mancha
Spain

The public defense will take place on December 14th, 2021 at 11:15 in Zoom.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

Copyright © 2021 by Hina Anwar

ISSN 2613-5906
ISBN 978-9949-03-767-4 (print)
ISBN 978-9949-03-768-1 (pdf)

University of Tartu Press
http://www.tyk.ee/

*To my daughter*

# ABSTRACT

In the current era of ubiquitous technology usage, gadgets like smartphones, tablets, and laptops are widely used. Since all of these devices are battery-operated, the question of energy efficiency has become one of the crucial parameters when users select a device. Energy efficiency aims at reducing the amount of energy required when providing products and services. The energy efficiency of a digital device has become part of its overall perceived quality.

Empirical studies have shown that mobile apps that do not drain battery usually get good ratings from users. Many studies have been published that present refactoring guidelines and tools to optimize the code to make mobile apps energy efficient. However, these guidelines cannot be generalized w.r.t. energy efficiency, as there is not enough energy-related data for every context. Existing energy enhancement tools and profilers are mostly prototypes applicable to only a small subset of energy-related problems. In addition, existing guidelines and tools focus on addressing energy issues *a posteriori*, i.e., once they have already been introduced into the code.

Android app code can be roughly divided into two parts: the custom code and the reusable code. Custom code is unique to each app. Reusable code includes third-party libraries that are included in apps to speed up the development process. As compared to desktop or web applications, Android apps contain multiple components that have user-driven workflows. A typical Android app consists of activities, fragments, services, content providers, and broadcast receivers. Due to the difference in architecture, the support tools used to develop traditional Java-based applications are not so useful in Android app development and maintenance.

We start by evaluating the energy consumption of various code smell refactorings in native Android apps. Then we conduct an empirical study on the impact of third-party network libraries used in Android apps. By analysing commonly used third-party network libraries in various usage scenarios, we show that the energy consumption of these libraries differs significantly. We discuss results and provide generalized contextual guidelines that could be used during app development.

Further, we conduct a systematic literature review to identify and study the current state of the art support tools available to aid the development of green Android apps. We summarize the scope and limitations of these tools and highlight research gaps. Based on this study and the experiments we conducted before, we highlight the problems in capturing and reproducing hardware-based energy measurements. We develop support tool **ARENA** (**A**nalysing ene**R**gy **E**fficiency in a**N**droid **A**pps) for the Android Studio IDE that could help in gathering energy data and analyzing the energy consumption of Android apps.

Last, we develop the support tool **REHAB** (**R**ecommending **E**nergy-efficient t**H**ird-p**A**rty li**B**raries) for the Android Studio IDE to recommend energy efficient third-party network libraries to developers during development.

# CONTENTS

**List of Figures** 12

**List of Tables** 13

**1. Introduction** 16
- 1.1. Problem Statement and Research Goals . . . . . . . . . . . . . . . 17
- 1.2. Research Approach . . . . . . . . . . . . . . . . . . . . . . . . . . 20
- 1.3. Contributions of the Thesis . . . . . . . . . . . . . . . . . . . . . 21
- 1.4. Structure of the Thesis . . . . . . . . . . . . . . . . . . . . . . . 22

**2. Background** 24
- 2.1. Android OS . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
- 2.2. Android Apps . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24
- 2.3. Android App Development Process . . . . . . . . . . . . . . . . . 25
- 2.4. Code Smells . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
- 2.5. Third-party Libraries . . . . . . . . . . . . . . . . . . . . . . . . 26
- 2.6. Software Sustainability . . . . . . . . . . . . . . . . . . . . . . . 27
- 2.7. Energy Measurement . . . . . . . . . . . . . . . . . . . . . . . . 28

**3. Related Work** 30
- 3.1. Code Smells Refactoring . . . . . . . . . . . . . . . . . . . . . . 30
- 3.2. Third-party Libraries . . . . . . . . . . . . . . . . . . . . . . . . 31
- 3.3. Tool Support for Developing Green Android Apps . . . . . . . . . 32

**4. Impact of Code Smell Refactoring on the Energy Consumption of Android Apps** 35
- 4.1. Research Method . . . . . . . . . . . . . . . . . . . . . . . . . . 35
  - 4.1.1. Research Questions . . . . . . . . . . . . . . . . . . . . . 35
  - 4.1.2. Code Smells . . . . . . . . . . . . . . . . . . . . . . . . . 36
  - 4.1.3. Code Smell Detection Tool and Refactoring . . . . . . . . 37
  - 4.1.4. Selected Apps . . . . . . . . . . . . . . . . . . . . . . . . 37
  - 4.1.5. Testing Tool . . . . . . . . . . . . . . . . . . . . . . . . . 37
  - 4.1.6. Energy Measurement . . . . . . . . . . . . . . . . . . . . 38
  - 4.1.7. Test Environment . . . . . . . . . . . . . . . . . . . . . . 38
  - 4.1.8. Experimental Design . . . . . . . . . . . . . . . . . . . . 39
- 4.2. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
  - 4.2.1. **RG1-RQ1:** *Is there a correlation between code smell refactoring and energy consumption of Android apps?* . . . . . . 41
  - 4.2.2. **RG1-RQ2:** *Is there a correlation between code smell refactoring and execution time of Android apps?* . . . . . . . . . 43
- 4.3. Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

4.4. Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . . 47

**5. Impact of Third-party HTTP Libraries on the Energy Consumption of Android Apps** **50**

5.1. Research Method . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.1.1. Research Questions . . . . . . . . . . . . . . . . . . . . . 51
5.1.2. Use Cases . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.1.3. Selected Libraries . . . . . . . . . . . . . . . . . . . . . . 53
5.1.4. Experimental Design . . . . . . . . . . . . . . . . . . . . 54
5.1.5. Test Environment . . . . . . . . . . . . . . . . . . . . . . 57
5.2. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.2.1. **RG1-RQ3-A:** *When making GET requests is there variance in energy consumption of Android third-party HTTP libraries?* 58
5.2.2. **RG1-RQ3-B:** *When making multi-part POST requests is there variance in energy consumption of Android third-party HTTP libraries?* . . . . . . . . . . . . . . . . . . . . . . 59
5.2.3. **RG1-RQ3-C:** *When sending structured JSON objects is there variance in energy consumption of Android third-party HTTP libraries?* . . . . . . . . . . . . . . . . . . . . . . 59
5.2.4. **RG1-RQ3-D:** *When receiving structured JSON objects is there variance in energy consumption of Android third-party HTTP libraries?* . . . . . . . . . . . . . . . . . . . . . . 61
5.2.5. **RG1-RQ3-E:** *When loading and displaying images on the screen is there variance in energy consumption of Android third-party HTTP libraries?* . . . . . . . . . . . . . . . . . 63
5.2.6. **RG1-RQ4:** *Is the energy consumption of a third-party HTTP library correlated with execution time?* . . . . . . . . . . . 64
5.3. Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.3.1. Recommendations . . . . . . . . . . . . . . . . . . . . . . 67
5.3.2. Energy Consumption versus Execution Time . . . . . . . . 69
5.3.3. Energy Drivers . . . . . . . . . . . . . . . . . . . . . . . . 69
5.3.4. Energy Consumption versus Popularity . . . . . . . . . . . 70
5.3.5. General Take-aways . . . . . . . . . . . . . . . . . . . . . 71
5.4. Threats to Validity . . . . . . . . . . . . . . . . . . . . . . . . . . 71

**6. Tool Support for Green Android Development** **73**

6.1. Research Method . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.1.1. Research Questions . . . . . . . . . . . . . . . . . . . . . 73
6.1.2. Search Query . . . . . . . . . . . . . . . . . . . . . . . . . 74
6.1.3. Screening of Publications . . . . . . . . . . . . . . . . . . 75
6.1.4. Classification and Analysis . . . . . . . . . . . . . . . . . 76
6.2. Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80

6.2.1. **RG2-RQ1:** *What state of the art support tools have been developed to aid software practitioners in detecting or refactoring Android specific code smells and energy bugs in Android apps?* . . . 82
6.2.2. **RG2-RQ2:** *What state of the art support tools have been developed to aid software practitioners in detecting or migrating third-party libraries in Android apps?* . . . 82
6.2.3. **RG2-RQ3:** *How do existing support tools compare to one another in terms of techniques they use for offering the support?* . . . 84
6.2.4. **RG2-RQ4:** *How do existing support tools compare to one another in terms of the support they offer to practitioners for improving energy efficiency in Android apps?* . . . 88
6.3. Discussion . . . 97
6.3.1. Support Tools for Code Smell and Energy Bug Detection or Refactoring . . . 97
6.3.2. Support Tools for Third-party Library Detection or Migration 98
6.4. Threats to Validity . . . 100

**7. ARENA: A Tool for Measuring and Analysing the Energy Efficiency of Android Applications 102**
7.1. ARENA Architecture . . . 103
7.1.1. Component 1: ExperimentRunner . . . 104
7.1.2. Component 2: CleanupRunner . . . 105
7.1.3. Component 3: AnalysisRunner . . . 106
7.1.4. Component 4: VisualizationRunner . . . 106
7.2. ARENA Implementation . . . 106
7.3. ARENA in Practice . . . 107
7.3.1. Comprehensive Usage Scenario . . . 107
7.3.2. Application Example . . . 108

**8. REHAB: A Tool for Recommending Energy-efficient Third-party Libraries to Android Developers 112**
8.1. REHAB Architecture . . . 112
8.1.1. Component 1: Knowledge Base . . . 113
8.1.2. Component 2: Code Inspection . . . 115
8.1.3. Component 3: Rule Mapping Module . . . 116
8.2. REHAB Implementation . . . 116
8.2.1. Usage Overview . . . 118
8.3. Scope Extension of REHAB . . . 121
8.4. REHAB Evaluation . . . 129

**9. Conclusion and Future Work 132**
9.1. Contributions and Findings . . . 132

9.1.1. Code Smell Refactorings . . . . . . . . . . . . . . . . . . . 132
9.1.2. Third-party Libraries . . . . . . . . . . . . . . . . . . . . . . 132
9.1.3. Tool Support for Developing Green Android Apps . . . . . 133
9.2. Opportunities for Future Work . . . . . . . . . . . . . . . . . . . . 134
9.2.1. Contextual Data for Refactoring . . . . . . . . . . . . . . . . 134
9.2.2. Trade-off Analysis . . . . . . . . . . . . . . . . . . . . . . . . 134
9.2.3. Energy Data for Creating Software Based Energy Prediction Models . . . . . . . . . . . . . . . . . . . . . . . . . . . 135

**Bibliography** **136**

**Appendix A. Statistics and Pairwise Comparison Results for Selected Third-party HTTP Libraries** **152**
A.1. Statistics for Selected Third-party HTTP Libraries . . . . . . . . . 152
A.2. Features and Methods of Selected Third-party HTTP Libraries . . 153
A.3. Results of Pairwise Comparison for the Mean Ranks of Energy by Library in Each Use-case . . . . . . . . . . . . . . . . . . . . . . . 154

**Appendix B. List of Selected Publications, Energy Bugs, and Code Smells** **162**
B.1. List of Selected Publications . . . . . . . . . . . . . . . . . . . . . 162
B.2. Android Energy Bugs Covered by Tools in the 'Detector' and 'Optimizer' Categories . . . . . . . . . . . . . . . . . . . . . . . . . 167
B.3. Android Code Smells Covered by Tools in 'Detector' and 'Optimizer' categories . . . . . . . . . . . . . . . . . . . . . . . . . . 168

**Appendix C. Instructions for Installing ARENA** **171**
C.1. Installation Pre-requisites . . . . . . . . . . . . . . . . . . . . . . 171
C.2. Plugin Installation . . . . . . . . . . . . . . . . . . . . . . . . . . 171

**Appendix D. Installation Instruction and Detailed Data - REHAB** **172**
D.1. Plugin Installation . . . . . . . . . . . . . . . . . . . . . . . . . . 172
D.2. Third-party Libraries and Version Used in Android Apps . . . . . 172

**Acknowledgement** **175**

**Sisukokkuvõte (Summary in Estonian)** **176**

**Curriculum Vitae** **178**

**Elulookirjeldus (Curriculum Vitae in Estonian)** **179**

**List of Original Publications** **180**

# LIST OF FIGURES

1.1. Structure of the thesis . . . . . . . . . . . . . . . . . . . . . . . 23
2.1. Android software stack . . . . . . . . . . . . . . . . . . . . . . . 24
2.2. Overview of Android app development process . . . . . . . . . 26
2.3. Dimensions of software sustainability . . . . . . . . . . . . . . . 28
2.4. Energy measurement process . . . . . . . . . . . . . . . . . . . . 29
4.1. Energy consumption in joules for Calculator app per treatment . 42
4.2. Energy consumption in joules for Todo-List app per treatment . 42
4.3. Energy consumption in joules for ‘Openflood’ app per treatment 42
4.4. Execution time in seconds for Calculator app per treatment . . . 44
4.5. Execution time in seconds for Todo-List app per treatment . . . 44
4.6. Execution time in seconds for ‘Openflood’ app per treatment . . 45
5.1. For each use case, the library used for making app versions and the respective app versions . . . . . . . . . . . . . . . . . . . . 55
5.2. Mean energy consumption per library in UC-GF . . . . . . . . . 58
5.3. Mean energy consumption per library in UC-PF . . . . . . . . . 60
5.4. Mean energy consumption per library in UC-PJO . . . . . . . . 60
5.5. Mean energy consumption per library in UC-GJO . . . . . . . . 62
5.6. Mean energy consumption per library in UC-GI . . . . . . . . . 63
5.7. Flowchart summarizing recommendations . . . . . . . . . . . . 68
6.1. Publication per year per category (Search Query 1) . . . . . . . 83
6.2. Publication per year per category (Search Query 2) . . . . . . . 83
6.3. Percentage of the tools in ’Detector’ and ’Optimizer’ categories that can detect Android energy bugs. . . . . . . . . . . . . . . . 92
6.4. Percentage of code smells detected by each tools in ‘Detector’ and ‘Optimizer’ categories. . . . . . . . . . . . . . . . . . . . . . . 93
7.1. An overview of the energy measurement and analysis process that is supported by ARENA . . . . . . . . . . . . . . . . . . . . . 103
7.2. Detailed workflow supported by ARENA tool . . . . . . . . . . 109
7.3. ARENA interface - Data collection tab . . . . . . . . . . . . . . 110
7.4. ARENA interface - Pre-processing tab . . . . . . . . . . . . . . 110
7.5. ARENA interface - Analysis tab . . . . . . . . . . . . . . . . . . 111
7.6. ARENA interface - Visualization tab . . . . . . . . . . . . . . . 111
8.1. An overview of REHAB . . . . . . . . . . . . . . . . . . . . . . 113
8.2. REHAB tool window . . . . . . . . . . . . . . . . . . . . . . . . 119
8.3. REHAB usage example . . . . . . . . . . . . . . . . . . . . . . . 120
8.4. Number of versions per change type per library . . . . . . . . . 122

# LIST OF TABLES

4.1. Characteristics of selected Android apps . . . . . . . . . . . . . 37
4.2. Number of refactorings for Calculator app . . . . . . . . . . . . 40
4.3. Number of refactorings for Todo-List app . . . . . . . . . . . . 40
4.4. Number of refactorings for Openflood app . . . . . . . . . . . . 40
4.5. ANOVA results for all apps (energy consumption) . . . . . . . . 41
4.6. Overview of energy consumption results (RG1-RQ1) . . . . . . 43
4.7. ANOVA results for all apps (execution time) . . . . . . . . . . . 44
4.8. Overview of execution time results (RG1-RQ2) . . . . . . . . . 45
4.9. Energy consumption results when a permutation of code smell refactorings was applied. . . . . . . . . . . . . . . . . . . . . . . 46
5.1. Android third-party HTTP libraries used in each selected use case for making app versions. . . . . . . . . . . . . . . . . . . . . . 56
5.2. Results for RG1-RQ3-A and summary statistics (UC-GF) . . . . 58
5.3. Results for RG1-RQ3-B and summary statistics (UC-PF) . . . . 60
5.4. Results for RG1-RQ3-C and summary statistics (UC-PJO) . . . 61
5.5. Results for RG1-RQ3-D and summary statistics (UC-GJO) . . . 62
5.6. Results for RG1-RQ3-E and summary statistics (UC-GI) . . . . 63
5.7. RG1-RQ4 Results and summary statistics . . . . . . . . . . . . 64
5.8. Spearman correlation results between energy and execution time for libraries in all use cases . . . . . . . . . . . . . . . . . . . . 65
6.1. Search query filter . . . . . . . . . . . . . . . . . . . . . . . . . 75
6.2. Quality assessment criteria . . . . . . . . . . . . . . . . . . . . 76
6.3. Categories of support tools (RG2-RQ1) . . . . . . . . . . . . . 77
6.4. Categories of support tools (RG2-RQ2)) . . . . . . . . . . . . . 78
6.5. Categories of techniques used in support tools for code smells or energy bugs (RG2-RQ3) . . . . . . . . . . . . . . . . . . . . . 78
6.6. Categories of techniques used in support tools for third-party libraries (RG2-RQ3) . . . . . . . . . . . . . . . . . . . . . . . . 79
6.7. Number of publications extracted per online repo. (search query 1) 81
6.8. Number of publications per screening step (search query 1) . . . 81
6.9. Quality score assigned to each selected publication (search query 1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
6.10. Number of publications extracted per online repo. (search query 2) 81
6.11. Number of publications after applying filters and screening steps (search query 2) . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.12. Quality score assigned to each selected publication (search query 2) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.13. Distribution of publications in each category (RG2-RQ1) . . . . 82
6.14. Distribution of publications in each category (RG2-RQ2) . . . . 83

6.15. Overview of support tools (for code smells or energy bugs detection and refactoring) showing the technique used for offering support to developers (RG2-RQ3) . . . . . . . . . . . . . . . . 84
6.16. Overview of support tools (for third-party library detection and migration) showing the technique used for offering support to developers (RG2-RQ3) . . . . . . . . . . . . . . . . . . . . . . . . 85
6.17. List of support tools in 'Profiler', 'Detector', and 'Optimizer' categories along with information about their inputs and outputs, user interface, IDE support and availability (RG2-RQ4) . . . . . . . 89
6.18. List of support tools in 'Identifier', 'Migrator', and 'Controller' categories along with information about their inputs and outputs, library coverage, UI, and availability (RG2-RQ4) . . . . . . . . 94
8.1. Android third-party HTTP libraries used in each selected use case. 114
8.2. Unique combinations of use-cases. . . . . . . . . . . . . . . . . . 115
8.3. Based on mean energy values the calculated minimum energy loss values are shown for selected third-party HTTP libraries in UC-GF and UC-PF . . . . . . . . . . . . . . . . . . . . . . . . . . 115
8.4. Elements for detecting different types of HTTP requests . . . . . 117
8.5. Number of versions per library available in MVN-central repository (as of January 2021). . . . . . . . . . . . . . . . . . . . . . 122
8.6. Number of groups created per library in which versions were divided. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
8.7. Number of apps in which selected third-party HTTP libraries were detected (irrespective of the version). . . . . . . . . . . . . . . . 123
8.8. Group of versions created for Volley library along with number of apps in which these groups were detected . . . . . . . . . . . . 124
8.9. Group of versions created for Retrofit library along with number of apps in which these groups were detected . . . . . . . . . . . 124
8.10. Group of versions created for OkHttp library along with number of apps in which these groups were detected . . . . . . . . . . . 125
8.11. Group of versions created for AndroidAsync library along with number of apps in which these groups were detected . . . . . . 126
8.12. Group of versions created for AsyncHttp library along with number of apps in which these groups were detected . . . . . . . . . 127
8.13. Group of versions created for Glide library along with number of apps in which these groups were detected . . . . . . . . . . . . 127
8.14. Group of versions created for Picasso library along with number of apps in which these groups were detected . . . . . . . . . . . 128
8.15. Group of versions created for UIL (Universal Image Loader) library along with number of apps in which these groups were detected . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 129
8.16. REHAB evaluation results (app-set 1) . . . . . . . . . . . . . . 130
8.17. REHAB evaluation results (app-set 2). . . . . . . . . . . . . . . 131

A.1. Statistics for selected libraries gathered from Awesome Android website (Oct. 2019) . . . . . . . . . . . . . . . . . . . . . . . . . 152
A.2. Statistics for selected libraries gathered from the 'AppBrain' website. (Oct. 2019) . . . . . . . . . . . . . . . . . . . . . . . . . . . 153
A.3. HTTP request methods . . . . . . . . . . . . . . . . . . . . . . 153
A.4. Features in selected network third-party libraries . . . . . . . . 154
A.5. Features in selected image loading third-party libraries . . . . . 154
A.6. Pairwise comparisons for the mean ranks of energy by library (UC-GF) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
A.7. Pairwise comparisons for the mean ranks of energy by library (UC-PF) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
A.8. Pairwise comparisons for the mean ranks of energy by library (UC-PJO) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 155
A.9. Pairwise comparisons for the mean ranks of energy by library (UC-GJO) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 158
A.10. Pairwise comparisons for the mean ranks of energy by library (UC-GI) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 161
B.1. Android energy bugs detected by each tool in 'Detector' and 'Optimizer' categories. . . . . . . . . . . . . . . . . . . . . . . . . 167
B.2. Android code smells detected by each tool in the 'Detector' and 'Optimizer' categories. . . . . . . . . . . . . . . . . . . . . . . 170
D.1. Group of versions created for Gson library along with number of apps in which these groups were detected . . . . . . . . . . . . 172
D.2. Group of versions created for Jackson library along with number of apps in which these groups were detected . . . . . . . . . . . 173
D.3. Group of versions created for Moshi library along with number of apps in which these groups were detected . . . . . . . . . . . . 174

# 1. INTRODUCTION

Global warming due to $CO_2$ emission has become one of the most prominent environmental issues in the past decades. A part of this $CO_2$ emission is contributed by the information and communication technology (ICT) industry [64]. Therefore, producing green and sustainable products and practices has been the focus of many researchers in the ICT community. Recently, the focus of research in the ICT community has shifted from optimizing the energy consumption of hardware to optimizing the energy consumption of software [11, 13, 24, 26, 40, 52, 62, 79, 88, 91, 109, 120, 139].

Software systems have such a significant impact on our everyday lives that changes towards environmental sustainability can ripple to other systems with which they interact and positively affect the industries in which they are used. This impact can be direct, indirect, or occur as a rebound effect [23]. As software indirectly consumes energy by controlling the equipment. An efficiently designed software might use resources optimally, thus reducing energy consumption [70, 84, 111]. A green or sustainable software is the one that is developed and used in such a way that it leaves a minimum negative impact on users, environment, economy and society in general [48]. Therefore, green software engineering consists of processes and practices that help produce sustainable software and everything related to the software product, be it development or maintenance, taking environmental aspects into account [23]. In "green software engineering practitioners care and think about energy when they build applications" [102].

In the current era of ubiquitous technology usage, gadgets like smart phones, tablets, and laptops are widely used. Among portable devices, mobile phones are the most commonly used. Statistics show that the usage of mobile devices will grow in the coming years [50], indicating an increase in the carbon footprint. Since mobile devices are battery operated, the question of energy efficiency has become one of the crucial parameters when users select a device. Energy efficiency aims at reducing the amount of energy required when providing products and services. In order to produce energy efficient devices, the software architecture, design, and code all need to be developed with the awareness that the software product will be power efficient or, in other words, green. However, software practitioners lack the necessary information and support infrastructure required to produce green software products [102]. Such information could be gathered via software analytics, which combines information from different software artefacts and converts it into useful information for software practitioners.

The term 'Android development' refers to the development of apps that are developed to operate on devices running the Android operating system. These apps could be developed in various languages; however, in this thesis, we focus

on Android development in Java. Android development differs from traditional software development in terms of context, user-experience and a touch-based interface. Android apps are designed for portable devices, which have limited resources such as memory or battery. A common struggle during Android app development is how to make the apps efficient in terms of resource usage. Banarjee et al. summarize the problem nicely as follows:

*"High computational power coupled with small battery capacity and the application development in an energy-oblivious fashion can only lead to one situation: short battery life and an unsatisfied user base"* [19].

Previously, studies have explored mobile app stores in order to define procedures to optimize energy consumption of apps [103, 118, 136, 156, 166, 170, 172]. Some studies have focused on profiling energy [16, 17, 32, 82, 125] consumed by apps, while others have developed support tools [20, 55]. As compared to desktop or web applications, Android apps contain multiple components that have user-driven workflows. A typical Android app consists of activities, fragments, services, content providers, and broadcast receivers. Due to the difference in architecture, the support tools used in the development of traditional Java-based applications are not so useful in Android app development and maintenance.

## 1.1. Problem Statement and Research Goals

Android app code can be roughly divided into two parts: the custom code and the reusable code. Custom code is unique to each app. Reusable code includes third-party libraries that are included in apps to speed up the development process. To produce a mobile app it is a common practice of developers to combine custom app code with library modules. As compared to app code which is unique to the project, libraries are reusable services that hide the low-level complexities and provide developers with a higher level abstraction for certain features. As the mobile app evolves the app code evolves with it while the library modules usually stay static and are not updated by the app developers [44, 87, 147]. In the domain of Android app development, research has been focused on development activities related to energy efficiency, memory usage, performance etc., and maintenance activities related to code smell detection and correction, energy bug detection and correction, detection and migration of third-party libraries etc.

Code smells are an indication of possible problems in source code or design of the apps. Such problems could be avoided by refactoring the code [59]. However, refactoring the code smells could impact the energy consumption of mobile apps as internal structure of the code is changed. In Android development, code smells typically appear due to frequent development and update cycles of

apps. As the source code of Android apps is mostly in Java, therefore, traditional code smells (as defined by M. Fowler [59]) can occur. These code smells are related to maintainability and can appear in applications irrespective of the platform. In addition to traditional code smells, there are Android-specific code smells that appear in Android apps. Android-specific code smells represent bad programming practices in Android apps and could impact maintainability and other non-functional requirements such as performance, security, sustainability etc. Many studies [72, 73, 122, 123, 135] have focused on identifying and cataloguing Android-specific code smells and profiling the energy consumption of Android-specific code smell refactorings. Energy bugs are scenarios that cause unexpected energy drains, such as preventing the mobile device from going into the idle state even after the app execution has been completed. Such malfunctioning can cause battery drains and should be avoided [19]. To build an energy-efficient Android app, developers need to identify and refactor code smells and energy bugs.

Third-party libraries are reusable components available to implement various functionalities in an app, such as billing, advertisement, and networking. Up till June 2021, the online Maven repository[1] contained 15,730,282 unique libraries. Such a huge supply of third-party libraries is linked to the demands and needs of developers [174]. Almost sixty percent of code in Android apps is related to third-party libraries [168]. However, these libraries could introduce various security, privacy, permission, and resource usage related issues in apps [179]. The research on the detection or migration of third-party libraries has many uses. Some studies have used third-party library detection techniques for finding security vulnerabilities [65, 77, 104, 115] in Android apps. While others have focused on privacy leaks [21, 60, 61, 80, 169]. Third-party libraries have been detected and removed as noise in clone, app repackage, and malicious app detection studies [27, 94, 150, 178, 181]. Third-party libraries are detected and removed from these studies in order to improve the accuracy of the analysis. Studies related to the energy impact of third-party Android libraries are limited [154].

To improve app code for energy efficiency developers could apply refactoring alternatives that are energy efficient. In case of libraries, choosing a stable and reliable third-party library is important as app success depends on it, however, if that third-party library is not energy efficient and will not be updated anytime soon it might negatively impact battery consumption. Despite a large number of studies published related to the energy efficiency of apps, there is still a pronounced gap in literature in terms of providing generalized contextual guidelines and open source support tools for improving the energy efficiency of apps. This gap exists because of the following problems:

- Research results related to energy consumption in software apps are con-

[1] https://search.maven.org/stats, Statistics for central repository

strained by specific contexts and might not be convincing enough for app developers in their work environment.

- The techniques used for measuring the energy consumption of apps during such evaluations usually have a steep learning curve and are unique by design. Therefore, they are not easy to adopt, reproduce or reuse.
- There is a lack of free open source support tools that could be used during energy measurements. Measurements are performed by writing specialized scripts for selected hardware and software components.

The research goal of this thesis is to help overcome these problems by evaluating the energy consumption of various code smell refactorings and third-party libraries used in Android app development, with the intention to add data to the already existing body of knowledge and to provide support tool(s) that could integrate with the current Android Studio IDE and be usable by app developers without the in-depth knowledge of energy issues related to mobile apps.

In this thesis, we address the following research goals:

**Main Research Goals**

RG1: To understand the influence of mobile apps' code structure on energy efficiency.

RG2: To improve tool support for developing energy efficient mobile apps.

Regarding RG1, we evaluate alternative coding patterns to save energy in mobile apps in two contexts 1) custom app code written by the developers; 2) library modules, i.e., reusable services added to the project. Research related to RQ1 will be interesting for mobile app developers as it will add to the already existing evidence on what is or is not energy efficient and will provide recommendations to the developers in some typical use cases. To address RG1, we answer the following research questions.

**RG1-RQ1:** *Is there a correlation between code smell refactoring and energy consumption of Android apps?* (Chapter 4)

**RG1-RQ2:** *Is there a correlation between code smell refactoring and execution time of Android apps?* (Chapter 4)

**RG1-RQ3:** *Is there variance in the energy consumption of Android third-party HTTP libraries?* (Chapter 5)

**RG1-RQ4:** *Is the energy consumption of a third-party HTTP library correlated with execution time?* (Chapter 5)

Regarding RG2, we explore the state of the art support tools for energy efficient app development and investigate their strengths and weaknesses. More precisely, we answer the four research questions listed below. This research will be interesting for researchers who want to conduct similar controlled experiments (as conducted in RG1) for Android-specific code smells or energy bugs or third-party libraries. Based on the answers to the research questions listed below, we provide two new support tools that overcome some of the limitations of the existing tools (Chapters 7 and 8).

**RG2-RQ1:** *What state of the art support tools have been developed to aid software practitioners in detecting or refactoring Android specific code smells and energy bugs in Android apps?* (Chapter 6)

**RG2-RQ2:** *What state of the art support tools have been developed to aid software practitioners in detecting or migrating third-party libraries in Android apps?* (Chapter 6)

**RG2-RQ3:** *How do existing support tools compare to one another in terms of techniques they use for offering the support?* (Chapter 6)

**RG2-RQ4:** *How do existing support tools compare to one another in terms of the support they offer to practitioners for improving energy efficiency in Android apps?* (Chapter 6)

## 1.2. Research Approach

To address RG1, we first conduct controlled experiments, to evaluate the energy consumption of various traditional code smell refactorings and third-party HTTP libraries commonly used in Android app development. We choose Android based development for investigation, because Android holds the largest market share and is open source. Android also provides the capabilities to manage different features using libraries and APIs. We select Java as the programming language for investigation, because it is the native language for Android developers and is among the most popular languages for Android development. We selected code smell types as well as their refactorings, from Fowler et al.'s [59] list. We assume that the types of code smells identified by Fowler et al. occur in Java code independent of platform. The third-party libraries are selected based on usage statistics and rankings provided on the popular third-party market (AppBrain[2]) and catalogue (Awesome Android[3]). We identified the typical use cases from the most downloaded apps in the Google play store. The selected third-party libraries could be used as alternatives to one another in each use case. To measure the energy consumption, we use hardware based energy measurements as they are more

[2] https://www.appbrain.com/stats/libraries

[3] https://android.libhunt.com

accurate. Statistical analysis is done on the energy data to measure the variance, effect size and percentage change in energy.

To address RG2, we study and analyze what is already built and what is needed. We conduct a systematic mapping study to identify and compare the existing support tools based on the support they offer to develop green Android apps. In the related literature, the energy impact of refactoring Android-specific code smells is not consistent. We know about some tools that detect and refactor energy bugs, among them, some also include detection for Android-specific code smells. We systematically check support tools that provide coverage for energy bugs and coverage for Android-specific code smells to aid the development of green Android apps. We also analyze support tools available to measure the energy consumption of apps. During the controlled experiments conducted to answer RG1, we found it challenging and resource-intensive to benchmark the energy consumed by different code smell refactorings and third-party libraries. The energy measurement process is human-intensive, and there are limited tools available to help automate this process. Therefore, manual effort is required to apply refactorings and use libraries in various contexts and combinations. Based on the analysis of state of the art, we develop the support tool ARENA to help automate the energy measurement process and thus making it more reliable and accurate. We also analyze support tools available to detect and migrate third-party libraries in Android apps. We systematically check existing support tools for coverage of the selected third-party libraries, energy-related support and IDE integration. We wanted to make the results from investigating the energy impact of various third-party libraries in RG1 readily available to developers. Based on the analysis of state of the art, it was clear that no tool offered energy-related support and IDE integration for third-party HTTP libraries. Therefore, we build another support tool REHAB that recommends energy-efficient third-party libraries to the developers.

## 1.3. Contributions of the Thesis

In this thesis we make three contributions that we hope will help developers and researchers estimate and improve the energy efficiency of mobile apps.

***Contribution 1:*** *Impact of code smell refactoring on the energy consumption of Android apps*. We investigated the custom app code of mobile apps and modify the code via code smell refactorings in an effort to find the energy efficient refactorings. For this purpose we conducted an experiment to assess 1) if there is a correlation between code smell refactorings and energy consumption, 2) if there is a correlation between code smell refactorings and execution time. The results of this experiment provide the following insights: 1) impact of refactoring only a single type of code smell on energy consumption was not consistent. However, where the effect size was medium or large, energy consumption decreased due to

refactoring. 2) Specific permutations of code smell refactorings should be used with caution as their energy impact might vary strongly depending on the selected Android app. 3) A significant reduction in energy consumption of Android apps does not necessarily correlate with a significant reduction or increase of execution time.

***Contribution 2:*** *Impact of third-party HTTP libraries on the energy consumption of Android apps.* The right choice of third-party libraries used in mobile apps is extremely important as it can affect the code quality. Third-party libraries save effort to write specialized code and speed up the development. We measured the energy consumption of alternative third-party Android HTTP libraries for selected use cases to assess 1) which one of the selected libraries are more energy efficient in a particular use case such as for making a multipart POST request to the server or for making a GET request from the server or for loading images from the server etc. 2) Is there a correlation between execution time and energy consumption of the selected third-party libraries. The results of our experiment show that there is a significant variance of energy consumption between the selected Android third-party HTTP libraries.

***Contribution 3:*** *Tool support for green Android development.* Aiming at identifying the problems that are resulting in a lack of support tools for energy efficient programming, it is important to first understand the current tools, methods, models, and profilers developed so far. To accomplish this, we conducted a systematic mapping study. Based on the results of this study, and the energy data produced in the contribution 2, we developed a support tool REHAB that recommends energy efficient third-party HTTP libraries to the developers. We also developed another support tool ARENA, to help automate the energy measurement process and to reduce the risks related to human errors during energy measurements. Both tools are open source and integrate with the Android Studio and IntelliJ IDEs.

The above contributions have been documented in publications III, VI, and VII listed at the end of the thesis (see "List of original publications")

## 1.4. Structure of the Thesis

Sections 1.1 to 1.3 provided the context for the thesis, outlined research goals, presented our research approach, and described contributions of the thesis. Figure 1.1 gives an overview of the thesis structure.

**Chapter 2** provides the background of how energy estimation can assist developers in making energy efficient Android apps. We also introduce the concepts relevant to this thesis.

**Chapter 3** positions our research by summarizing literature related to app code and related measures for improving energy efficiency in Android apps.

**Chapter 4** tackles RQ1 and RQ2 of RG1 of the thesis and is based on the work from [158]. This work was published in the proceedings of the *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'19)*.

**Chapter 5** tackles RQ3 and RQ4 of RG1 of the thesis and is based on the work from [154]. This work was published in the proceedings of the *IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems (MOBILESoft'20)*.

**Chapter 6** tackles RQ1 to RQ4 of RG2 of the thesis and is based on work from [155]. This work was published as book chapter in the Springer book *Software Sustainability* .

**Chapter 7** addresses RG2 and provides a description of the support tool ARENA.

**Chapter 8** addresses RG2 and provides a description of the support tool REHAB.

**Chapter 9** gives the concluding remarks and points out possible future research directions.

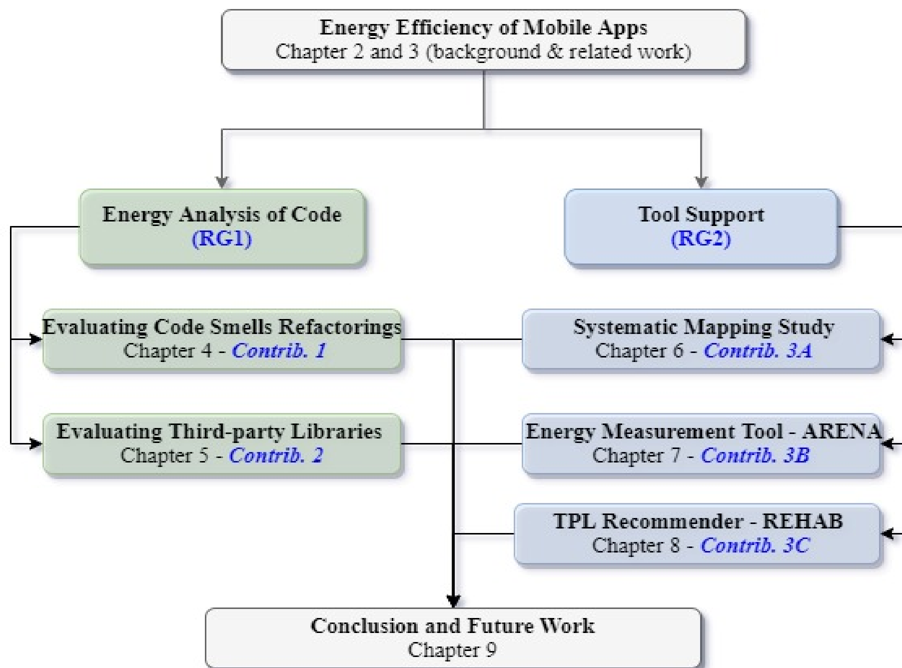


Figure 1.1: Structure of the thesis

# 2. BACKGROUND

In this chapter, first, we explain the key Android concepts, Android app development process and how code smells and third-party libraries effect Android apps. Then, we introduced the sustainability concept and approaches used to measure energy consumption of Android apps.

## 2.1. Android OS

Android is a Linux based mobile operating system developed by Open Handset Alliance[1] and Google. It was launched in 2007. It is free and open-source, commonly referred to as Android open source project (ASOP). Figure 2.1 gives an overview of the Android software stack (we derived this figure from the detailed Android software stack figure published in the Android developer blog[2])

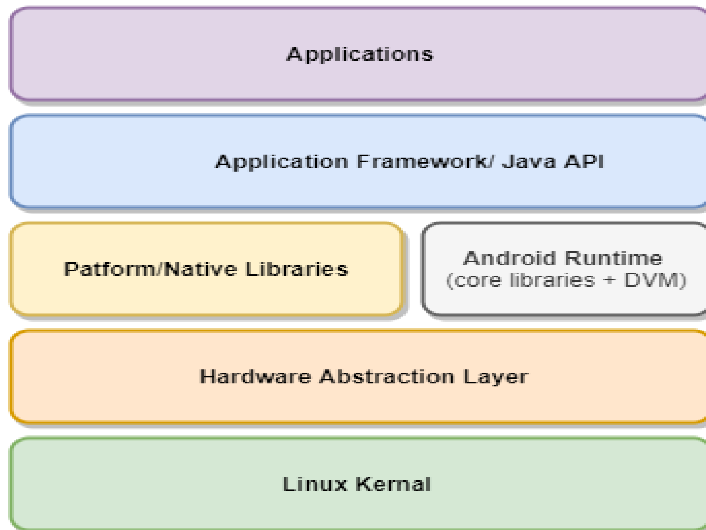


Figure 2.1: Android software stack

## 2.2. Android Apps

Android software development kit (SDK) tools compile all contents of an Android app along with any data and resource files into an Android package called APK, which is an executable file with .apk extension. The APK file can be installed on any device with the Android OS. Upon execution a unique Linux user ID is assigned to each app (as Linux is multi-user system in which each app is a user).

[1] https://www.openhandsetalliance.com/index.html
[2] https://developer.android.com/guide/platform

Only apps with correct user IDs and granted permissions can access files within an app. Each app has its own process which is started when some component of the app is executed and is destroyed when it is no longer needed. As compared to desktop applications, Android apps have a complex structure containing multiple app components such as activities, fragments, services, content providers, and broadcast receivers[3]. An Android app code roughly consists of two parts: 1) custom app code written by the developers, and 2) library modules, i.e., reusable services added to the project. The third-party libraries included in an app use underlying system APIs in the Android software stack to perform specific tasks.

## 2.3. Android App Development Process

The basic workflow to build an Android app is same as for other app platforms [4]. However, building an app is not just the basic work flow. It combines software development, project management and continuous integration activities as well. Therefore, we present a high level view of the Android app development process divided into five phases as shown in Figure 2.2 (Figure derived from Android developer blog[4]). The first phase 'discover and define' involves conceptualizing and clarifying app functionalities, goals and potential users. Features are prioritized based on requirement documents and milestones are set accordingly. The second phase 'design' involves creating prototypes/wireframes to understand the business viewpoint and improve the user experience by helping to design a good user interface. The third phase 'development and quality assurance/testing' involves setting up the environment for development. Android app development is an iterative process. Typically, a project manager, product owner or chief architect starts by setting up a project management system such as JIRA[5] and a communication channel such as Slack[6]. The development process is divided into a series of short development cycles, where each cycle consist of refined/prioritized requirements, planning, coding, debugging and testing. During development, developers have full control over the selection of software development kits (SDKs), as well as native and third-party libraries included in the project. The fourth phase 'publishing' involves preparing the app for release by configuring, building and testing the app in release mode. Finally, the app is marketed and distributed to the users via an app store. The fifth and last phase, 'maintenance and support' involves post deployment tasks such as tracking the causes of crashes, gathering follow-up statistics, gathering user feedback and reviews, and planing new features and updates.

[3] https://developer.android.com/guide/components/fundamentals
[4] https://tool.oschina.net/uploads/apidocs/android/guide/developing/index.html
[5] https://www.atlassian.com/software/jira
[6] https://slack.com/intl/en-ee/

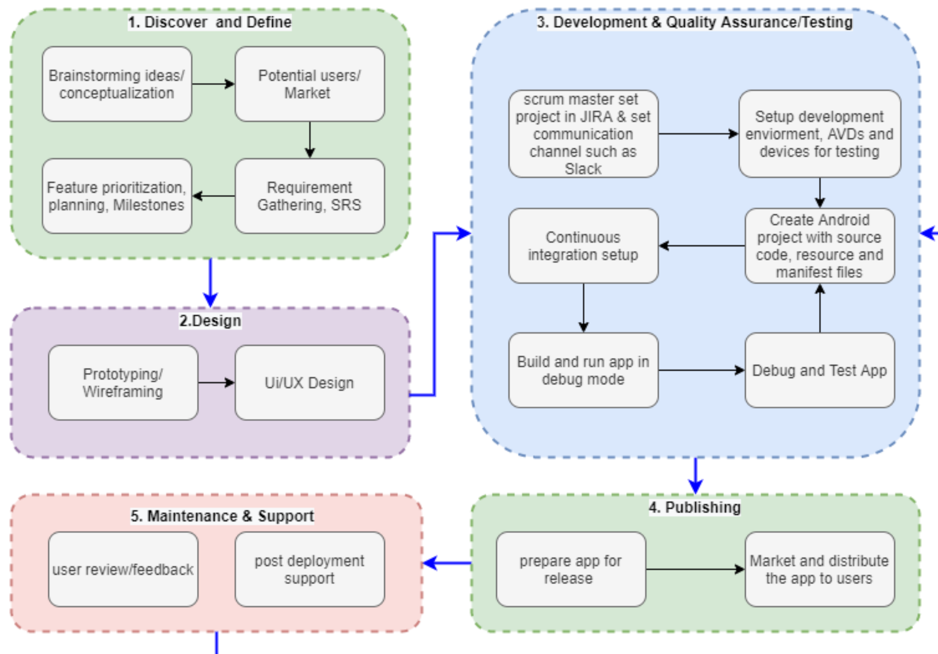


Figure 2.2: Overview of Android app development process

## 2.4. Code Smells

Due to time constraints and rapid release cycles, code maintainability might decrease. In order to implement a current requirement quickly, developers may apply solutions that make future changes in the code costly. Code smells are design and implementation patterns in an app source code that leads to decrease in software maintainability. Code smells can be removed via Refactoring. Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [59] . Manual refactoring of code is a tedious task. Automated systems designed to detect code smells are commonly referred as static code analyzers. Static code analyzers perform static analysis on code without executing it. M.Fowler [59] has defined many commonly occurring code smells in Java applications along with their refactorings. An example of a code smell is Long Method. Long Method is a smell that points to a method which has become too long from the beginning or over time and could affect the maintainability of the app in the long run. A possible refactoring technique is to divide the long method into smaller methods which are called from inside the original method.

## 2.5. Third-party Libraries

Third-party libraries are reusable components provided by other developers than the one who is developing a new app. The reusable components typically help im-

plement various standard functionalities in an app, such as billing, advertisement, networking and so forth. Up till June 2021, the online Maven repository[7] contained 15,730,282 unique libraries. Such a huge supply of third-party libraries is linked to the demands and needs of developers [174]. Almost sixty percent of code in Android apps is related to third-party libraries [168]. However, these libraries could introduce various security, privacy, permission, and resource usage related issues in apps [179]. There are various online catalogues (such as Awesome Android[8], AppBrain[9] etc.) available where developers can search for an appropriate library that they can include in their project. In these online catalouges, third-party libraries are divided into various categories bases on functionalities they offer. Developers can compare similar third-party libraries within a category based on statistics such as popularity, activity, stability etc. Selection of third-party library is time consuming process that requires that developer go through documentation and gather statistics about these libraries to make an informed decision.

## 2.6. Software Sustainability

The term software sustainability envelopes sustainability by software and sustainability in software. Sustainability by software means using software products to make other domains of life more sustainable. Sustainability in software refers to the study and practice of designing, developing, maintaining and disposing of software products in a way that it has minimal negative impact on the environment, community, economy, individuals and technology [22, 127]. In this thesis, we focus on sustainability in software. Usually three types of resources are required in software processes i.e., human resources, economic resources and energy resources. Therefore, software sustainability can be divided into dimensions [22] based on resources required in software processes. Figure 2.3 (figure derived from [22]) provides an overview of software sustainability dimensions .

- *Social Dimension*: encompasses software development and maintenance and how it affects the software development community.
- *Economic Dimension*: encompasses software lifecycle processes that are related to stakeholders, risk reduction, investment etc.
- *Environmental/Green software Dimension*: encompasses development of software products and how their usage and maintenance affect energy consumption.

[7] https://search.maven.org/stats, Statistics for central repository

[8] https://android.libhunt.com/

[9] https://www.appbrain.com/

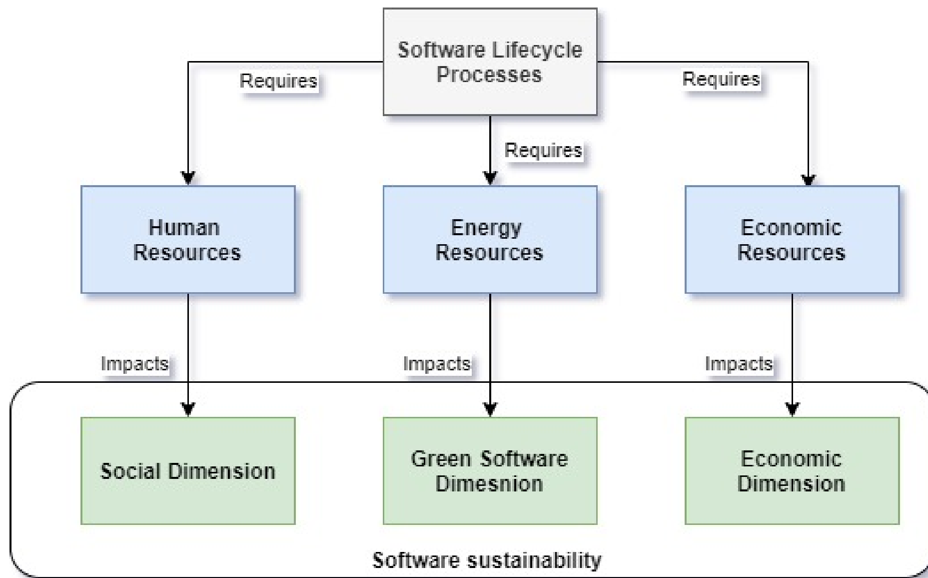


Figure 2.3: Dimensions of software sustainability

## 2.7. Energy Measurement

Energy consumption is an accumulation of power dissipation over time. Power is not the same as energy. Energy refers to the ability of making a change in the state of something while power refers to the rate of energy transfer. Power is measured in watts, whereas energy is measured in joules. As an example, if a task requires five seconds to complete and use ten watts, it consumes 50 joules of energy. In the context of software engineering, when measuring energy consumption we must take into account the hardware, context and time. The choice of hardware or platform such as WiFi or 3G can effect energy. Similarly, the context in which software is built and used affects energy consumption. Often, Android developers use the reduction in execution time as a proxy for reduction in energy consumption, which is not always true. For example, reducing the execution time via increased performance may use the CPU in a high power state, thus causing an increase in energy consumption. A typical energy software measurement process involves 1) setting up a measurement environment, 2) executing the app under test on a mobile device, 3) recording energy data via hardware based or software based approach, and 4) cleaning and aggregating data for analyses, reports and visualizations. There are two different approaches for capturing energy measurements 1) software based approach, 2) hardware based approach.

***Software based approach:*** Software based approaches are easier to implement and are based on estimation of a set of features such as system calls, number of packets transferred over the network, or cpu cycles etc. for measuring energy consumption. Software based approaches are not very accurate.

***Hardware based approach:*** Hardware based approaches are difficult to set up and depend on a physical measurement device. However, once the one time setup is done they are more accurate than software based approaches. The current/voltage data is usually recorded at the rate of 5kHz and higher. Figure 2.4 gives an overview of all the steps needed to complete the energy measurement process via hardware based approach. We prepare app version(s) with test cases that could mimic user interaction in a given use-case(s). Next, we execute the test scenario several times on the mobile phone and energy measurements are captured via physical measurement device such as Monsoon power monitor. Then, energy data is collected, filtered and aggregated. The captured energy data and related ADB logs are filtered by matching timestamps to identify start and end of test scenario. In the end, statistical analysis is performed on filtered data to identify significant changes in energy consumption.

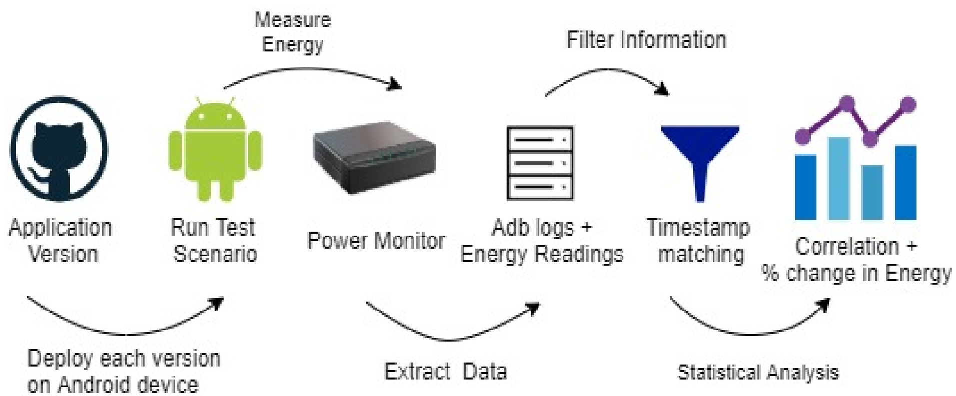


Figure 2.4: Energy measurement process

# 3. RELATED WORK

This chapter reviews the existing work related to app code and related measures for improving energy efficiency in Android apps. To put our research in perspective, we organized the existing literature along three directions relevant to our research questions; (1) code smells refactoring, (2) third-party libraries, (3) support tools.

## 3.1. Code Smells Refactoring

Code smell refactoring in Android apps has been discussed with respect to quality characteristics such as performance and maintainability [116, 175], but the energy impact of code smells has not yet been fully explored [69, 132, 141, 165]. Studies by Pinto and Kamei [131], Yamashita and Moonen [176] indicate that developers do care about code smells (such as 'Long Method', 'Feature Envy') and conduct refactoring. Verdecchia et al. [163] discussed the energy impact of five code smell refactorings on three ORM-based Java web applications. Their results indicate that in one out of three apps code smell refactoring significantly impacted energy consumption. Castillo et al. [132] analyzed the energy impact of 'God Class' refactoring on two Java applications. Their results indicated that God class refactoring has a negative impact on energy consumption. Tufano et al. [161] and Delchey et al. [43] discussed when and how code smells appear in source code during development.

Android has the biggest share in the smartphone market and mobile app users prefer native mobile apps over desktop and web applications [152]. Several empirical studies have investigated the energy impact of code smells in the mobile app domain. Sahin et al. [145] applied six code refactorings from Eclipse IDE refactoring tool to the source code and evaluated their energy consumption. Morales et al. [108] analyzed the energy impact of eight types of anti-patterns in 20 open-source Android apps. Rodriguez et al. [141] evaluated the trade-off between object oriented design and battery consumption of mobile apps. Reimann et al. [137] published a catalogue of 30 quality code smells for Android platform different from Fowler et al.'s [59] list of code smells. Carette et al. [25] proposed an automated approach called HOT-PEPPER, which enables developers to detect and correct three Android-specific code smells. Cruz et al. [36] provide list of design patterns for improving the energy efficiency in mobile apps. Chowdhury et al. [31] recommend applying small changes to design pattern in combination with the bundling technique as it can significantly effect on energy consumption of mobile apps. In another study, Cruz et al. [37] present a tool to automatically refactor energy code smells in mobile apps. They also studied [39] how maintainability in Android application is impacted by changes done to improve energy

efficiency.

In related studies analyzing code smells in Android apps, the focus is on Android specific code smells related to specific mobile resources, and their correlation with performance, maintainability, and sustainability. However, traditional code smells defined by M.Fowler could appear in applications irrespective of the platform, Therefore, in this thesis, we extend previous studies by investigating the energy impact of refactoring five traditional code smell types (first individually per type and then in permutation) on native Android open source apps. We also study the effect of code smell refactorings on execution time to check if there is a correlation.

## 3.2. Third-party Libraries

There are several million apps available on Android app stores like Google Play Store which could be mined to extract information related to apps and app usage. Some studies have focused on app store analyses and extracted information about billing functionality and ad libraries [14, 63, 103, 166, 170, 172]. Other studies focused on the identification of libraries in mobile apps [18, 98, 100].

Another group of studies [18, 98, 100] have explored types and distribution of commonly used third-party libraries (such as ads, billing, and social libraries) in Android apps and how they affect performance and security. These studies highlighted that third-party libraries were frequently used in Android app development. However, they did not investigate the energy consumed by third-party libraries. Wang et al. [173] presented an algorithmic solution to model the energy minimization problem for ad prefetching in Android apps. Lee et al. [89] provided a run-time energy estimation system for ads in mobile apps. Rasmussen et al. [134] conducted a study to compare the power efficiency of various methods of blocking advertisements on an Android platform. They found many cases where ad-blocking software or methods resulted in increased power usage. In Android apps, there could be many reasons for long-running operations in the background that continuously consume resources. Such operations could cause battery drain and performance degradation. Shao et al. [148] demonstrated through experiment that sometimes such behaviour could be caused by third-party libraries used in the apps.

Choosing a third-party library is not simple, developers compare similar libraries by reading documentation or searching online. Therefore, some studies [41, 106, 107, 162] explored different online sources and repositories and ranked third-party libraries and APIs based on metrics (such as popularity, backward compatibility, release cycle, etc.) so that developers can choose the right library during development. None of the previous studies discussed the energy consumption of Android third-party HTTP libraries. There are studies [90, 92] that fo-

cused on measuring and optimizing the energy of HTTP requests in Android apps. However these studies do not focus on third-party libraries for handling HTTP requests.

In existing studies, information related to the energy impact of third-party Android libraries is limited. Library-related research for Android apps, so far, has been restricted to ad libraries, billing libraries, and social libraries, investigating how these libraries are used and identified in mobile apps and how they impact the security of Android apps and users' privacy. Other often used third-party libraries such as Android third-party HTTP libraries have received less attention.Therefore, in this thesis, we investigate the energy consumption of eight Android third-party HTTP libraries in five use cases. We checked if there is variance in the energy consumption of popular Android third-party HTTP libraries. In addition, we also checked whether there is a correlation between energy consumption and execution time of these libraries in selected use cases.

## 3.3. Tool Support for Developing Green Android Apps

Most Android projects use Java as the programming language, however, the support tools and techniques used for Java projects reviewed by previous secondary studies [58, 83, 149] cannot be effectively applied to Android projects as Android development differs from traditional software development in terms of context, user-experience and a touch-based interface. Therefore, many specialized support tools have been developed to improve the quality of Android apps with regard to maintainability, performance, security, or energy efficiency.

Studies that present support tools for developing green android apps cover different aspects. Some studies have focused on improving quality of apps by refactoring traditional code smells or by identifying various types of third-party libraries (as discussed above). Other studies have focused on developing tools and techniques to address Android specific code smells and energy bugs [25, 37, 53, 68]. Some empirical studies have investigated the energy consumption of mobile apps by developing tools and methods for energy profiling and enhancement of mobile apps [16, 20, 29, 32, 55, 82, 125]. Di Nucci et al. [46] introduced a new software based tool PETrA, for measuring the energy consumption of Android apps. Rua et al. [143] reported on GreenSource infrastructure, which characterize energy consumption in the Android ecosystem. GreenSource used AnaDroid tool to instrument app code for software-based energy measurements and to automate application execution.

The primary studies mentioned above are just a glimpse of literature presenting support tools. To get a complete picture secondary studies are quite useful. Studies that compare state-of-the-art support tools and techniques can help in selection of an appropriate support tool for a given task. However, secondary studies

related to measuring and improving energy efficiency in Android apps are scarce. Related secondary studies in the domain of Android development have focused on comparing tools and techniques related to static analysis of code, resource usage, security, malware detection, repackaging, and benchmarking of development approaches.

Li et al. [95] performed a systematic literature review to analyze static source code analysis techniques and tools proposed for Android to assess issues related to security, performance, or energy. The authors have reviewed work published between 2011 and 2015, consisting of 124 studies. The review concluded that the majority of static analysis techniques only uncover security flaws in Android apps. Qiu et al. [133] provide a comparison between three static analysis tools: FlowDroid/IccTA, Amandroid, and Droidsafe. They evaluated these tools using a common configuration setup and the same set of benchmark applications. Results were compared to the results of previous studies in order to identify reasons for inaccuracy in existing tools.

Degu A. [42] performed a systematic literature review to classify primary studies with a focus on resource usage, energy consumption, and performance in Android apps. The classification is high level based on the main research focus, type of contribution and type of evaluation method adopted in selected studies. Their results did not provide an in-depth review of support tools in green Android development. Rawassizadeh R. [136] benchmarked apps based on their resource usage. They proposed a method using a utility function and evaluated their results on two apps comparing their CPU and memory consumption.

Corrodi et al. [35] review the state-of-the-art in Android data leak detection tools. Out of 87 state-of-the-art tools, they executed five based on availability. They compared those five tools against a set of known vulnerabilities and discussed the overall performance of the tools. Ndagi et al. [112] provided a comparison of machine classifiers for detecting phishing attacks in Adware in Android apps. This study concluded that many existing machine classifiers if adequately explored could yield more accurate results for phishing detection. Cooper et al. [34] provide an overview of security threats posed by Android malware. They also survey some common defence techniques to mitigate the impact of malware apps and characteristics that are commonly found in malware apps that could enable detection techniques. Li et al. [93] provide a literature review that summarises the challenges and current solutions for detecting repackaged apps. They concluded that many existing solutions merit further research as they are tested on closed datasets and might not be as efficient or accurate as they claim to be. Roy et al. [142] provide a qualitative comparison of clone detection techniques and tools. They classify, compare, and evaluate these tools.

Oliveira et al. [118] evaluated alternative development solutions available on online sites like Rossetta and compared the energy efficiency and performance of the

most commonly used approaches to develop apps on Android: Java, JavaScript, and C/C++. Their results indicated that "JavaScript was more energy efficient in 75 percent of all benchmarks as compared to Java and C++ while performance-wise, both Java and C++, outperformed JavaScript in most of the benchmarks".

To best of our knowledge, none of the previous secondary studies provide an overview of the state of the art w.r.t to support tools available to aid development of green Android apps. Most secondary studies discussed above have covered published work until 2015 or 2017 and many of the reviewed tools in those studies are now out-dated/obsolete. To build effective Android-specific support tools to aid development of green Android apps, we first need to understand what is already available and what is still missing. Therefore, in this thesis, we provide a different view of literature by conducting a systematic mapping study to analyze recently developed support tools for energy profiling, code optimization, refactoring, and third-party library detection or migration in Android development to improve energy efficiency in apps. We explore if these support tools aid green Android development. We provide an overview of the techniques used in these support tools, information about their interface, availability, and IDE integration.

# 4. IMPACT OF CODE SMELL REFACTORING ON THE ENERGY CONSUMPTION OF ANDROID APPS

In this chapter, we focus on RG1 (defined in Section 1.1) by investigating the influence of code structure on the energy efficiency of mobile apps. Android app code can be roughly divided into two parts: the custom code and the reusable code. In order to reduce the energy consumption of mobile devices, the quality of custom code in mobile apps could be improved. This could be achieved by constant refactoring. Refactoring is "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. Code smells are an indicator of a problem in software design and quality that requires refactoring" [59]. Refactoring code smells could impact the energy consumption of mobile apps. Therefore, we investigate the energy impact of refactoring five code smell types (first individually per type and then in permutation) in custom code of native Android open source apps. We also study the effect of code smell refactorings on execution time to check if there is a correlation. We expect that our findings will help developers improve their Android apps not only with regard to maintainability but also in terms of energy efficiency.

## 4.1. Research Method

In this section we introduce the research questions, identify the code smells for refactoring, present the tools used for the detection of code smells, and define the experimental setup of our study.

### 4.1.1. Research Questions

Our research questions are the following:

**RG1-RQ1**

*Is there a correlation between code smell refactoring and energy consumption of Android apps?*

**RG1-RQ2**

*Is there a correlation between code smell refactoring and execution time of Android apps?*

RG1-RQ1 investigates the energy impact of code refactorings from two aspects 1) impact, i.e., how do code refactorings impact the energy consumption of Android

apps and 2) Consistency, i.e., are the observed effects of code smell refactorings in Android based Java applications similar to the effects observed in other Java based applications discussed in related work. RG1-RQ2 investigates the impact of code refactoring on the apps execution time, i.e., What is the correlation between execution time and energy efficiency?

### 4.1.2. Code Smells

In this study we focus on commonly occurring code smells in Java applications and investigate their energy impact in Android apps. We assume that the types of code smells identified by Fowler et al. [59] occur in Java code independent of platform, although the frequency with which they occur might be distributed differently [138]. Therefore, we picked the set of code smell types analyzed in our study, as well as their refactorings, from Fowler et al.'s list.

The selected smells are 'Long Method', 'Feature Envy', 'Type Checking', 'Duplicated Code', and 'God Class'. These smells have previously been investigated for their energy impact (cf. Chapter III) but the results from those studies are limited by the type and number of apps on which these code smells were analyzed.

The **'Long Method'** code smell is one of the most commonly occurring smells in Android apps. As defined by Fowler [59], this smell points to a method which has become too long over time and could affect the maintainability of the app. Several refactoring techniques are defined by Fowler to get rid of this smell. In our study, we apply the 'Extract Method' refactoring for this smell. As a result of this refactoring, the 'Long method' is divided into smaller methods which are called inside the original method.

The **'Feature Envy'** code smell occurs when one class envies the features of another class, i.e., a method in one class sends many 'get method' calls to another class, which indicates that this method could be placed inside the other class whose features it envies [59]. We apply the 'Move method' refactoring for this code smell.

The **'Type Checking'** code smell occurs when an attribute in a class representing the state is checked for different values. This state attribute is referred to as a type field. The term 'Type Checking' was used by the creator of JDeodorant for this smell [59, 159]. In our study, we use the refactoring 'replace type code with state/strategy'. This refactoring results in new methods and subclasses.

The **'Duplicated Code'** smell occurs when the same code is used in many places inside an app. This could be intentional or it could be due to the copying of the code or it could be the result of another refactoring. It could be refactored by combining the different code clone structures into one [59].

The **'God Class'** code smell occurs when over time many functionalities are

added to the same class in a project, making it responsible for a large percentage of the app's architecture. This smell is removed by dividing the large class into smaller classes, each having a unique functionality [59].

### 4.1.3. Code Smell Detection Tool and Refactoring

Based on previous studies [57, 121, 160, 164] that compared different code smell detection and refactoring tools, we chose to use JDeodorant for code smell detection. JDeodorant follows all the refactoring activities stated by Mens et al. [105]. According to the studies mentioned above JDeodorant is very effective in detecting the selected code smells. We manually evaluated each of the refactorings suggested by JDeodorant before applying them to ensure that they do not change the external behavior of the app.

### 4.1.4. Selected Apps

We chose open source apps in Android for this study because Android holds the biggest market share in the world's smartphone market. We selected from F-Droid[1] open source native Android apps that were more than two years old, had more than 10K downloads, and at least a rating of '4' in the Google play store. The selected apps[2][3][4] were picked from the categories 'Tools' and 'Puzzles'. The detection of code smells and the generation of refactoring candidates was done using JDeodorant with Eclipse IDE. Table 4.1 summarizes the characteristics of each of the selected apps. The first row (Years) shows the age of the project (up till February 2019) for example, the age of 'Calculator' app is 6 years and 8 months. The second row (LOC) shows the source line of codes and gives an estimate of the size of the project. The third row (Downloads) shows the number of times the selected app is downloaded by users from the Google play store. The last row (Ratings) shows the mean of ratings given to the app by Google play store users.

Table 4.1: Characteristics of selected Android apps

| | Calculator | Todo-List | Openflood |
|---|---|---|---|
| years | 6.8 | 3 | 3.1 |
| LOC | 7758 | 6145 | 1236 |
| Downloads | 1,000k+ | 10k+ | 10k+ |
| Ratings | 4.5 | 4 | 4.6 |

### 4.1.5. Testing Tool

We used Espresso to create test scripts as it comes built-in with Android Studio and, according to a recent studies [38, 99], is very fast and reliable outperforming

[1] https://www.f-droid.org/
[2] https://f-droid.org/en/packages/com.xlythe.calculator.material/
[3] https://f-droid.org/packages/org.secuso.privacyfriendlytodolist/
[4] https://f-droid.org/en/packages/com.gunshippenguin.openflood/

other tools for testing native Android apps. Since we do not intend to navigate outside the app under test, Espresso is a good choice as it supports white box testing and UI tests can be created easily. The test scripts created in Espresso do not need time delays to function properly. Therefore, the overhead introduced in the execution of the apps when using Espresso is low.

We used our test scripts for energy measurements and also to ensure the correctness of the refactored app code. The test cases[5] included in the test scripts may seem trivial but they were solely defined to ensure that the code containing a smell is actually executed. To validate that the test scripts triggered the execution of code containing a code smell as well as the execution of the corresponding refactored code, execution traces were inspected.

#### 4.1.6. Energy Measurement

We used hardware based approach for energy measurement. We used Monsoon power monitor to measure the energy consumption, recording the power measurements at a rate of 5KHz. The time between two readings was 0.0002 seconds. In this study, energy consumption corresponds to the total amount of energy used by the mobile device within a period of time. Energy is measured in Joules which is power (watts) times measurement period (seconds). We calculated the energy associated with each reading as E = Power x (0.0002). The total energy consumption corresponds to the sum of the energy associated with each reading.

Before starting the experiment, a baseline was recorded to measure the energy consumption of the mobile device in an idle state, which was then subtracted from the actual energy readings during the experiment to filter out the energy used by the app under test. During the experiment, the screen brightness was set to minimum and only essential Android services were run on the phone.

The timestamps from the adb logs recording during energy measurement and the CSV files containing energy readings were matched to mark the start and end of an execution run. Each app version was executed 10 times to account for underlying variation in the mobile device.

#### 4.1.7. Test Environment

The test environment consisted of an HP Elite Book, the Monsoon power monitor, and an LG Spirit Y70 Phone having Android 5.0.1 as the operating system with 1GB of RAM, and a 2100mAh battery. The test was controlled from the HP Elite Book using a script that automated the process and runs each app version 10 times. This saved manual effort and ensured that no problems were created during the experiment due to human error. The mobile device was connected via USB cable

[5] Calculator app test cases: https://bitbucket.org/hinaanwar2003/calculator/wiki/Test%20Cases. Todo-List app test cases: https://bitbucket.org/hinaanwar2003/todolist/wiki/Test%20Cases. Openflood app test cases: https://bitbucket.org/hinaanwar2003/open_flood_src/wiki/Test%20Cases

to the Monsoon power monitor according to the instruction manual of the power monitor [12].

#### 4.1.8. Experimental Design

In this section, we describe the setup of our experiment to measure the energy consumption of code smell refactorings on Android apps. JDeodorant is provided as a plugin in Eclipse IDE and works only with Java projects. Since Android studio projects cannot be opened directly inside Eclipse like Java projects, the apps were first modified using an Eclipse plugin in Android studio generating a file structure that helps Eclipse IDE to open the project with Gradle and recognize the Java files. Build path dependencies for projects opened in Eclipse IDE were solved manually by the author. Code smell refactorings were applied in two ways: 1) code smell refactorings per code smell type and 2) code smell refactorings for all code smell types (one type after the other) in various permutations. For single code smell type, we first detected the smells using JDeodorant and then candidate refactorings suggested by JDeodorant were applied. For each suggested refactoring, the test script was executed to ensure that the refactoring could be applied without altering the functionality of the app. If the test script failed for a suggested refactoring, that refactoring was ignored. We applied refactorings for a single code smell type and made new versions of the refactored app. For assessing whether the order of refactoring has an effect, we also made versions in which all code smell types were refactored in various permutations grouped by type. As the total number of all possible permutations of five code smell types was 120, we chose a sample of permutations randomly using Fisher-Yates shuffle [56]. Fisher-Yates shuffle was selected as it is relatively simple and unbiased i.e., the probability occurrence for every permutation is equal. For versions of the app where refactorings of all code smell types were applied in various permutations, we ignored any new candidate refactorings identified as a result of applying refactorings to a previous code smell type. For most code smell types, more refactorings could be applied but we only applied and reported the refactorings for which the test script was successfully executed.

Tables 4.2, 4.3, and 4.4 show the numbers of applied code smell refactorings per app. The code smell names are abbreviated as LM (Long Method), FE (Feature Envy), TC (Type Checking), DC (Duplicated Code), and GC (God Class). In the first column 'Refactoring', the row 'Single Refactoring' refers to the situation where only code refactorings of one code smell type were applied. The rest of the rows in the first column represent the situation where refactorings of all code smell types were applied one by one in various permutations. For statistical analysis, normality and homogeneity of the energy data were checked before doing the analysis of variance[6][7] (ANOVA) related to RG1-RQ1 and RG1-RQ2. For our

[6]The *aov()* function is R is used to perform analysis of variance.

[7]The data used in the analysis is available at https://figshare.com/s/53e2d06d8ab9c0d78427

analyses, alpha was set to 0.05. We calculated the effect size using Cliff's delta [33].

Table 4.2: Number of refactorings for Calculator app

| Refactoring | LM | FE | TC | DC | GC |
|---|---|---|---|---|---|
| Single Refactoring | 41 | 1 | 3 | 3 | 6 |
| LM-TC-GC-FE-DC | 41 | 1 | 2 | 2 | 5 |
| FE-LM-TC-DC-GC | 41 | 1 | 2 | 2 | 5 |
| TC-GC-LM-FE-DC | 39 | 1 | 3 | 2 | 5 |
| GC-LM-DC-TC-FE | 41 | 1 | 2 | 2 | 6 |
| FE-DC-TC-GC-LM | 40 | 1 | 3 | 3 | 5 |
| TC-LM-DC-FE-GC | 39 | 1 | 3 | 2 | 5 |
| LM-DC-TC-GC-FE | 41 | 1 | 2 | 2 | 5 |
| LM-TC-DC-FE-GC | 41 | 1 | 2 | 2 | 5 |
| DC-TC-FE-LM-GC | 40 | 1 | 3 | 3 | 5 |

Table 4.3: Number of refactorings for Todo-List app

| Refactoring | LM | FE | TC | DC | GC |
|---|---|---|---|---|---|
| Single Refactoring | 45 | 3 | 3 | 4 | 11 |
| LM-TC-GC-FE-DC | 45 | 1 | 3 | 2 | 6 |
| FE-LM-TC-DC-GC | 44 | 3 | 3 | 2 | 6 |
| TC-GC-LM-FE-DC | 41 | 1 | 3 | 2 | 11 |
| GC-LM-DC-TC-FE | 43 | 1 | 3 | 2 | 11 |
| FE-DC-TC-GC-LM | 36 | 3 | 3 | 4 | 11 |
| TC-LM-DC-FE-GC | 45 | 1 | 3 | 2 | 6 |
| LM-DC-TC-GC-FE | 45 | 1 | 3 | 2 | 6 |
| LM-TC-DC-FE-GC | 45 | 1 | 3 | 2 | 6 |
| DC-TC-FE-LM-GC | 40 | 3 | 3 | 4 | 6 |

Table 4.4: Number of refactorings for Openflood app

| Refactoring | LM | FE | TC | DC | GC |
|---|---|---|---|---|---|
| Single Refactoring | 7 | 1 | 0 | 0 | 1 |
| LM-GC-FE | 7 | 0 | 0 | 0 | 1 |
| FE-LM-GC | 7 | 1 | 0 | 0 | 1 |
| GC-LM-FE | 6 | 0 | 0 | 0 | 1 |
| FE-GC-LM | 6 | 1 | 0 | 0 | 1 |
| LM-FE-GC | 7 | 1 | 0 | 0 | 1 |
| GC-FE-LM | 6 | 0 | 0 | 0 | 1 |

## 4.2. Results

In this section, we present the results of our investigation regarding the impact of code refactoring on energy consumption and execution time for Android apps.

### 4.2.1. RG1-RQ1: *Is there a correlation between code smell refactoring and energy consumption of Android apps?*

An analysis of variance (ANOVA) showed that the effect of code smell refactorings on energy consumption of Android apps was significant in two out of three apps. The null hypothesis related to RG1-RQ1 states that the energy consumption between the original version and all possible refactored versions of the app remains the same. Based on the p-values and F-values (cf. Table 4.5), the null hypothesis could be rejected for the 'Calculator' and 'Todo-List' apps (p-values below aplha=0.05 and large F-values) but not for 'Openflood'.

Figures 4.1, 4.2, and 4.3 show the boxplots of Treatment vs. Energy consumption in joules for each of the apps. The treatments are along the x-axis while energy consumption is shown along the y-axis. Treatments refer to the type of code smell refactoring applied (or=Original, LM=Long Method, FE=Feature Envy, TC= Type Checking, GC=God Class, DC=Duplicated Code, ALL=Avg. of all versions where the refactorings of all code smell types were applied in permutations). From figures 4.1, 4.2, and 4.3 we see that for permutations of code smell refactorings, for all apps, there was no significant difference in energy consumption. For individual code smell refactorings, two out of three apps had a significant difference in energy consumption, but the direction of the effect is not uniform across treatments. Only TC, DC, FE, and LM consistently saving energy in both apps, however, the strength of the effect is not uniform. Table 4.6 gives details about the median, standard deviation, percentage change, and effect size for each app after each treatment. In Table 4.6, the negative percentage change means a reduction in energy consumption, and the positive percentage change means an increase in energy consumption. We see that for the 'Calculator' app maximum reduction in energy consumption was recorded in app versions where 'Duplicated code' (10.8%) and 'Type checking' (10.5%) code smell refactorings were applied.

Table 4.5: ANOVA results for all apps (energy consumption)

| Application | p-Value | F-value |
|---|---|---|
| Calculator | 0.00005343 | 7.3503 |
| Todo-List | 0.0009859 | 4.3473 |
| Openflood | 0.4351 | 0.9666 |

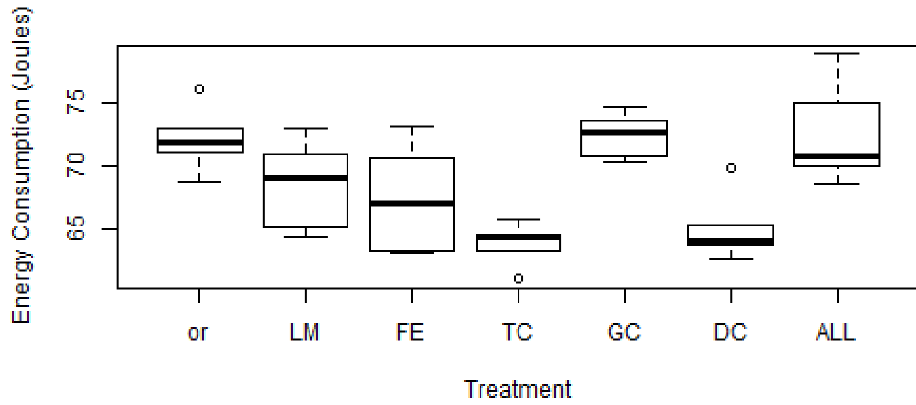


Figure 4.1: Energy consumption in joules for Calculator app per treatment

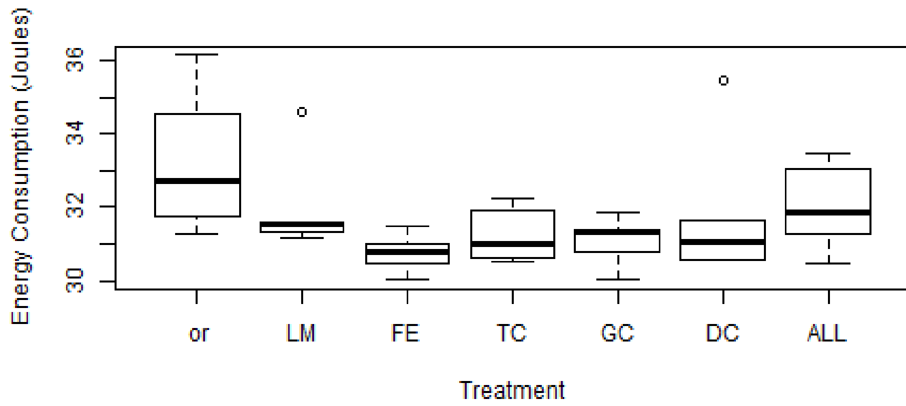


Figure 4.2: Energy consumption in joules for Todo-List app per treatment

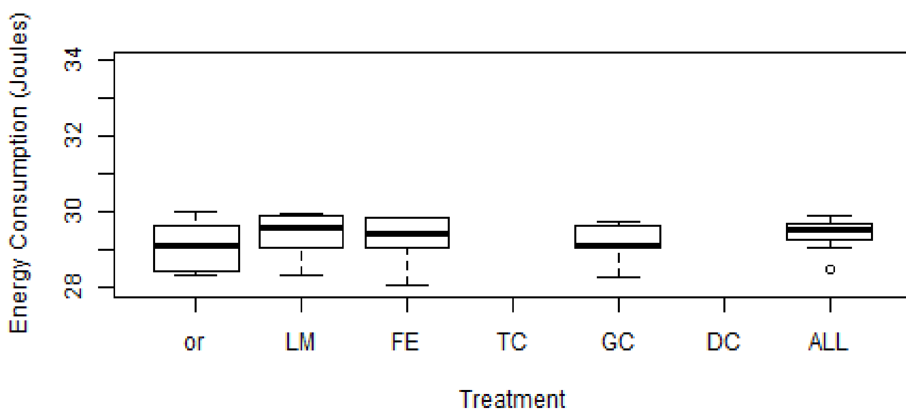


Figure 4.3: Energy consumption in joules for 'Openflood' app per treatment

Table 4.6: Overview of energy consumption results (RG1-RQ1)

| Application (joules) | | Treatments | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Or | LM | FE | TC | GC | DC | ALL |
| Calculator | Median | 71.9 | 69.1 | 67.1 | 64.3 | 72.6 | 64.1 | 71.6 |
| | SD | 2.6 | 3.6 | 4.4 | 1.7 | 1.8 | 2.8 | 3.4 |
| | % | - | -3.8 | -6.6 | -10.5 | +1.0 | -10.8 | -1.3 |
| | ES | - | L | L | L | N | L | S |
| Todo-List | Median | 32.7 | 31.5 | 30.8 | 31.0 | 31.3 | 31.0 | 31.8 |
| | SD | 2.3 | 1.4 | 0.5 | 0.7 | 0.7 | 2.0 | 1.0 |
| | % | - | -3.6 | -5.9 | -5.2 | -4.4 | -5.1 | -2.6 |
| | ES | - | M | L | L | L | L | M |
| Openflood | Median | 29.1 | 29.5 | 29.4 | N/A | 29.1 | N/A | 29.5 |
| | SD | 0.6 | 0.5 | 0.5 | - | 0.4 | - | 0.4 |
| | % | - | +1.5 | +1.1 | N/A | -0.02 | N/A | +1.3 |
| | ES | - | S | S | N/A | N | N/A | S |

(SD=standard deviation, %= percentage change, ES=effect size, L=large, M= medium, S=small, N=negligible)

### 4.2.2. RG1-RQ2: *Is there a correlation between code smell refactoring and execution time of Android apps?*

Based on our measurement we could not find a significant impact of code refactoring on execution time for the selected Android apps. The p-values (cf. Table 4.7) were 0.2584, 0.113, and 0.384 for the ‘Calculator’, ‘Todo-List’, and ‘Openflood’ apps respectively, which was greater than the alpha value of 0.05. Therefore, the null hypothesis stating that the execution time between the original version and all possible refactored versions of the app remains the same could not be rejected.

Figures 4.4, 4.5, and 4.6 show the boxplots of Treatment vs. Execution time in seconds for each of the apps. The treatments are along the x-axis while execution time is shown along the y-axis. Treatments refer to the type of code smell refactoring applied (or=Original, LM=Long Method, FE=Feature Envy, TC=Type Checking, GC=God Class, DC=Duplicated Code, ALL=Avg. of all versions where the refactorings of all code smell types were applied in permutations). From figures 4.4, 4.5, and 4.6 we see that that for permutations of code smell refactorings, for all apps, there was no significant difference in execution time. For individual code smell refactorings, only LM and FE show reduction in execution time in two out of three apps, but the strength of the effect is not uniform.

Table 4.8 gives details about the median, standard deviation, percentage change, and effect size for each app after each treatment. In Table 4.8, the negative percentage change means a reduction in execution time, and the positive percentage change means an increase in execution time. In ’Openflood’ no significant change

in execution time was recorded. For ’Todo-List’ app, for all versions execution time reduced but effect size was not consistent. In ’Todo-List’ app maximum reduction in execution time was recorded where ‘Feature Envy’ (1.26%) and ’God Class’ (1.25%) code smell refactorings were applied. For ’Calculator’ app, version where ’Long Method’ and ’Feature Envy’ code smell refactoring were applied execution time reduced but effect size was negligible. For versions were ’Type checking’, ’God Class’, ’Duplicated Code’ and all permutations of code smell refactorings were applied, execution time was increased however effect size was not uniform.

Table 4.7: ANOVA results for all apps (execution time)

| Application | p-Value | F-value |
|---|---|---|
| Calculator | 0.258 | 1.365 |
| Todo-List | 0.113 | 1.902 |
| Openflood | 0.384 | 1.067 |

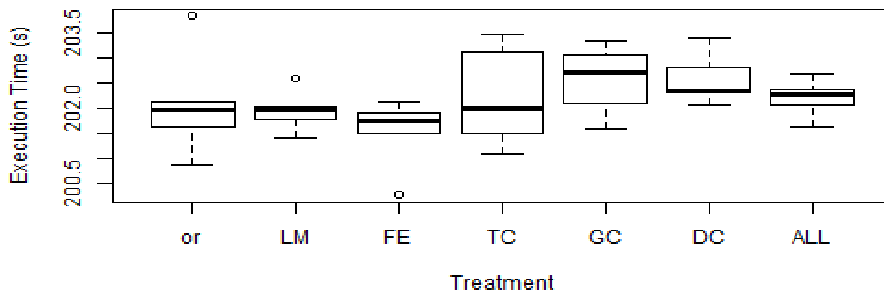


Figure 4.4: Execution time in seconds for Calculator app per treatment

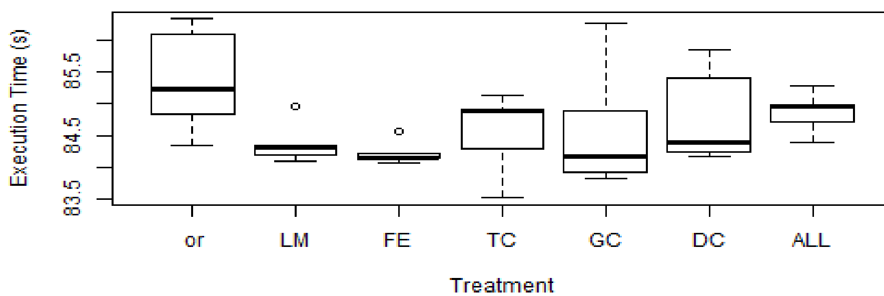


Figure 4.5: Execution time in seconds for Todo-List app per treatment

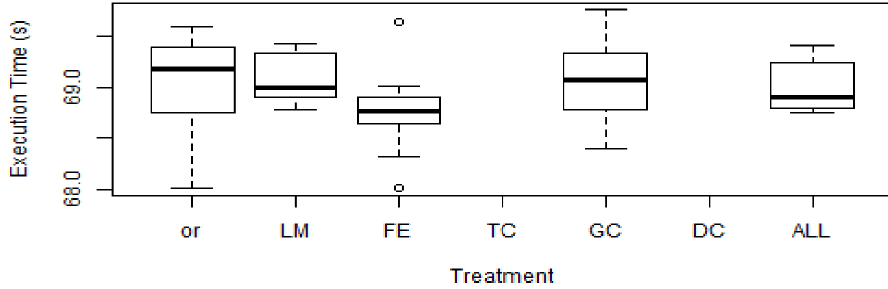


Figure 4.6: Execution time in seconds for 'Openflood' app per treatment

Table 4.8: Overview of execution time results (RG1-RQ2)

| Application (seconds) | | Treatments Or | LM | FE | TC | GC | DC | ALL |
|---|---|---|---|---|---|---|---|---|
| Calculator | Median | 201.9 | 201.9 | 201.7 | 202 | 202.7 | 202.3 | 202.3 |
| | SD | 1.09 | 0.43 | 0.73 | 1.03 | 0.71 | 0.53 | 0.34 |
| | % | - | -0.005 | -0.1 | +0.01 | +0.36 | +0.17 | +0.15 |
| | ES | - | N | M | N | S | L | M |
| Todo-List | Median | 84.3 | 84.1 | 84.8 | 84.1 | 84.3 | 84.9 | 85.2 |
| | SD | 0.33 | 0.19 | 0.65 | 1.0 | 0.76 | 0.27 | 0.84 |
| | % | - | -1.08 | -1.26 | -0.40 | -1.25 | -0.98 | -0.33 |
| | ES | - | L | L | L | L | M | M |
| Openflood | Median | 69.1 | 68.9 | 68.7 | N/A | 69 | N/A | 68.9 |
| | SD | 0.55 | 0.22 | 0.43 | - | 0.43 | - | 0.24 |
| | % | - | 0 | -0.37 | N/A | +0.47 | N/A | -0.09 |
| | ES | - | S | M | N/A | N | N/A | N |

(SD=standard deviation, %= percentage change, ES=effect size, L=large, M= medium, S=small, N=negligible)

## 4.3. Discussion

In this section, we take a closer look at how different configurations of code smell refactorings affected the energy consumption of each of the analyzed Android apps. Verdecchia et al. [163] reported maximum energy reduction in versions where 'Long Method' and 'Feature Envy' code smell refactorings were applied. In our experiment, the number of applied refactoring of both 'Type Checking' and 'Duplicated Code' code smells were smaller as compared to 'Long Method' code smell refactorings but the recorded effect size for energy reduction was larger. For app versions where 'Long Method' code smell refactorings were applied our results were similar to Verdecchia et al. [163].

Table 4.9 shows the median, standard deviation, percentage change, and effect size when the refactorings of all code smell types were applied in permutations. The negative percentage change indicates a reduction in energy consumption, and the positive percentage change indicates an increase in energy consumption. We only analysed the 'Calculator' and 'Todo-List' app because for 'Openflood' app the change in energy consumption was not significant. In the 'Calculator' and 'Todo-List' apps. In 'Calculator' app the permutation 'LM-TC-DC-FE-GC' resulted in a maximum increase of energy consumption up to 7.25%, while the permutation 'DC-TC-FE-LM-GC' resulted in a maximum decrease of energy consumption up to 9.3%. In the 'Todo-List' app no permutation resulted in a significant increase in energy consumption, However, the maximum decrease of energy consumption was in permutation 'FE-DC-TC-GC-LM' (up to 7%). If not used carefully the refactorings for different code smells could cancel out each other's positive effects. To find a clear pattern between a permutation of code smell refactoring types and energy consumption, we need experiments with more permutations on a bigger corpus of Android apps.

Table 4.9: Energy consumption results when a permutation of code smell refactorings was applied.

| Refactoring | Calculator | | | | Todo-List | | | |
|---|---|---|---|---|---|---|---|---|
| | Med | SD | % | ES | Med | SD | % | ES |
| Original (baseline) | 71.9 | 2.6 | - | - | 32.7 | 2.0 | - | - |
| LM-TC-GC-FE-DC | 74.8 | 6.3 | +4.0 | N | 31.3 | 0.9 | -4.3 | L |
| FE-LM-TC-DC-GC | 66.8 | 2.8 | -7.0 | L | 31.4 | 0.8 | -3.9 | M |
| TC-GC-LM-FE-DC | 66.8 | 6.9 | -7.1 | S | 31.3 | 0.9 | -4.1 | L |
| GC-LM-DC-TC-FE | 75.8 | 6.9 | +5.4 | S | 32.3 | 2.3 | -1.1 | N |
| FE-DC-TC-GC-LM | 76.6 | 6.5 | +6.9 | L | 30.4 | 0.5 | -7.0 | L |
| TC-LM-DC-FE-GC | 70.9 | 2.5 | -1.3 | S | 33.8 | 0.7 | +3.4 | N |
| LM-DC-TC-GC-FE | 69.4 | 4.8 | -3.4 | S | 32.2 | 1.5 | -1.3 | N |
| LM-TC-DC-FE-GC | 77.1 | 4.8 | +7.2 | L | 30.9 | 0.5 | -5.3 | L |
| DC-TC-FE-LM-GC | 65.1 | 9.6 | -9.3 | L | 31.9 | 1.2 | -2.4 | S |

(Med=median, SD= standard deviation, %= percentage change, ES=effect size, L=large, M= medium, S=small, N=negligible)

The results related to RG1-RQ2 are contradictory to the results reported by Verdecchia et al. [163]. They reported that observed energy reduction was due to performance-related improvements. However, in our results the energy consumption was not clearly related to increase or decrease in execution time. For Calculator app, energy consumption reduced in versions where 'Long Method', 'Feature Envy', 'Type checking' and 'Duplicated' code smell refactorings were applied. However, for the same versions execution time was reduced (with negli-

gible effect size) only in versions where 'Long Method' and 'Feature Envy' code smell refactorings were applied. For all other versions execution time increased and the effect size was not uniform. For 'Todo-List' app, energy consumption and execution reduced in all versions, however, effect size was not uniform. For 'Openflood' app, there was negligible change is execution time and energy consumption. The direction of the effect was not uniform across treatments. Therefore, more experimental evidence is required to confirm the assumption that a trade-off between execution time and energy consumption exists.

We detailed the statistics of the selected Android apps in section 4.1.4. The age of apps 'Todo-List' and 'Openflood' were the same yet there was a significant difference in the number of code smells refactored in these two apps. This shows that for the Android app of the same age the probability of detected code smells could be dependent on the size of the projects and also possibly on the experience and knowledge of the contributors of that project regarding code smell refactoring.

**Insight 1**

*Impact of refactoring only a single type of code smell on energy consumption of selected apps was not consistent. However, in two out of three selected apps, where the effect size was medium or large, the energy consumption decreased due to refactoring.*

**Insight 2**

*The energy impact of overall permutations of code smell refactorings in the selected Android apps was small. However, specific permutations of code smell refactorings should be used with caution as their energy impact might vary strongly depending on the selected Android app.*

**Insight 3**

*Significant reduction in energy consumption of Android apps does not necessarily correlate with a significant reduction or increase of execution time.*

## 4.4. Threats to Validity

In this section, we discuss the possible threats to validity of this study and our strategy to mitigate them.

***Internal validity:*** In order to avoid threats related to the order in which the tests were executed, caches and data related to each app was cleaned from the mobile device before each run. When applying multiple treatments, the order of treat-

ments could effect the results. To mitigate effects that might occur due to specific order of refactoring we performed randomization using Fisher-Yates shuffle [56]. We used JDeodorant for detecting code smells and for generating candidate refactorings. The accuracy of this tool might affect the accuracy of the results. We selected JDeodorant because it complies with Mens et al. [105] list of activities that should be followed by a good detection and refactoring tool. JDeodorant has been used in various previous studies for code smell detection, based on the above two reasons we think that the results produced by JDeodorant were reliable. However, we cannot ensure that the same candidate code smell refactorings will be produced if this study is replicated using another code smell detection tool.

***External validity:*** The apps chosen for experimentation were selected from F-Droid but to ensure that they were representative of a real world Android apps we checked that these apps are also available on Google play store and have more than 10K downloads and a rating of 4 at least. We only analyzed three apps therefore the results might not be generalizable in every context. The current results are specific to context (in terms of OS, programming language and type of applications) used in the experiment. Increasing the magnitude of study in terms of a number of included apps in future work is desirable. Increasing the number of apps in the experiment also exponentially increases the number of app versions that need to be tested. As writing test scenarios, checking each candidate refactoring before applying and mapping of Android projects to Eclipse Java projects is a time consuming and slow process that requires manual effort, we limited this experiment to three apps. Nevertheless, we did consider apps from two different app categories of Google play store.

It is important to note that in order to replicate the results of this experiment same phone model and Android OS version, measurement tools and app version should be used. The reason is that different mobile phones with different Android OS might use different levels of energy. Changing the version of the apps might result in failure of test cases. To ensure that the selected device was a good representation of an Android device we checked the device specifications, i.e., Android version, Chipset, screen resolution and size, battery capacity and RAM size.

***Construct validity:*** Accuracy of energy measurements depend on the sampling frequency of the measurement device. Energy measurements were recorded using the Monsoon power monitor which recorded power consumption at a rate of 5KHz or one sample every 200 microsecond. The same specification has been used in previous studies [20, 38, 81, 91, 180] so sampling frequency in enough to produce precise readings. Energy measurements were for app plus Android OS related activities. We tried to minimize this threat by using adb logs and matching the timestamps with the energy trace to filter out any unwanted readings. A baseline was recorded and subtracted from readings. We repeated the tests 10 times for each version and included the averaged reading to further mitigate variations in

the readings. However, recording the logs for the app introduces an overhead which in itself could be energy consuming. But as this overhead was constant between the versions of the app it could be ignored. During the experiments, the screen brightness is set to a minimum and only essential Android services are run on the phone.

# 5. IMPACT OF THIRD-PARTY HTTP LIBRARIES ON THE ENERGY CONSUMPTION OF ANDROID APPS

In this chapter, we also focus on RG1 (defined in Section 1.1) by investigating the influence of code structure on the energy efficiency of mobile apps. Android app code can be roughly divided into two parts: the custom code and the reusable code. In previous chapter we conducted a study on the custom code of Android apps. In this chapter we focus on the reusable code i.e., third-party libraries included in the Android app code.

The use of third party libraries for implementing various functionalities in mobile apps is very common. Wang et al. [168] have found that over 60% of Android apps' code is contributed by commonly used libraries. Developers prefer to use libraries because it speeds up the development process. Usually, for a specific task, there are many different libraries available that can help developers achieve that task. If developers are not well informed, then the task of choosing the correct library may not be easy. Developers compare libraries based on different characteristics such as performance, usability, community support or functionality of the library to find the best alternative. The information available on sites such as stack overflow is usually limited as each online post focuses on a specific aspect of the library and typically provided information is based on the personal experience of the posting developers. Developers can also consult online catalogs such as 'Awesome Android', 'AppBrain', and 'MindOrk'[1] which give information about a library's popularity, usage, number of installs, etc. (information usually extracted from the official documentation and Github pages of the libraries.), however, none of these sites provide information about the energy consumption of Android third-party libraries. A library once added in the app is hardly ever updated [147] and, therefore, it is crucial that developers know how much energy a particular library consumes for a particular task to avoid unnecessary battery drain. Measuring the energy consumption of these libraries could be beneficial to the developers as they could make better choices about which libraries to use in their apps.

In this chapter, we investigate Android third-party network libraries (sometimes also referred to as network packages). Among the network libraries, we focus on libraries used to establish network connections for sending and receiving data to/from the server. Apps in different categories (such as games, business, utilities, education, lifestyle, travel) have different requirements and functionalities. Moreover, user expectations are also different for each category. However, there exist features that are found in apps across categories. For instance, user account

[1] https://android.libhunt.com/
https://www.appbrain.com/
https://mindorks.com/android/store

authentication, web search, push notifications, image/video/user-data upload to servers, etc. The most common mechanism used behind these features is HTTP request methods like GET and POST. Since different third-party libraries adopt different mechanisms to achieve the same task, the energy consumption of these libraries could be different. The results in [157] indicate that there is a difference in the energy consumption of different libraries when making HTTP POST requests. Moreover, the execution time for the same task varies among libraries. Therefore we select popular Android network libraries (that could be used as an alternative to one another) offering capabilities to handle HTTP GET and POST request methods. In this chapter, we refer to them as third-party HTTP libraries. The selected libraries have the highest relative popularity rating in their respective categories in online catalogs like 'Awesome Android', 'MindOrks', and 'AppBrain'. We measure the energy consumption of eight third-party HTTP libraries in five use cases. In each use case, we created custom app versions using the selected libraries. For some use cases, additional app versions were made due to multiple implementation choices. In total, we made 45 app versions and used them for energy and execution time measurements. Results from this study will help software developers to make an informed decision when choosing popular Android third-party HTTP libraries.

## 5.1. Research Method

In this section, we introduce the research questions, present the use cases, introduce the selected third-party HTTP libraries, and define the experimental design and test environment of our study.

### 5.1.1. Research Questions

RG1-RQ3

*Is there variance in the energy consumption of Android third-party HTTP libraries?*
***RG1-RQ3-A:*** *When making GET requests is there variance in energy consumption of Android third-party HTTP libraries?*
***RG1-RQ3-B*** *When making multi-part POST requests is there variance in energy consumption of Android third-party HTTP libraries?*
***RG1-RQ3-C:*** *When sending structured JSON objects is there variance in energy consumption of Android third-party HTTP libraries?*
***RG1-RQ3-D:*** *When receiving structured JSON objects is there variance in energy consumption of Android third-party HTTP libraries?*
***RG1-RQ3-E:*** *When loading and displaying images on the screen is there variance in energy consumption of Android third-party HTTP libraries?*

RG1-RQ4

*Is the energy consumption of a third-party HTTP library correlated with execution time?*

RG1-RQ3 investigates whether there is variance in energy consumption of the selected Android third-party HTTP libraries and within each use case which selected library is most energy efficient. RG1-RQ4 deals with the execution time of each library in order to complete each task. We compare this data with energy data to see if selected third-party HTTP libraries are also efficient in terms of execution time.

### 5.1.2. Use Cases

In order for an app to get information, usually, it needs to connect to a remote server and transfer information via HTTP protocol. It is common for different platforms to provide an HTTP based API so that apps can easily integrate with them. For good app design, HTTP requests should not be made in the UI thread. As additional code is required to wrap such calls in a separate thread, therefore, developers prefer to use third-party libraries that handle HTTP requests for them.

**Motivational examples:** YouTube is one of the most popular mobile apps by Google. It is a platform where users upload and watch various types of video-based content. For unregistered users, one basic feature of this app is the topic-based search for videos. These features fetch videos or playlists based on the text phrase entered by the user. HTTP GET request is used behind this feature, which gets information from the remote server as per user request. On the other hand registered users have access to additional features like upload videos and comment on videos. To upload videos, create playlists, create subscriptions, create users, etc. HTTP POST requests are used. More details about YouTube features and API can be found in [3]. YouTube also provides analytics to its users like most viewed videos, the number of channel subscribers, number of user subscriptions, etc. HTTP GET request methods are used behind the scenes to get this information from the server. Similarly, in Google Chrome, HTTP GET request and JSON object serialization is used to get and present search results to the user. Each of the above discussed apps has more than 5 billion downloads and there are thousands of similar apps by other developers in play store which offer similar features.

According to [10], in 2018, 52.2% of all internet traffic was from mobile devices and most of this traffic is generated by a handful of top apps such as YouTube and Facebook. HTTP requests are a major part of mobile internet traffic [51]. Among HTTP requests, GET and POST are the most common ones. There are other HTTP request methods[2] as well like DELETE, PUT, OPTIONS, HEAD,

[2] See Annex A for more details about other HTTP request methods.

and PATCH but their usage frequency is very low due to security risks.

The use cases we have selected for our experiment are motivated from the above examples and are based on common features of some most downloaded apps and browsers (like YouTube, Facebook, chrome, etc.).

- **UC-GF:** Making an HTTP GET request to the server to download a file and logging the server's response. *(RG1-RQ3-A)*
- **UC-PF:** Making a multipart HTTP POST request to the server to upload a file and logging the server's response. *(RG1-RQ3-B)*
- **UC-PJO:** Making a multipart HTTP POST request for sending Java objects serialized as JSON to the server and logging the server's response. *(RG1-RQ3-C)*
- **UC-GJO:** Making an HTTP GET request to the server for receiving JSON Objects, de-serializing them and mapping them to the Java objects. *(RG1-RQ3-D)*
- **UC-GI:** Making a GET request to the server to load images and displaying them on screen. *(RG1-RQ3-E)*

Many other use cases can be derived based on the common features of the apps like Databases, image processing, UI capabilities, e-commerce, etc. but, as a start, we looked only at the above selected five use cases related to HTTP requests.

### 5.1.3. Selected Libraries

The Android third-party HTTP libraries usually can be found under the 'network' category in online catalogs like 'Awesome Android', 'AppBrain', and 'MindOrks'.

'MindOrks' do not provide any additional statistics about third party libraries. Therefore, we gathered the statistics[3] about Android third-party libraries from online catalogs 'Awesome Android' and 'AppBrain' only. Based on the extracted information, we chose the top five libraries in the 'network' category based on popularity, number of watchers, and quality ranking. Upon closer look, we found that only two of these libraries also provided an additional capability of image loading. Therefore we selected the top three Android third-party libraries from the 'image loading' category that could be used as an alternative in UC-GI. The Android third-party libraries that perform only image loading are usually listed under the category of 'image loading' in online catalogs. The selected libraries[4] are listed below.

[3] Annex A, See Table A.1 and Table A.2.

[4] Annex A, detail features of each selected library can be seen in section A.2.

***Volley (Version 1.1.1)***. This is an HTTP library that excels at RPC-type operations used to populate a UI, such as fetching a page of search results as structured data. It offers support for raw strings, images, and JSON [45].

***Retrofit (Version 2.5.0)***. This is a REST Client for Android and Java. It makes it relatively easy to retrieve and upload JSON (or other structured data) via a REST-based web service [67, 78].

***OkHttp (Version 3.13.1)***. This is an open-source project designed to be an efficient HTTP client. It supports the SPDY protocol. SPDY is the basis for HTTP 2.0 and allows multiple HTTP requests to be multiplexed over one socket connection [66, 151].

***Androidasynchttp (version 1.4.9)***. This is an asynchronous callback-based HTTP client for Android built on top of Apache's HTTP Client libraries [1].

***Androidasync (version 2.2.1)***. This is a low level network protocol library used mostly for a raw socket, HTTP(s) client/server, and web socket [49].

***Picasso (Version 2.7)***. This is an image library for Android. It is created and maintained by Square and caters to image loading and processing. It simplifies the process of displaying images from external locations [2, 8].

***Universal Image Loader (UIL) (Version 1.9.5)***. This library aims to provide a powerful, flexible, and highly customizable instrument for image loading, caching, and displaying. It provides a lot of configuration options and good control over the image loading and caching process [7].

***Glide (Version 4.9.0)***. This is a fast and efficient open-source media management and image loading framework for Android that wraps media decoding, memory and disk caching, and resource pooling into a simple and easy to use interface [6].

### 5.1.4. Experimental Design

In this section, we describe the setup of our experiment to measure the energy consumption of selected third-party HTTP libraries used in Android apps. For this experiment, we made app versions[5], depending upon the use cases, as shown in Figure 5.1 below.

For UC-GF and UC-PF, we made app versions with a blank screen that performs the HTTP request and logs the server's response. In each app version, an alternate network library (vo, re, ok, async-h, async, see Table 5.1) was used.

For UC-PJO and UC-GJO, three app versions were made for each library as within the implementation of the selected third-party libraries like Volley, Retrofit, etc. There are many choices available regarding which serialization/de-serialization

[5] Code for all app versions: https://github.com/hina86/third_party_HTTP_libraries.git

Figure 5.1: For each use case, the library used for making app versions and the respective app versions

libraries to use. We selected three libraries Gson (G), Moshi (M), and Jackson (J) based on the recommendations and discussion in Android related blogs. In each app version, an alternate network library (vo, re, ok, async-h, async) is used in combination with a serialization/de-serialization library. For example, Volley is first used with Gson (G) and we refer to this combination of libraries as Volley (G) and so on.

For UC-GI, we made app versions in which images were loaded from the server and displayed on the screen. In each app version, alternatively vo, async-h, and pic, uil, and gil are used. In all app versions, tasks are performed asynchronously on a separate thread (not in the main UI thread) by the selected libraries. In UC-GF, UC-PF, and UC-GI, the code for library related operations is put inside a loop with $N = 100$. We keep the loop size to 100 because making just one HTTP request might make a very small difference in energy which might not be detectable. However, sending a larger number of requests will make it easier to detect changes in energy consumption due to these requests.

In UC-GJO and UC-GI, the code for library related operation is put inside a loop with $N = 30$. We keep the loop size to 30, because in each iteration of the loop, besides the HTTP request, an additional step of serialization/deserialization is performed on the payload. Increasing the size of the loop could potentially increase the invocation of the garbage collector to free the memory for the serialization/de-serialization process. This could potentially affect the energy readings. In all use cases, the payload size in each HTTP request is 1MB. Once the operation is complete, the app terminates automatically. Sending HTTP requests in a loop and getting the server response for all HTTP requests is counted as one run. For each app version, we perform ten runs to account for underlying variation in mobile device. The timestamps from the adb logs extracted from the device and the CSV files containing energy measurements are matched to mark the start and end of an execution run.

Table 5.1: Android third-party HTTP libraries used in each selected use case for making app versions.

| ID | Library | UC<br>GF<br>(RQ3-A) | PF<br>(RQ3-B) | PJO<br>(RQ3-C) | GJO<br>(RQ3-D) | GI<br>(RQ3-E) |
|---|---|---|---|---|---|---|
| vo | Volley | | x | | | x |
| vo(G) | Volley(G) | | | x | x | |
| vo(M) | Volley(M) | | | x | x | |
| vo(J) | Volley(J) | | | x | x | |
| re | Retrofit | x | x | | | |
| re(G) | Retrofit(G) | | | x | x | |
| re(M) | Retrofit(M) | | | x | x | |
| re(J) | Retrofit(J) | | | x | x | |
| ok | OkHttp | x | x | | | |
| ok(G) | OkHttp(G) | | | x | x | |
| ok(M) | OkHttp(M) | | | x | x | |
| ok(J) | OkHttp(J) | | | x | x | |
| async-h | Androidasynchttp | x | x | | | x |
| async-h(G) | Androidasynchttp(G) | | | x | x | |
| async-h(M) | Androidasynchttp(M) | | | x | x | |
| async-h(J) | Androidasynchttp(J) | | | x | x | |
| async | Androidasync | x | x | | | |
| async(G) | Androidasync(G) | | | x | x | |
| async(M) | Androidasync(M) | | | x | x | |
| async(J) | Androidasync(J) | | | x | x | |
| pic | Picasso | | | | | x |
| uil | UIL | | | | | x |
| gli | Glide | | | | | x |

For each app version, we recorded energy measurements using the Monsoon power monitor [12], which records the power measurements at a rate of 5KHz. The energy is measured in Joules which is power (watts) times measurement period (seconds). We calculate the energy associated with each reading as $E = \mathit{Power} \times (0.0002)$. The total energy consumption corresponds to the sum of the energy associated with each reading. For filtering energy data related to libraries and for gathering data for execution time, we use adb logs.

Dependent variables in this experiment are energy and execution time, while the independent variable is the choice of library. We assume that our data is non-normal and unpaired; therefore we use the Kruskal-Wallis rank sum test[6] [76] to measure the variance in energy consumption of app versions using different libraries within each use case. Pairwise comparisons[7] between libraries is used to identify significant differences based on $\alpha = 0.05$. Spearman correlation analysis [76] among energy consumption and execution time of libraries within each use

[6] the kruskal.test stats package in R was used to perform kurskal-Wallis rank sum test

[7] The pairwise.wilcox.test stats package in R was used to perform pairwise comparisons. Benjamini & Hochberg (BH) correction was used as correction method.

case is used to evaluate the strength of the relationship.

### 5.1.5. Test Environment

The test environment consists of an HP Elite Book, the Monsoon power monitor, and an LG Spirit Y70 Phone having Android Lollipop (version 5.0.1) as the operating system with 1GB of RAM, and a 2100mAh battery. According to Google distribution dashboard [5], Android Lollipop is among the top five most popular Android OS, more than 15% of Android phones globally are using this version[8]. The selected device uses Qualcomm chipset, which is used globally in more than 45% of the smartphones [9]. Therefore, the selected mobile device is a good representation of an Android smartphone.

Before starting the experiment, each app version is checked for the presence of energy bugs as described by Banerjee et al. [19]. A baseline is recorded to measure the energy consumption of the mobile device in an idle state, which is then subtracted from the actual energy readings during the experiment to filter out the energy used by the app under test. During the experiment, Android settings for screen timeout, display brightness, and sound profiles are kept constant. We made 45 different versions of the app, each using a different library and use case. For connecting to the configured server, WiFi is enabled on the phone. The experiment is controlled from the HP Elite Book using a script that automates the process and runs each app version ten times. This saves manual effort and ensures that no problems are created during the experiment due to human error. The mobile phone is connected to the host computer with a USB cable via Monsoon power monitor [12] which disables the USB phone charging once the energy reading starts.

## 5.2. Results

In this section, we present the results of the analysis of variance (ANOVA). In addition, to better understand the nature of variance we made additional analysis where we make pairwise comparisons between libraries (full results are in Annex A[9]). Throughout this section, following abbreviations are used in tables: KW=Kruskal-Wallis Rank Sum Test, MR=Mean Rank, M=Mean, SD=Standard Deviation, $SE_M$ = Standard Error of Mean. In all figures in this section, the Android third-party HTTP library used in each app version in the particular use case is shown along the x-axis while energy consumption in joule is shown along the y-axis.

[8] Google dashboard: accessed 2019-11-04

[9] Additional material in Annex A. For RG1-RQ3-A, see Table A.6. For RG1-RQ3-B, see Table A.7. For RG1-RQ3-C, see Table A.8. For RQ RG1-RQ3-D, see Table A.9. For RQ RG1-RQ3-E, see Table A.10.

### 5.2.1. RG1-RQ3-A: *When making GET requests is there variance in energy consumption of Android third-party HTTP libraries?*

To answer RG1-RQ3-A, we checked whether the energy consumption between versions of the app making the GET requests using the selected Android third-party HTTP libraries remains the same. This hypothesis is rejected.

Table 5.2 presents the results of the Kruskal-Wallis rank sum test and also shows the summary statistics for energy consumption in joules for libraries included in UC-GF. The analysis of variance using Kruskal-Wallis test showed significant variance, based on $\alpha = 0.05, \chi^2(4) = 42.94, p < .001$, indicating that the mean rank of energy was significantly different between the selected libraries.

Figure 5.2 shows the boxplot of energy consumption in joules for app versions using the selected libraries in UC-GF. Based on mean values, Volley is consuming the most energy, while Androidasync is consuming the least energy. Retrofit (re) and OkHttp (ok) are comparable. They are significantly worse than Androidasync (async) on the one hand and significantly better than Volley (vo) and Androidasynchttp (async-h) on the other hand.

Table 5.2: Results for RG1-RQ3-A and summary statistics (UC-GF)

| | KW Test (p <.001) | Summary statistic for energy consumption (Joule) | | | | |
|---|---|---|---|---|---|---|
| ID. | MR | M | SD | $SE_M$ | Min | Max |
| async | 5.50 | 9.85 | 1.26 | 0.39 | 8.53 | 12.4 |
| ok | 19.00 | 15.44 | 1.22 | 0.38 | 13.80 | 17.18 |
| re | 22.00 | 16.03 | 1.05 | 0.33 | 14.69 | 17.77 |
| async-h | 38.50 | 23.86 | 2.63 | 0.83 | 21.65 | 29.93 |
| vo | 42.50 | 25.42 | 2.07 | 0.65 | 22.0 | 28.56 |

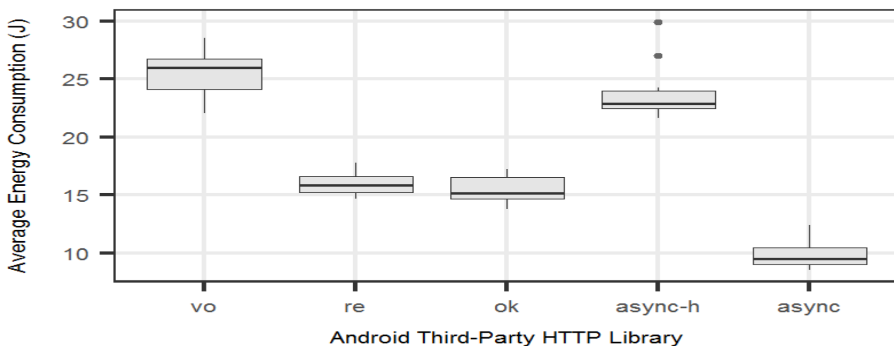


Figure 5.2: Mean energy consumption per library in UC-GF

### 5.2.2. RG1-RQ3-B: *When making multi-part POST requests is there variance in energy consumption of Android third-party HTTP libraries?*

To answer RG1-RQ3-B, we checked whether the energy consumption between versions of the app making multi-part POST requests using the selected Android third-party HTTP libraries remains the same. This hypothesis is rejected.

Table 5.3 presents the results of the Kruskal-Wallis rank sum test for UC-PF and also shows the summary statistics for energy consumption in joules for libraries included in UC-PF. The analysis of variance using the Kruskal-Wallis test showed significant variance, based on $\alpha = 0.05, \chi^2(4) = 32.75, p < .001$, indicating that the mean rank of energy was significantly different between the selected libraries.

Figure 5.3 shows the boxplot of energy consumption in joules for app versions using the selected libraries in UC-PF. Based on mean values we can see that Volley is consuming the most energy while the other four libraries, i.e., Retrofit (re), OkHttp (ok), Androidasynchttp (async-h), and Androidasync (async), did not show significant differences with regards to energy consumption.

### 5.2.3. RG1-RQ3-C: *When sending structured JSON objects is there variance in energy consumption of Android third-party HTTP libraries?*

To answer RG1-RQ3-C, we checked whether the energy consumption between versions of the app sending structured JSON objects using the selected Android third-party HTTP libraries remains the same. This hypothesis is rejected.

Table 5.4 presents the results of the Kruskal-Wallis rank sum test for UC-PJO and also shows the summary statistics for energy consumption in joules for libraries included in UC-PJO. The analysis of variance using the Kruskal-Wallis test showed significant variance, with $\alpha = 0.05, \chi^2(14) = 102.87, p < .001$, indicating that the mean rank of energy was significantly different between the selected libraries.

Figure 5.4 shows the boxplot of energy consumption in joules for app versions using the selected libraries in UC-PJO. Each facet shows the serialization library used with the selected HTTP libraries. Based on the mean values, Volley (vo) has clearly the highest energy consumption while the other four libraries cannot be distinguished significantly. This pattern holds for the situation when using Gson (G) and Moshi (M), while in the sub-case of using Jackson (J) Volley (vo) is having the highest energy consumption but the difference to the other libraries is not as large as in the other two situations.

Table 5.3: Results for RG1-RQ3-B and summary statistics (UC-PF)

| | KW Test (p <.001) | Summary statistic for energy consumption (Joule) | | | | |
|---|---|---|---|---|---|---|
| ID. | MR | M | SD | $SE_M$ | Min | Max |
| re | 10.60 | 9.246 | 1.66 | 0.526 | 6.63 | 11.73 |
| async | 17.40 | 10.74 | 1.70 | 0.54 | 7.90 | 13.99 |
| ok | 25.60 | 12.62 | 2.53 | 0.801 | 9.60 | 17.82 |
| async-h | 28.40 | 13.51 | 3.06 | 0.97 | 8.84 | 17.88 |
| vo | 45.50 | 52.77 | 2.12 | 0.672 | 49.65 | 56.65 |

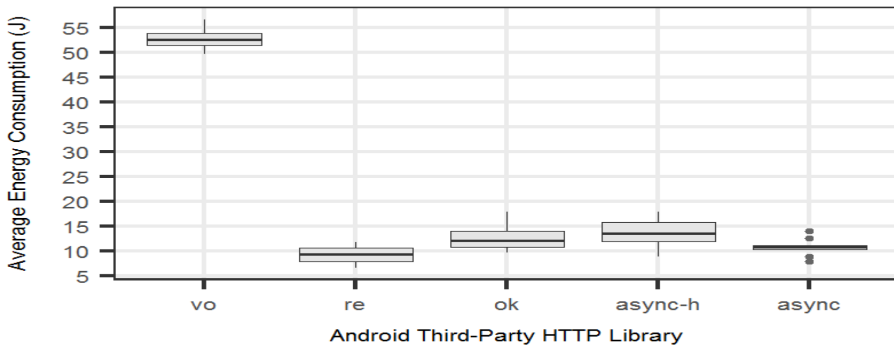


Figure 5.3: Mean energy consumption per library in UC-PF

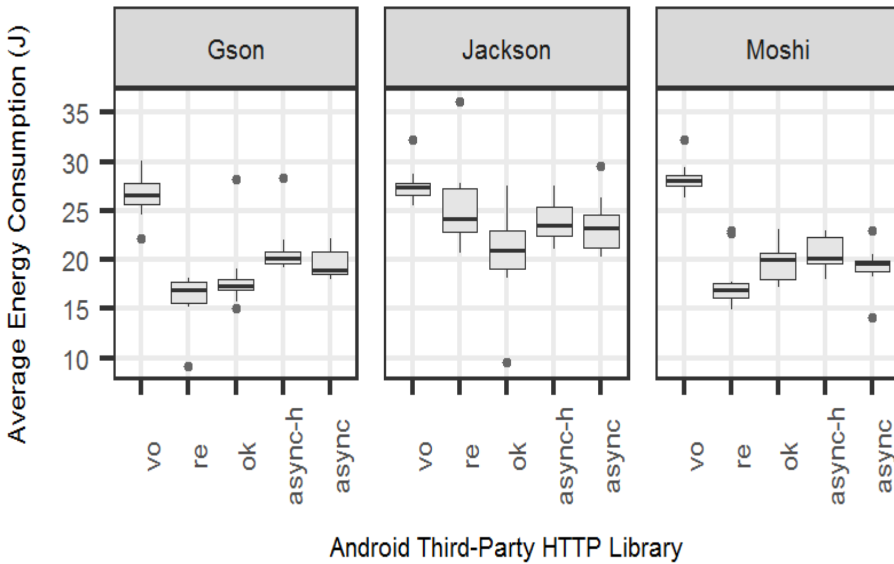


Figure 5.4: Mean energy consumption per library in UC-PJO

Table 5.4: Results for RG1-RQ3-C and summary statistics (UC-PJO)

| | KW Test (p <.001) | Summary statistic for energy consumption (Joule) | | | | |
|---|---|---|---|---|---|---|
| ID. | MR | M | SD | $SE_M$ | Min | Max |
| re(G) | 16.80 | 16.07 | 2.67 | 0.85 | 9.12 | 18.08 |
| re(M) | 29.90 | 17.81 | 2.79 | 0.88 | 14.89 | 23.03 |
| ok(G) | 32.10 | 18.26 | 3.69 | 1.17 | 15.01 | 28.25 |
| async(M) | 51.20 | 19.28 | 2.21 | 0.70 | 14.1 | 22.97 |
| async(G) | 53.20 | 19.57 | 1.44 | 0.45 | 17.9 | 22 |
| ok(M) | 55.00 | 19.80 | 2.04 | 0.64 | 17.14 | 23.06 |
| async-h(M) | 63.80 | 20.61 | 1.88 | 0.59 | 17.95 | 22.93 |
| async-h(G) | 67.40 | 20.98 | 2.73 | 0.86 | 19.25 | 28.36 |
| ok(J) | 70.90 | 20.60 | 4.76 | 1.51 | 9.51 | 27.55 |
| async(J) | 96.50 | 23.51 | 2.89 | 0.91 | 20.33 | 29.57 |
| async-h(J) | 99.90 | 23.89 | 2.13 | 0.67 | 21.13 | 27.48 |
| re(J) | 109.56 | 25.82 | 4.56 | 1.52 | 20.75 | 36.11 |
| vo(G) | 119.00 | 26.50 | 2.20 | 0.70 | 22.17 | 30.13 |
| vo(J) | 127.22 | 27.74 | 1.96 | 0.65 | 25.46 | 32.29 |
| vo(M) | 133.80 | 28.36 | 1.60 | 0.51 | 26.36 | 32.22 |

### 5.2.4. RG1-RQ3-D: *When receiving structured JSON objects is there variance in energy consumption of Android third-party HTTP libraries?*

To answer RG1-RQ3-D, we checked whether the energy consumption between versions of the app receiving structured JSON objects using the selected Android third-party HTTP libraries remains the same. This hypothesis is rejected.

Table 5.5 presents the results of the Kruskal-Wallis rank sum test for UC-GJO and also shows the summary statistics for energy consumption in joules for libraries included in UC-GJO. The analysis of variance using the Kruskal-Wallis test showed significant variance, based on $\alpha = 0.05, \chi^2(14) = 140.70, p < .001$, indicating that the mean rank of energy was significantly different between the libraries.

Figure 5.5 shows the boxplot of energy consumption in joules for app versions using the selected libraries in UC-GJO. Each facet shows the de-serialization library used with the selected HTTP libraries. Based on mean values Androidasynchttp (async-h) was the most energy consuming library whether it is used with Gson (G), Moshi (M) or Jackson (J), while OkHttp (ok), Retrofit (re) and Volley (vo) in combination with Jackson (J) library were least energy consuming.

Table 5.5: Results for RG1-RQ3-D and summary statistics (UC-GJO)

| | KW Test (p <.001) | Summary statistic for energy consumption (Joule) | | | | |
|---|---|---|---|---|---|---|
| ID. | MR | M | SD | $SE_M$ | Min | Max |
| ok(J) | 9.10 | 6.98 | 0.60 | 0.19 | 5.62 | 7.69 |
| re(J) | 16.20 | 7.98 | 1.12 | 0.35 | 6.54 | 9.54 |
| vo(J) | 26.10 | 8.99 | 1.67 | 0.53 | 7.03 | 12.06 |
| ok(G) | 36.44 | 10.76 | 0.47 | 0.16 | 9.81 | 11.34 |
| vo(G) | 48.90 | 11.59 | 1.16 | 0.37 | 10.30 | 14.25 |
| re(G) | 54.00 | 11.68 | 0.43 | 0.14 | 10.82 | 11.08 |
| ok(M) | 65.10 | 12.82 | 1.28 | 0.41 | 11.56 | 14.83 |
| re(M) | 70.80 | 13.89 | 2.26 | 0.72 | 11.26 | 17.12 |
| vo(M) | 77.50 | 14.30 | 1.28 | 0.41 | 12.30 | 16.39 |
| async(J) | 94.50 | 18.54 | 0.55 | 0.17 | 17.48 | 19.60 |
| async(G) | 107.67 | 22.90 | 0.55 | 0.18 | 21.82 | 23.55 |
| async(M) | 117.30 | 23.97 | 1.04 | 0.33 | 22.19 | 25.96 |
| async-h(J) | 117.80 | 24.49 | 1.82 | 0.58 | 22.43 | 28.14 |
| async-h(G) | 133.82 | 28.69 | 1.93 | 0.58 | 23.58 | 31.40 |
| async-h(M) | 143.30 | 30.87 | 1.18 | 0.37 | 29.41 | 33.00 |

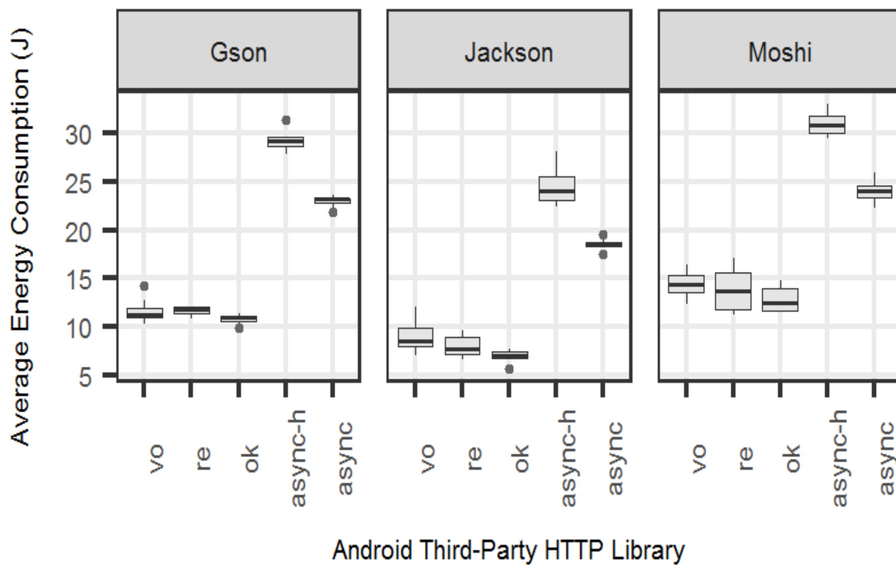


Figure 5.5: Mean energy consumption per library in UC-GJO

### 5.2.5. RG1-RQ3-E: *When loading and displaying images on the screen is there variance in energy consumption of Android third-party HTTP libraries?*

To answer RG1-RQ3-E, we checked whether the energy consumption between versions of the app loading and displaying images on the screen using the selected Android third-party HTTP libraries remains the same. This hypothesis is rejected.

Table 5.6 presents the results of the Kruskal-Wallis test for UC-GI and also shows the summary statistics for energy consumption in joules for libraries included in UC-GI. The analysis of variance using the Kruskal-Wallis test showed significant variance, based on $\alpha = 0.05, \chi^2(4) = 40.04, p < .001$, indicating that the mean rank of energy was significantly different between the selected libraries.

Figure 5.6 shows the boxplot of energy consumption in joules for app versions using the selected libraries in UC-GI. Volley (vo) and Androidasynchttp (async-h) were consuming significantly more energy than Glide (gli), Picasso (pic) and UIL (uil). The variance in energy consumption between Picasso (pic), Glide (gli), and UIL (uil) was not significant.

Table 5.6: Results for RG1-RQ3-E and summary statistics (UC-GI)

| | KW Test (p <.001) | Summary statistic for energy consumption (Joule) | | | | |
|---|---|---|---|---|---|---|
| ID. | MR | M | SD | $SE_M$ | Min | Max |
| uil | 7.80 | 2.65 | 0.3 | 0.1 | 2.2 | 3.23 |
| pic | 18.30 | 3.32 | 0.7 | 0.22 | 2.69 | 4.78 |
| gli | 20.40 | 3.42 | 1.02 | 0.32 | 2.76 | 6.28 |
| async-h | 38.30 | 29.98 | 1.85 | 0.58 | 27.95 | 33.71 |
| vo | 42.70 | 30.76 | 1.32 | 0.42 | 27.75 | 32.65 |

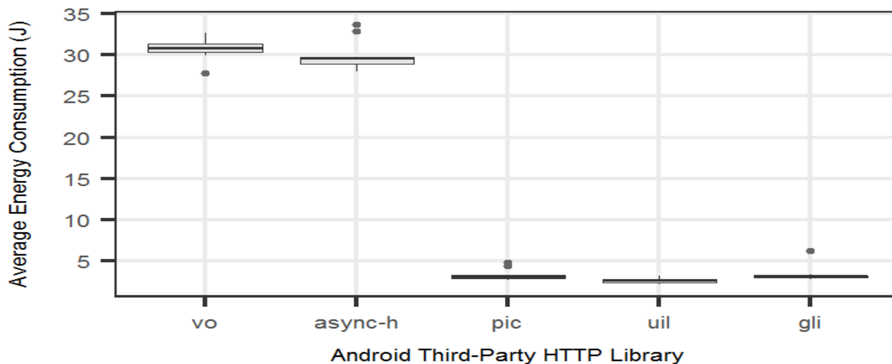


Figure 5.6: Mean energy consumption per library in UC-GI

### 5.2.6. RG1-RQ4: *Is the energy consumption of a third-party HTTP library correlated with execution time?*

To answer RG1-RQ4, we check whether there is positive correlation between the energy consumption and execution time of Android third-party HTTP libraries[10]. This hypothesis is rejected. Table 5.7 shows the results of the Kruskal-Wallis Rank Sum Test and summary statistics for execution time in seconds for all use cases. The results of the spearman correlation between energy and execution time for libraries in each use case are presented in Table 5.8. The analysis of variance using the Kruskal-Wallis test showed significant variance, based on $\alpha = 0.05, p < .001$, indicating that the mean rank of execution time was significantly different between the selected libraries. The correlations were determined based on $\alpha$ = 0.05. Holm corrections used to adjust p-values. Based on $r_s$ and p-value, significant positive correlations were observed between some libraries in each use case (highlighted in blue in Table 5.8). No other significant correlation was observed as p-values for other libraries were greater than 0.05.

Table 5.7: RG1-RQ4 Results and summary statistics

| Library | KW Test (p <.001) | M | SD | $SE_M$ | Min | Max |
|---|---|---|---|---|---|---|
| UC-GF | | | | | | |
| async | 5.50 | 7.75 | 0.81 | 0.26 | 7.17 | 9.99 |
| re | 21.40 | 25.13 | 1.25 | 0.39 | 23.82 | 27.15 |
| ok | 24.50 | 26.31 | 3.07 | 0.97 | 24.08 | 31.77 |
| async-h | 30.70 | 28.93 | 8.23 | 2.60 | 24.39 | 52.22 |
| vo | 45.40 | 58.04 | 5.23 | 1.65 | 51.35 | 69.20 |
| UC-PF | | | | | | |
| async | 14.80 | 26.39 | 5.08 | 1.61 | 18.28 | 35.41 |
| re | 16.20 | 27.35 | 5.98 | 1.85 | 17.56 | 41.17 |
| ok | 25.10 | 32.55 | 8.12 | 2.57 | 22.08 | 47.62 |
| async-h | 25.90 | 32.55 | 6.49 | 2.02 | 22.31 | 42.25 |
| vo | 45.50 | 145.7 | 8.72 | 2.76 | 136.3 | 167.6 |
| UC-PJO | | | | | | |
| re(M) | 9.80 | 23.33 | 0.26 | 0.08 | 23.01 | 23.90 |
| reG) | 16.00 | 22.85 | 3.06 | 0.97 | 14.25 | 24.87 |
| async(M) | 31.00 | 24.66 | 1.14 | 0.36 | 23.43 | 27.66 |
| async(J) | 46.70 | 25.25 | 0.34 | 0.11 | 24.80 | 26.03 |
| async(G) | 47.35 | 25.30 | 0.53 | 0.17 | 24.51 | 26.37 |
| ok(M) | 53.90 | 25.89 | 1.14 | 0.36 | 24.68 | 28.21 |
| ok(G) | 63.95 | 27.20 | 3.42 | 1.08 | 23.82 | 35.46 |
| async-h(M) | 80.15 | 27.90 | 0.42 | 0.13 | 27.20 | 28.65 |

[10]The execution time and energy data used in the analysis is available here https://figshare.com/s/b7a6b37e367ebd351cb9.

| | | | | | | |
|---|---|---|---|---|---|---|
| ok(J) | 90.20 | 30.35 | 6.35 | 2.01 | 15.93 | 38.66 |
| async-h(J) | 92.90 | 28.65 | 0.45 | 0.14 | 28.19 | 29.52 |
| async-h(G) | 93.65 | 31.00 | 6.30 | 1.99 | 27.08 | 47.42 |
| re(J) | 103.67 | 34.00 | 7.16 | 2.39 | 27.40 | 47.24 |
| vo(J) | 125.00 | 41.68 | 0.80 | 0.27 | 40.47 | 42.85 |
| vo(G) | 131.10 | 44.09 | 3.24 | 1.02 | 40.34 | 50.04 |
| vo(M) | 140.10 | 48.55 | 3.54 | 1.12 | 40.80 | 53.14 |
| UC-GJO | | | | | | |
| ok(J) | 8.90 | 9.69 | 0.52 | 0.17 | 9.04 | 10.88 |
| ok(G) | 20.44 | 11.47 | 0.74 | 0.25 | 10.46 | 12.79 |
| re(J) | 22.00 | 12.05 | 2.97 | 0.94 | 9.24 | 16.73 |
| ok(M) | 31.60 | 13.50 | 1.31 | 0.41 | 11.78 | 16.11 |
| re(G) | 41.70 | 15.55 | 0.14 | 0.04 | 15.34 | 15.74 |
| vo(G) | 60.40 | 18.77 | 2.23 | 0.71 | 16.97 | 24.42 |
| re(M) | 67.00 | 19.46 | 0.65 | 0.20 | 18.76 | 20.70 |
| vo(J) | 78.00 | 23.30 | 7.98 | 2.52 | 16.25 | 42.89 |
| vo(M) | 81.70 | 23.73 | 3.58 | 1.13 | 19.71 | 32.07 |
| async-h(J) | 94.20 | 28.38 | 3.08 | 0.97 | 24.77 | 35.83 |
| async(J) | 99.90 | 29.26 | 0.24 | 0.08 | 28.92 | 29.66 |
| async(G) | 119.78 | 36.58 | 0.58 | 0.19 | 36.06 | 37.90 |
| async-h(G) | 120.45 | 36.93 | 2.50 | 0.75 | 33.72 | 42.84 |
| async-h(M) | 132.90 | 39.27 | 2.25 | 0.71 | 36.73 | 44.77 |
| async(M) | 140.50 | 40.28 | 0.38 | 0.12 | 39.79 | 40.78 |
| UC-GI | | | | | | |
| uil | 7.70 | 2.55 | 0.24 | 0.07 | 2.30 | 3.14 |
| pic | 19.20 | 2.98 | 0.45 | 0.14 | 2.56 | 3.94 |
| gil | 19.60 | 3.27 | 1.50 | 0.47 | 2.61 | 7.51 |
| async-h | 35.50 | 33.20 | 8.50 | 2.69 | 27.60 | 55.17 |
| vo | 45.50 | 75.20 | 7.08 | 2.24 | 64.31 | 86.77 |

Table 5.8: Spearman correlation results between energy and execution time for libraries in all use cases

| Library | $r_s$ | Lower | Upper | p-value |
|---|---|---|---|---|
| UC-GF | | | | |
| vo | 0.82 | 0.39 | 0.96 | .004 |
| re | 0.55 | -0.12 | 0.88 | .098 |
| ok | 0.58 | -0.08 | 0.88 | .082 |
| async-h | -0.03 | -0.65 | 0.61 | .934 |
| async | 0.77 | 0.27 | 0.94 | .009 |
| UC-PF | | | | |
| vo | 0.18 | -0.51 | 0.73 | .627 |

| | | | | |
|---|---|---|---|---|
| re | -0.14 | -0.71 | 0.54 | .701 |
| ok | -0.27 | -0.77 | 0.43 | .446 |
| async-h | 0.44 | -0.26 | 0.84 | .200 |
| async | -0.21 | -0.74 | 0.48 | .556 |
| UC-PJO | | | | |
| vo(G) | -0.08 | -0.67 | 0.58 | .829 |
| vo(M) | 0.45 | -0.25 | 0.84 | .187 |
| vo(J) | 0.20 | -0.54 | 0.76 | .606 |
| re(G) | 0.62 | -0.01 | 0.90 | .054 |
| re(M) | 0.72 | 0.17 | 0.93 | .019 |
| re(J) | 0.77 | 0.21 | 0.95 | .016 |
| ok(G) | 0.75 | 0.22 | 0.94 | .013 |
| okM) | 0.48 | -0.22 | 0.85 | .162 |
| ok(J) | 0.43 | -0.27 | 0.83 | .214 |
| async-h(G) | 0.31 | -0.40 | 0.79 | .385 |
| async-h(M) | -0.41 | -0.82 | 0.30 | .244 |
| async-h(J) | 0.02 | -0.62 | 0.64 | .960 |
| async(G) | 0.05 | -0.60 | 0.66 | .881 |
| async(M) | 0.39 | -0.31 | 0.82 | .260 |
| async(J) | 0.27 | -0.43 | 0.77 | .446 |
| UC-GJO | | | | |
| vo(G) | 0.27 | -0.43 | 0.77 | .446 |
| vo(M) | 0.88 | 0.56 | 0.97 | < .001 |
| vo(J) | 0.92 | 0.67 | 0.98 | < .001 |
| re(G) | 0.26 | -0.45 | 0.76 | .476 |
| re(M) | 0.84 | 0.45 | 0.96 | .002 |
| re(J) | 0.89 | 0.60 | 0.97 | < .001 |
| ok(G) | 0.32 | -0.44 | 0.81 | .406 |
| ok(M) | 0.83 | 0.41 | 0.96 | .003 |
| ok(J) | 0.56 | -0.10 | 0.88 | .090 |
| async-h(G) | 0.43 | -0.23 | 0.82 | .190 |
| async-h(M) | 0.72 | 0.17 | 0.93 | .019 |
| async-h(J) | 0.77 | 0.27 | 0.94 | .009 |
| async(G) | -0.02 | -0.67 | 0.65 | .966 |
| async(M) | 0.75 | 0.22 | 0.94 | .013 |
| async(J) | 0.32 | -0.39 | 0.79 | .365 |
| UC-GI | | | | |
| vo | 0.33 | -0.37 | 0.80 | .347 |
| async-h | 0.41 | -0.30 | 0.82 | .244 |
| pic | 0.48 | -0.22 | 0.85 | .162 |
| uil | 0.59 | -0.07 | 0.89 | .074 |
| gli | 0.79 | 0.32 | 0.95 | .007 |

## 5.3. Discussion

In this section, we start out with giving recommendation on how to chose third-party HTTP libraries with regards to energy consumption in various use cases. Then we discuss the relationship between energy consumption and execution time as well as energy consumption and popularity of libraries. We finish the discussion with a summary of general take-aways.

### 5.3.1. Recommendations

In each use case, we observed a significant variance between the eight identified Android third-party HTTP libraries. To help developers choose the most appropriate library in the selected use cases, we provide recommendations in the form of a flowchart (see Figure 5.7).

In UC-GF and UC-PF, based on results of RG1-RQ3-A and RG1-RQ3-B, we recommend simply not to use Volley (vo) if there is not a strong reason for doing so. This recommendation is corroborated when looking at the execution time in RG1-RQ4 (cf. Table 5.7) where, again, Volley (vo) is the worst.

In UC-PJO, based on results of RG1-RQ3-C combined with the results from RG1-RQ4, we recommend using Retrofit (re) in combination with either Gson (G) or Moshi(M) as good alternatives for implementing UC-PJO. This recommendation is corroborated when looking at the execution time in RG1-RQ4 (cf. Table 5.7) where, again, Retrofit (re) with Moshi(M) or Gson (G) performs best.

In UC-GJO, based on the results of RG1-RQ3-D, we recommend using OkHttp (ok), Retrofit (re), and Volley (vo) in combination with Jackson (J) library as alternatives for implementing UC-GJO as compared to Androidasync (async) and Androidasynchttp (async-h). If execution time is also important, then, based on the results of RG1-RQ4, we recommend OkHttp (ok).

In UC-GI, based on the results of RG1-RQ3-E, we recommend UIL (uil), Picasso (pic), Glide (gli) libraries as alternatives for implementing UC-GI. If execution time is also important, then, based on the results of RG1-RQ4, we recommend UIL (uil).

Manual effort is required by developer in order to use the recommendations presented in this section. Therefore to help developers, we provide an open source support tool REHAB[11], implemented as an IntelliJ/Android Studio plugin. If initiated during development, REHAB can help developers choose an energy efficient third-party HTTP library. At the moment, REHAB automates the manual process and is useful in identifying the selected third-party libraries, HTTP request types and use-cases used in this study. However, in future, if energy measurement

[11] see chapter 8 for more details about REHAB tool architecture, implementation and usage

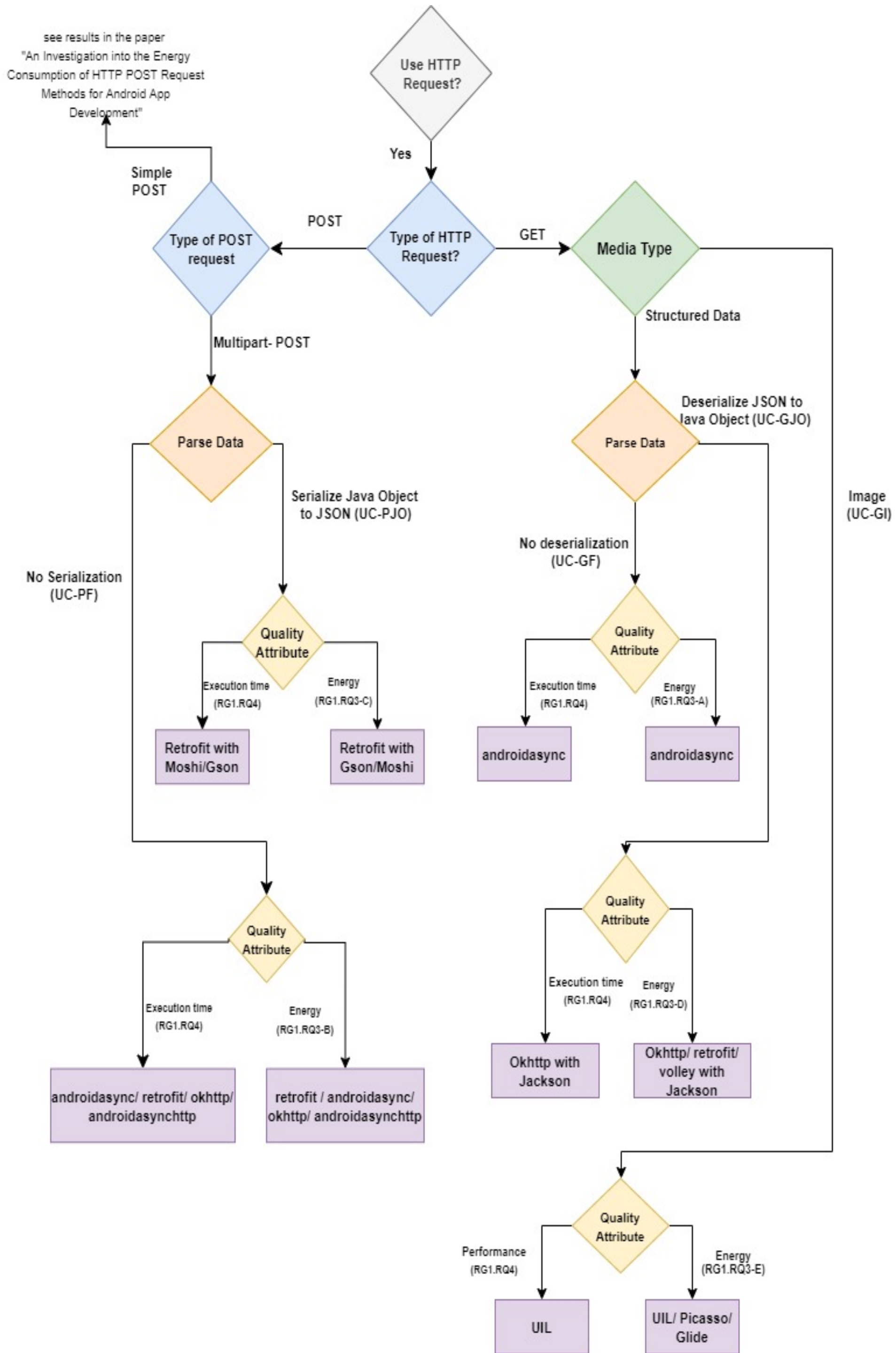


Figure 5.7: Flowchart summarizing recommendations

are done for more third-party libraries, the scope of REHAB can be widened to include more use-cases and related recommendations.

### 5.3.2. Energy Consumption versus Execution Time

We did not observe a positive correlation between energy consumption and execution time in most of the app versions. Only 14 out of 45 app versions in all use cases showed a strong positive correlation between energy consumption and execution time. For each use case, based on the mean value of execution time in summary statistics, we could see that there were significant differences between the execution time of libraries as well. If energy consumption of the third-party library is not a concern, the results of RG1-RQ4 (cf. Table 5.7) could still be used by developers to choose an Android third-party HTTP library based on execution time.

### 5.3.3. Energy Drivers

Although energy consumption of libraries varied significantly, the nature of the variance was inconsistent across the use cases. If one is looking for a result that consistently occurs across use cases, one could, perhaps, mention that the popular library Volley (vo) was most energy consuming in use cases UC-GF, UC-PF, UC-PJO and UC-GI among all libraries selected in our experiment. Generally, the variance in the energy consumption of third-party HTTP libraries could be related to how these libraries deal with multiple threads and requests, if they use thread pools and caching mechanisms. In Volley (vo) by default only four threads are created and reused for asynchronous requests. Volley (vo) uses memory cache (RAM) (it loads all data in memory during parsing), however updating cache state, limited number of threads and frequent context switching due to callback on UI thread might be causing Volley to use more energy than other alternative libraries like Retrofit (re), OkHttp (ok), and Androidasync (async). Retrofit (re) is built on top of OkHttp (ok) and use unlimited thread pools and no caching to complete the requests. Androidasync (async), on the other hand, is built on top of NIO (non-blocking Input/output) model and use single thread along with NIO buffers to efficiently complete the requests. Volley (vo), Retrofit (re), OkHttp (ok,) and Androidasynchttp (async-h) use connection pooling. Starting/stopping a connection is always expensive as it creates an overhead. Reusing an idle connection is much more feasible (as done by Retrofit and OkHttp). Pooled/shared connection also means more complex error handling in the app. Androidasync (async) is avoiding overhead by not using thread and connection pool, which might be the reason behind its less energy consumption. Loading and displaying images on the screen requires memory management especially for large size images to avoid *OutofMemory* error. Picasso (pic), Glide (gli) and UIL (uil) provide better memory management then Volley (vo). Picasso (pic) download the image in its actual size to disk cache and retrieve it from there when needed. In contrast, Glide

(gli) and UIL (uil) get the images from the server, but based on imageview resize it and save it in different resolutions for later use.

### 5.3.4. Energy Consumption versus Popularity

When considering UC-GF to UC-GJO together, then Retrofit (re) and OkHttp (ok) are either the most energy efficient or, at least, at an acceptable middle position regarding energy consumption. Since in an app it is probable that all use cases apply in one way or the other, this might be another argument for developers to use those libraries that are acceptable across the board instead of choosing libraries that are most efficient with regards to energy consumption (and possibly also execution time) in only one or two use cases, such as Androidasync (async). When looking at the popularity statistics we gathered (cf. Section 3.3) from online catalogs, it turns out that not Androidasync (async) but, instead, Retrofit (re) and OkHttp (ok) are the most popular HTTP libraries in the 'network' category and used in most apps. There could be many reasons why Androidasync (async) is not as popular as Retrofit (re) and OkHttp (ok). One could simply be that Androidasync (async) is a new library that only recently has become available. This potential reason can be excluded because all three libraries have been available for almost ten years (cf. Table 5.1 in additional material). Another reason could be that Retrofit (re) and OkHttp (ok) are more popular because they offer more features and offer more support for HTTP/2 and SPDY which Androidasync (async) does not have. Another reason could be the architectural style that the libraries use. Both REST (Representational State Transfer) and RPC (Remote Procedure Calls) use HTTP verbs/methods (GET, POST, DELETE etc.). The choice of library could be dependent on the context and personal preference of the developers. REST uses HTTP verbs to manipulate resource through HTTP protocol, while, RPC use operations to manipulate data and use HTTP for transport. Therefore, REST might seem attractive to developers due to its obvious semantics and could be a possible reason behind the popularity of Retrofit (re) and OkHttp (ok). Finally, it could be that Retrofit (re) and OkHttp (ok) have a better execution time than Androidasync (async). However, as we could see from the results related to RG1-RQ4, in UC-GF, Androidasync (async) is performing better than both Retrofit (re) and OkHttp (ok). In UC-GI, result of RG1-RQ3-E indicate that Picasso (pic), Glide (gli) and UIL(uil) consumes the least energy with no significant variance among them. However, related to the results of RG1-RQ4 UIL (uil) get a clear edge based on its execution time. Based on popularity statistics Picasso (pic) and Glide (gli) are more popular as compared to and UIL (uil), Volley (vo), and Androidasynchttp (async-h). Possible reasons for the popularity of these libraries could be due to their superior memory management and customization capabilities. Assuming that developers make rational decisions, the fact that better energy consumption and better execution time don't automatically make Androidasync (async) or UIL (uil) the first choice indicates that there is a large range of criteria

that developers take under consideration when choosing an Android third-party HTTP library. Energy consumption might not be among them. Our results might help to give a more complete picture of their selection criteria but in order to provide actionable decision-support more detailed trade-off analyses are needed.

#### 5.3.5. General Take-aways

Assuming that in the future the libraries used in this experiment might be depreciated or merged or new versions are published, the take away from this experiment is that the energy consumption of third-party libraries should be taken into consideration during selection. Third-party libraries that offer task-specific features (such as Picasso (pic), Glide (gli), OkHttp (ok) and Retrofit (re)) have less energy consumption than alternative third-party libraries (such as Volley (vo)) providing an extended feature set. Third-party libraries that are task specific are also easier to replace if requirements change in future, hence the effort required to maintain the app code is reduced. Development of task specific third-party libraries and their usage should be encouraged to avoid excessive energy consumption and to improve performance of the apps.

### 5.4. Threats to Validity

In this section, we discuss the possible threats to validity of this study and our strategy to mitigate them.

***Internal validity:*** In order to ensure accuracy of measurements, each app version was run under same conditions. Only necessary services were run on Android device. The caches and data related to each app version was cleaned from the mobile device after each run. To account for underlying variation in energy consumption measurement due to OS related operations, ten samples were recorded for each app version.

***External validity:*** The libraries chosen for energy measurement were selected based on the fact that they were available on online catalogue site like 'AppBrain' and 'Awesome Android'. To ensure that the selected libraries were representative of a commonly used third party HTTP libraries, we checked relative popularity rating in their respective categories. In addition, we also ensured that these libraries were used in a considerable number of apps on the play store.

Changing the version of the selected libraries might also produce different results. The version of the libraries used in the experiment is not necessarily used in all the apps that use these selected libraries. We looked at the changes made in several of the releases of the selected libraries and updates mostly related to error handling and retry policies. Further research could be performed on different versions of the same library to identify the most energy efficient version. In our experiment, we did not consider versions of the same library as it exponentially increased the

number of times the execution needs to be performed and it was not possible to measure energy for all of them in all the possible scenario (that would require a much larger study both in terms of time, processing power, and human resources). There is a risk that our results might not generalize to the Android app's population, hence to minimize it we selected popular third-party libraries.

Increasing the magnitude of study in terms of a number of included libraries in future work is desirable. Increasing the number of libraries in the experiment also exponentially increases the number of app versions that need to be tested. As writing app versions is a slow process that requires manual effort, we limited this experiment to eight libraries.

It is important to note that in order to replicate the results of this experiment the same phone model and Android OS version, measurement tools and app version should be used. The reason is that different mobile phones with different Android OS versions might use different levels of energy. To ensure that the selected device was a good representation of an Android device we checked the device specifications, i.e., Android version, Chipset, screen resolution and size, battery capacity and RAM size.

***Construct validity:*** Accuracy of energy measurements depend on the sampling frequency of the measurement device. Energy measurements were recorded using the Monsoon power monitor [12], which recorded power consumption at a rate of 5KHz. The same specification has been used in previous studies [20, 38, 81, 91, 97, 157, 180] so sampling frequency is sufficient to produce precise readings. Energy measurements were made for the apps plus Android OS related activities. We tried to minimize this threat by using adb logs and matching the timestamps with the energy trace to filter out any unwanted readings. We repeated the tests ten times for each app version and included the averaged reading to further mitigate variations in the readings. However, recording the logs for the app introduces an overhead which in itself could be energy consuming. Since this overhead was constant between the versions of the app it could be ignored.

# 6. TOOL SUPPORT FOR GREEN ANDROID DEVELOPMENT

In this chapter, we focus on RG2 (defined in Section 1.1) to improve the tool support for energy efficient mobile app development. In order to build effective Android-specific support tools to aid development of green Android apps, we first need to understand what is already available, what is still needed, and how the problems in existing tools could be overcome. Based on published literature we outlined an explorative analysis of support tools available to 1) optimize code in Android apps through code smell or energy bug detection or refactoring, and 2) to optimize reusable code in Android apps through detection or migration of third-party libraries. We explore if these support tools aid green Android development. We also provide an overview of the techniques used in these support tools.

## 6.1. Research Method

We conducted a systematic mapping study following the method described in [129]. First, we formulated research questions, then based on those research questions we formulated two general search queries and conducted the search in the following online repositories for primary publications: IEEE Xplore, ACM digital library, Science Direct, and Springer. We cover publications from 2014 to June 2020, as from 2014 onwards the focus of many publications has been Android and energy-efficient app development, indicating a shift in research focus.

### 6.1.1. Research Questions

As the objective of this study is to analyze the current support tools available to improve custom code through detection or refactoring of code smell or energy bugs and to improve reusable code through detection or migration of third-party libraries in Android apps, we formulated the following research questions.

RG2-RQ1

*What state of the art support tools have been developed to aid software practitioners in detecting or refactoring Android specific code smells and energy bugs in Android apps?*

RG2-RQ2

*What state of the art support tools have been developed to aid software practitioners in detecting or migrating third-party libraries in Android apps?*

RG2-RQ3

*How do existing support tools compare to one another in terms of techniques they use for offering the support?*

RG2-RQ4

*How do existing support tools compare to one another in terms of the support they offer to practitioners for improving energy efficiency in Android apps?*

RG2-RQ1 and RG2-RQ2 aims to classify publications based on the tools they offer. RG2-RQ3 aims to classify and analyze publications based on techniques used in the tool to offer support to developers. RG2-RQ4 deals with the characterization of all the identified tools in terms of the support (such as output or interface or availability, etc.) they offer to developers to aid green Android development.

### 6.1.2. Search Query

We derived search terms to use in our search queries from the research questions of this study. We looked for alternatives to the search terms in publications we already knew and refined our search terms to return the most relevant publications. We used the '*' operator to cover possible variations of the selected search terms in the search query. Keyword 'OR' was used to improve search coverage.

The first search query is designed to retrieve publications that provide a support tool to detect/refactor code smells/energy bugs in Android apps. This search query extends our previous study [54] in following ways: we improved the search strings to cover missing papers e.g. the terms ("energy effici*" OR "code optimization") were replaced by ("energy" OR "efficien*" OR "code smell" OR "optimiz*") and so on. We also extended our search in terms of publication years to include one more year.

The second search query is designed to retrieve publications that provide a support tool to detect/migrate third-party libraries in Android apps.

We did not use the search terms 'mobile development', 'apps', 'optimization', 'green', 'sustainability', and 'recommendation' in isolation as they were too high level and produced a quite large corpus consisting of a high number of irrelevant

publications while the search terms ‘resource leaks’, ‘API’, ‘tool’, ‘framework’, and ‘technique’ were eliminated to avoid being too specific. The search queries were applied to popular online repositories (IEEE Xplore, ACM digital library, Science Direct, and Springer) to find a dataset of relevant primary publications. In each repository, based on available advanced search options, filters were applied to refine the query results. Applied filters are shown in Table 6.1. The search queries were applied to the titles, abstracts, and keywords of the publications.

**Search Query 1:**
*Android AND (energy OR code smell OR bug OR refactor* OR correct* OR detect* OR optimiz* OR efficien*) AND NOT (environ* OR iot OR edu* OR hardware OR home)*

**Search Query 2:**
*(Android) AND (“third-party libr*” OR “third-party Android lib*” OR "libr*") AND NOT (environ* OR iot OR indus* OR edu* OR hardware OR home)*

Table 6.1: Search query filter

| Filter | Value |
|---|---|
| Publication year | 2014 – 2020 (up till June) |
| Content-Type | Journal Article, Conference Paper |

### 6.1.3. Screening of Publications

We first removed duplicate results and then defined inclusion, exclusion, and quality criteria for further screening of search results.

***Duplicate Removal***. The search results from online repositories were first loaded in Zotero (an open-source reference management system) to create a dataset of relevant publications. Using the feature in Zotero duplicates publications were removed from the dataset. Next, we manually applied inclusion, exclusion, and quality criteria on the remaining publications.

***Inclusion Criteria***. For inclusion, the selected publication should be a primary study generally related to the software engineering domain with a focus on third-party libraries or code smells or energy bugs in Android apps. A tool/automated technique for third-party library/code smell/energy bug detection, modification or replacement was presented in the publication to support Android development. We considered only conference and journal articles published in English.

***Exclusion Criteria***. Publications that were unrelated to Android development or third-party library/code smell/energy bugs in Android apps were excluded. The

publications that focused only on hardware, environmental, security, privacy, networks, malware, clones, repackaging of apps, obfuscation issues, iOS, or present secondary data were also excluded. The work presented in a thesis or a book chapter is usually published in relevant journals or conferences as well. Therefore, doctoral symposium papers, magazine articles, book chapters, work-in-progress papers, and papers that were not in English were excluded as well.

***Quality Criteria***. The quality criteria applied to selected publications are shown in Table 6.2. Abstracts of the publications and structure of the publication were inspected for further quality assessment. If a quality rule was true for a publication, it was awarded full points; otherwise, no points were awarded. In case a rule partially applied to a publication, half points were awarded. After applying all five quality rules, the points were added to get a final quality score for a publication. A maximum quality score of 3 could be assigned to a publication. If a publication was below a total quality score of 2, it was removed from the results. The quality score threshold is set to 2 so that publications above the 50% score are included. It ensures the rigour and relevance of selected primary publications.

Table 6.2: Quality assessment criteria

| ID | Description | Rating |
|---|---|---|
| 1 | Does the publication clearly state contribution(s) that is directly related to third-party libraries/code smells/ energy bugs in Android apps? | 0.5 |
| 2 | Is the contribution(s) related to green in Android development?? | 0.5 |
| 3 | Is contribution(s) a tool or automated technique that could be used in Android development or maintenance? | 1 |
| 4 | Is the research method adequately explained? | 0.5 |
| 5 | Are threats to validity and future research directions separately discussed? | 0.5 |
| | **TOTAL** | **3** |

### 6.1.4. Classification and Analysis

To answer RG2-RQ1-RG2-RQ3, we identified the main keywords of the selected publications along with the commonly used terms in the abstracts to define categories of support tools. Research methodology and results of selected publications were additionally studied when needed. We kept extracted data in excel spreadsheets for further processing. During data extraction, if there was a conflict of opinion, it was discussed among authors until a consensus was reached.

To answer RG2-RQ1 and RG2-RQ2, a bottom-up merging technique was adopted to build our own classification schemes (see Table 6.3 and 6.4). Once classification schemes were established, we extracted data from each selected publication to identify its main contribution and assigned the tool mentioned in the publication

to a category based on the classification scheme.

To answer RG2-RQ3, a classification scheme was needed to classify techniques used in support tools for offering support to aid Android development. We used the bottom-up approach to build this classification scheme by combining the specialized analysis methods/techniques into more generic higher-level techniques. The identified generic techniques along with their definitions, are described in Tables 6.5 and 6.6. Once we had established the classification schemes, we extracted data from the abstract and research methodology of each selected publication and assigned it to a category defined in the classification schemes.

To answer RG2-RQ4, we extracted data form each selected publication to gather information about the kind of support the identified tool offers. We compare these tools based on the inputs of the tool, outputs of the tool, code smells/energy bugs/third-party libraries coverage, interface type, integrated development environment (IDE) support, and availability. In general, a code smell is defined as "a surface indication that usually corresponds to a deeper problem in the system" [59] and an energy bug is defined as "error in the system (application, OS, hardware, firmware, external conditions or combination) that causes an unexpected amount of high energy consumption by the system as a whole" [124]. A third-party library is a reusable component related to specific functionality that can be integrated into the application to speed up the development process. A third-party library could be for advertising, analytics, Image, Network, Social Media, Utility etc. [177]. In the light of these definitions, we looked for Android-specific code smells, energy bugs, and third-party libraries in the studies.

Table 6.3: Categories of support tools (RG2-RQ1)

| ID | Category | Description |
|---|---|---|
| CP | Profiler | A software program that measures the energy consumption of an Android app or parts of apps. |
| CD | Detector | A software program that only identifies and detects energy bugs/code smells in an Android app. |
| CO | Optimizer | A software program that identifies energy bugs/code smells as well as refactor source code of an Android app to improve energy consumption. |

Table 6.4: Categories of support tools (RG2-RQ2))

| ID | Category | Description |
|---|---|---|
| CI | Identifier | A software program that only identifies and detects third-party libraries in an Android app. |
| CM | Migrator | A software program that identifies third-party libraries as well as helps in updating or migrating the third-party libraries (to an alternative library or version) in the source code of an Android app. |
| CC | Controller | A software program that identifies third-party libraries to control, isolate, or deescalate privileges and permissions granted to third-party libraries in an Android app. |

Table 6.5: Categories of techniques used in support tools for code smells or energy bugs (RG2-RQ3)

| ID | Technique | Definition |
|---|---|---|
| T1 | Byte Code Manipulation | A technique that injects code in the Smali files of the app under test. The injected code is either a log statement or an energy evaluation function. These statements help find out the part of the source code that consumes a specific amount of energy at runtime. |
| T2 | Code Instrumentation | A technique that instruments the app, using instrumented test cases that are capable of running specific parts of the app, in such a way that it is run in a specific environment while calling known methods/classes of the app under test. It uses finite state machines and device-specific power consumption details to measure energy. |
| T3 | Logcat Analysis | A technique that uses system-level log files to obtain energy consumption information provided by OS for the app under test. These logs are compared with application-level logs to give graphical information about the energy consumption of the app. |

| | | |
|---|---|---|
| T4 | Static Source Code Analysis | A technique that uses the source code of the app and analyses it using one or combination of the following methods: control flow graphs analysis, point-to-analysis, inter-procedural, intra-procedural, component call analysis, abstract syntax tree traversal or taint analysis. |
| T5 | Search-based algorithms | A technique that uses a multi-objective search algorithm to find multiple refactoring solutions and the most optimal solution is selected as final refactoring output by iteratively comparing the quality of design and energy usage. |
| T6 | Dynamic Analysis | A technique based on the identification of information flow between objects at run time for the detection of vulnerabilities in the app under test. It monitors the spread of sensory data during different app states. |

Table 6.6: Categories of techniques used in support tools for third-party libraries (RG2-RQ3)

| ID | Technique | Definition |
|---|---|---|
| T7 | Feature Similarity | A technique that uses machine learning to extract code clusters or train classifiers, by using feature hashing or similarity metrics or pattern digest or similarity digest, in order to identify and classify third-party libraries. |
| T8 | Whitelist Comparison | A technique that compares third–party library names/versions/package information to whitelist in order to detect third-party libraries. |
| T9 | API Hooking | A technique that intercepts or redirect API calls at various levels in order to regulate permission or policy-related operations. |
| T10 | Module Decoupling | A technique to divide code in modules and extract code features such as package name, package structure, inheritance relationships for clustering/classification to detect library. |

| T11 | Process Isolation | A technique to isolate untrusted components in the operating system. This technique requires system-level modification. |
|---|---|---|
| T12 | Class Profile Similarity | A technique to extract (strict or relaxed) profiles from libraries and apps code based on structural hierarchies. Based on similarity (exact or fuzzy) between these profiles library is detected. |
| T13 | Collaborative Filtering | A technique to predict or recommend third-party libraries based on feature vectors and its similarity against a set of similar apps or neighbourhood apps. It Includes model-based approaches (such as matrix factorization), memory and item-based approaches. |
| T14 | Natural Language Processing | A technique used to identify or recommend third-party libraries based on textual descriptions. It includes techniques such as word embedding, skip-gram model, continuous bag-of-words model, domain-specific relational and categorical tag embedding, and topic modelling. |

## 6.2. Results

In this section, first we present the publications that were found when applying the search strings and the other steps outlined in the method (cf Section 6.1.4) for a) support tools for code smells or energy bugs and b) for support tools for third-party libraries. The list of selected publications and additional details about code smells or energy bugs covered by support tools are shown in a Annex B. Then, each subsection (6.2.1, 6.2.2, 6.2.3, and 6.2.4), is dedicated to answering one research question.

*Search Query 1 (support tools for code smells or energy bugs)*: As a result of running search query 1 and applying filters on search results, 2334 publications were found from the selected online repositories. These publications were loaded into the Zotero software for the screening and removal of duplicates, the total number of publications were reduced to 2241 after duplicate removal. Inclusion and exclusion criteria were applied to the remaining publications, and the number was reduced to 575. We read abstracts of these publications and looked at the structure to assign them a quality score based on quality criteria. After applying the quality criteria, the number of selected publications was reduced to 24. (See Tables 6.7, 6.8, and 6.9)

*Search Query 2 (support tools for third-party libraries)*: As a result of running search query 2 and applying filters on search results, 545 publications were found from the selected online repositories. These publications were loaded into the Zotero software for the screening and removal of duplicates, the total number of publications were reduced to 521 after duplicate removal. Inclusion and exclusion criteria were applied to the remaining publications and the number was reduced to 131. We read abstracts of these publications and looked at the structure to assign them a quality score based on quality criteria. After applying the quality criteria, the number of selected publications was reduced to 27. (See Tables 6.10, 6.11, and 6.12).

The four RQs are answered one by one in the following subsections.

Table 6.7: Number of publications extracted per online repo. (search query 1)

| Sr. | Repo. | No. of studies | Conf. Papers | Journal Articles |
|---|---|---|---|---|
| 1 | IEEE Xplore | 1170 | 910 | 260 |
| 2 | ACM Digital library | 483 | 459 | 24 |
| 3 | Springer | 595 | 362 | 231 |
| 4 | Science Direct | 86 | 4 | 82 |

Table 6.8: Number of publications per screening step (search query 1)

| Sr. | Step in the screening of publications | No. of publications |
|---|---|---|
| 1 | Search string results after applying filters | 2334 |
| 2 | Remove duplicates | 2241 |
| 3 | Apply inclusion and exclusion criteria | 575 |
| 4 | Apply quality criteria | 24 |

Table 6.9: Quality score assigned to each selected publication (search query 1)

| Publication ID | Quality Score |
|---|---|
| P2, P11, P12,P14, P16, P17, P19, P20 | 2 |
| P5, P6, P7, P13, P21, P24 | 2.25 |
| P1, P4, P8, P9, P10, P15, P22, P23 | 2.5 |
| P3, P18 | 2.75 |

Table 6.10: Number of publications extracted per online repo. (search query 2)

| Sr. | Repo. | No. of studies | Conf. Papers | Journal Articles |
|---|---|---|---|---|
| 1 | IEEE Xplore | 312 | 296 | 12 |
| 2 | ACM Digital library | 177 | 157 | 20 |
| 3 | Springer | 28 | 22 | 6 |
| 4 | Science Direct | 28 | 0 | 28 |

Table 6.11: Number of publications after applying filters and screening steps (search query 2)

| Sr. | Step in the screening of publications | No. of publications |
|---|---|---|
| 1 | Search string results after applying filters | 545 |
| 2 | Remove duplicates | 521 |
| 3 | Apply inclusion and exclusion criteria | 131 |
| 4 | Apply quality criteria | 27 |

Table 6.12: Quality score assigned to each selected publication (search query 2)

| Publication ID | Quality Score |
|---|---|
| P31, P43 | 2 |
| P26, P29, P33, P34, P35 ,P36, P39, P40, P48, P49 | 2.25 |
| P25, P27, P28, P30, P32, P37, P38, P41 ,P42, P44, P45, P46, P47, P50, P51 | 2.5 |

### 6.2.1. RG2-RQ1: *What state of the art support tools have been developed to aid software practitioners in detecting or refactoring Android specific code smells and energy bugs in Android apps?*

To answer RG2-RQ1, the classification scheme defined in Table 6.3 (cf. Section 6.1.4) was used and the selected publications were divided into three categories, i.e., 1) 'Profiler', 2) 'Detector', and 3) 'Optimizer', based on the support tool they offer to aid green Android development. Table 6.13 gives an overview of the distribution of selected publications in each category, along with the total number of tools in each category. Figure 6.1 shows the number of publications each year. The color in bars indicates the number of tools in each category each year from 2014 to 2020. We can see a decrease in the number of 'Profiler' tools while an increase in the number of 'Optimizer' tools. In 2019 and 2020 (till June), no new 'Detector' tool has been published.

Table 6.13: Distribution of publications in each category (RG2-RQ1)

| Category | Selected Publications | No. of Tools |
|---|---|---|
| Profiler | P6, P14, P16, P12, P13, P20, P19 | 7 |
| Detector | P1, P3, P4, P5, P8, P9, P7, P17 | 8 |
| Optimizer | P10, P11, P15, P18, P2, P21, P22, P23, P24 | 9 |

### 6.2.2. RG2-RQ2: *What state of the art support tools have been developed to aid software practitioners in detecting or migrating third-party libraries in Android apps?*

To answer RG2-RQ2, the classification scheme defined in Table 6.4 (cf. Section 6.1.4) was used and the selected publications were divided into categories: 1)

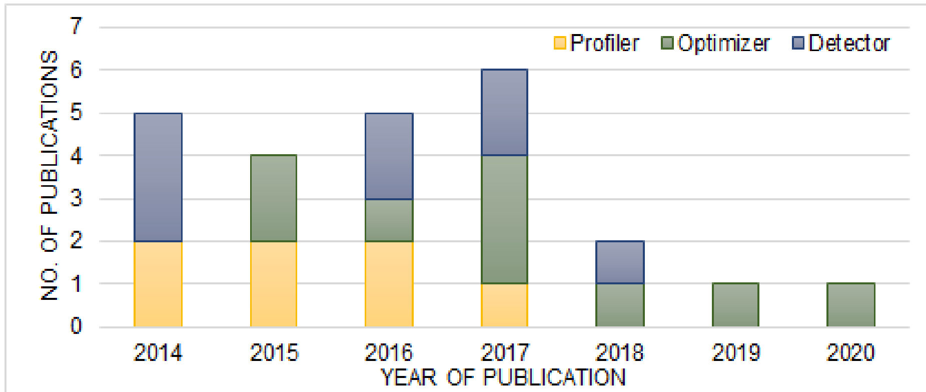


Figure 6.1: Publication per year per category (Search Query 1)

'Identifier', 2) 'Migrator', 3) 'Controller', based on the support tool they offer to aid Android development. Table 6.14 gives an overview of the distribution of selected publications in each category, along with the total number of tools in each category. Figure 6.2 shows the number of publications each year. The color in bars indicates the number of tools in each category each year from 2014 to 2020 (till June). We can see at least one 'Identifier' and 'Controller' tool each year. We can see an increase in the number of 'Migrator' tools in 2019 and 2020.

Table 6.14: Distribution of publications in each category (RG2-RQ2)

| Category | Selected Publications | No. of Tools |
|---|---|---|
| Identifier | P26, P27, P29, P30, P31, P32, P33, P37, P40, P41, P42, P44, P47, P48, P49 | 16 |
| Migrator | P35, P45, P50, P51 | 4 |
| Controller | P25, P28, P34, P36, P38, P39, P43, P46 | 7 |

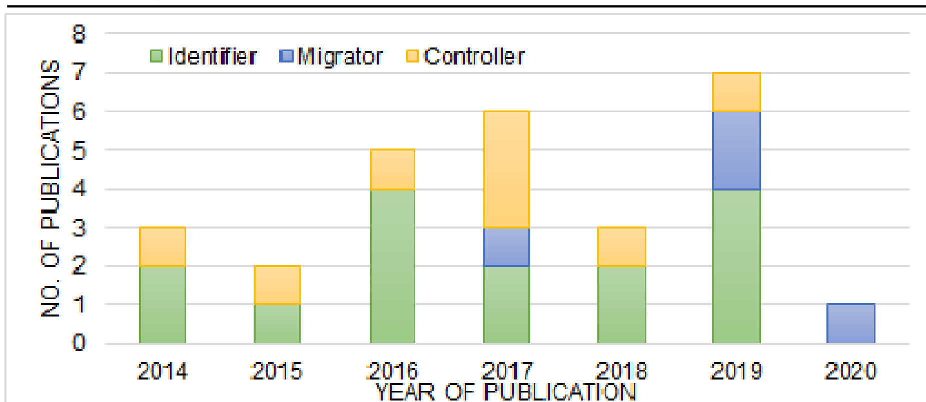


Figure 6.2: Publication per year per category (Search Query 2)

### 6.2.3. RG2-RQ3: *How do existing support tools compare to one another in terms of techniques they use for offering the support?*

To answer RG2-RQ3, we identify techniques used in each tool for improving the energy efficiency of apps. Table 6.15 and 6.16 gives an overview of tools and techniques along with reference to selected publications. Based on Table 6.15, we observed that no tool in any category used a combination of techniques. Each tool could be easily classified into exactly one category of techniques (defined in Section 6.1.4). However, in Table 6.16 many tools used a combination of techniques such as module decoupling and feature similarity, or collaborative filtering and natural language processing. As a result of fine-tuning the search query 1, we were able to identify three new 'Optimizer' tools [P22, P23, and P24] which used static source code analysis to refactor and optimize the app code.

**Profiler.** Profiling tools measure the energy consumption of the software and the used hardware resources. Profilers [P6, P16, and P14] were designed to inject logging statements into the Smali files to gather relevant log data. This data was then transformed as needed and matched against pre-defined API names or method names to identify energy intensive APIs and methods in the program. In these profilers, Dumpsys or System level logs or ADB data plotted on a graph was used to measure energy usage. Another group of profilers analyzed the paths in source codes using finite state models (FSM) or inter procedural control-flow paths. The authors of [P19] created a dynamic model that instrumented the app to measure its power consumption. Methods were invoked using tests and were classified into three different levels (classes, packages, and projects) based on their energy consumption. The classification was then presented graphically. The authors of [P12] modelled energy consumption as an FSM model by collecting events from an Android device. By executing the instrumented unit tests the line of source code with the energy consumption issue was identified. The authors of [P13] sliced the

Table 6.15: Overview of support tools (for code smells or energy bugs detection and refactoring) showing the technique used for offering support to developers (RG2-RQ3)

| | Techniques | | | | | |
|---|---|---|---|---|---|---|
| Ct. | T1 | T2 | T3 | T4 | T5 | T6 |
| CP | P6, P16 | P12, P13, P19 | P14, P20 | - | - | P17 |
| CD | - | - | - | P1, P3, P7,P8, P9, P4, P5 | - | - |
| CO | - | - | - | P11, P21, P2, P10, P22, P23, P24 | P15 | P18 |

CP=Profiler, CD= Detector, CO= Optimizer, T1=Byte code manipulation, T2= Code instrumentation, T3=Logcat analysis, T4= Static source code analysis, T5=Search-based algorithms, T6=Dynamic analysis

Table 6.16: Overview of support tools (for third-party library detection and migration) showing the technique used for offering support to developers (RG2-RQ3)

| | Technique | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Ct. | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 |
| CI | P26, P42, P49, P33, P37, P40 | P31, P47 | - | P27, P29, P32, P44, P33, P37, P40, P47 | - | P30, P41, P48 | - | - |
| CM | - | - | - | P45 | - | - | P35, P51 | P35, P50 |
| CC | - | - | P34, P36, P38, P39, P43, P46 | P28 | P25 | - | - | - |

CI=Identifier, CM= Migrator, CC=Controller, T7= Feature similarity , T8=Whitelist comparison, T9= API hooking, T10=Module decoupling, T11=Process isolation, T12=Class profile similarity, T13= Collaborative filtering, T14= Natural language processing

program into executable slices for generating feature predictors. Predictors were generated based on some predefined metrics which quantified energy consumption. The authors of [P20] measured energy consumption by encapsulating the code blocks with energy evaluation functions. Sampling collector extracted energy consumption and a runtime manager extracted application information from OS.

**Detector.** The authors of [P17] used JPF to statically analyze the state space of the app to find the usage of sensors and wake locks. The utilization of sensory data was defined by a coefficient. The authors of [P7] converted APK files into byte code using ‘SOOT’ which is a static source code analyzer. The converted code was then used to identify and report various antipatterns in code. The authors of [P5] identified resource intensive items in an app using control-flow graphs (CFGs). Using resource protocols as a guide a taint-like analysis was performed on CFGs. In the study [P1] valid inter-procedural control-flow paths were analyzed in each callback method and its transitive callees, in order to detect energy draining operations related to missing deactivation behaviors. In the study [P3], a static analysis tool ‘SOOT’ was used to find energy bugs in graphics intensive mobile apps along with point-to analysis tool, ‘SPARK’. Energy bugs were identified based on the frequency at which GPU was being used to perform certain actions e.g. texture transformations etc. The authors of [P8] used flexible bug pattern specification notation (FBPSN) to specify bug patterns. The source code was transformed to CFGs which were used to detect bugs with the help of FBPSN specified bug patterns. In the study [P9] the APK files were analyzed using ‘APKTool’ and resource leak were analyzed using ‘SAAF’. ‘Lint’ was used for lay-

out defect analysis. 'SAAF' internally uses inter-procedural, intra-procedural and component call analysis and resource leak detection. The output from 'Lint' was passed through a filter based on a set of rules defined in a defect table to generate the report. The results of both these tools are used to generate the respective reports. In the study [P4] an abstract syntax tree of classes was traversed to apply a set of detection rules based on a specific set of code smells.

**Optimizer.** The authors of [P2] introduced the tool 'DelayDroid' which used static analysis and byte code refactoring using ASM library to find the parts of code that performed energy intensive network related tasks and batched them together to perform those tasks at a delayed interval to reduce energy consumption. The tool 'Hot-pepper' was presented in [P10] which detected energy smells using the 'Paprika' tool and computed the energy consumption of code smells using the 'Naga-Viper' tool. Corrections were made using a tool called 'SPOON' that used static analysis for transformation. The authors of [P11] presented a tool developed on top of the Eclipse refactoring engine which converted AsyncTask to Intent-Service in order to improve the asynchronous operations. In the study [P15] a novel tool 'EARMO' was presented which created code abstractions to search for anti-patterns based on QMOOD metrics. To correct anti-patterns in the app, the 'ReCon' tool was used. In [P18] a framework was presented that had three main components: Design extractor, refactoring component and code generation component. Design and defect expressions were generated from EFG using deterministic finite automata. Their intersection was used to refactor the code. The authors of [P21] used 'PMD' and 'Android Lint' to create an Eclipse plugin for refactoring of source code. 'PMD' created an Abstract Source Tree (AST) to analyze code and apply the pre-defined rules.

**Identifier.** 'Identifier' tools mostly use feature similarity or module decoupling or both techniques to detect third-party libraries. The authors of [P26] used similarity digest (which are similar to standard hashes) and compared them against a database consisting of original compiled code of third-party libraries. The authors of [P42] also used similarity digests to measure the similarity between data objects. The authors of [P49] used design pattern digests, fuzzy signatures along with fuzzy hash to match design patterns from app and library code. The authors of [P27 and P29] identified third-party libraries by decoupling an app into modules using package hierarchy clustering, and clustering based on locality sensitive hashing, respectively. The authors of [P32 and P44] decoupled app into modules to extract package dependencies for identifying third-party libraries. The authors of [P33, P37 and P40] used a combination of module decoupling and feature hashing/digests to provide list of detected third-party libraries. The authors of [P47] used whitelist based detection for non-obfuscated apps and used motifs subgraph based detection for obfuscated apps. The authors of [P31] used whitelist based detection by comparing library name and package information against a list of commonly used third-party libraries. The authors of [P31, P41, and P48] ex-

tracted method signatures, and package hierarchy structures from libraries to build profiles per library and used these profiles for third-party library identification.

**Migrator.** 'Migrator' tools are mostly a combination of collaborative filtering and natural language processing techniques. The authors of [P35] used collaborative filtering in combination with topic modelling (applied to the textual description in readme files). Based on results of topic modelling, similar apps were identified, and the set of third-party libraries extracted from these similar apps were then used to recommend libraries to developers. The authors of [P50] applied word embedding and domain-specific relational and categorical knowledge on stack overflow questions to recommend alternative libraries. The authors of [P51] used collaborative filtering and applied matrix factorization approach to neutralize bias while recommending libraries. The authors of [P45] used 'LibScout' tool to extract library profiles. These profiles are then used to determine if a library version should be updated or not.

**Controller.** 'Controller' tools mostly use API hooking techniques to provide control over library privileges based on policy. The authors of [P34] intercept and control framework APIs. The authors of [P36] intercept system APIs to extract runtime library sequence information. The authors of [P38] track the execution entry of the module and all related asynchronous executions at thread level. The authors of [P39] use the tool 'Soot Spark' to get call graphs in order to identify Android APIs that leak data (based on a given policy). The authors of [P43] used binder hooking, in-VM API hooking and GOT (global offset table) hooking to regulate permission and file related operation of third-party libraries. The authors of [P46] intercept permissions protected calls and check them against a compiled list of third-party libraries in order to regulate privileges. The authors of [P28] extract code features and package information to train a classifier to detect libraries and grant them privileges. The authors of [P25] used system-level process isolation in order to separate third-party library privileges.

*Techniques used to provide support by the various categories of support tools for detecting or refactoring code smells or energy bugs are the following:*

*'Profiler' tools typically use a variety of techniques to measure energy consumption but none of the tools in this category uses static source code analysis. Almost all 'Detector' and 'Optimizer' tools use static source code analysis of APK/SC based on a predefined set of rules.*

*Techniques used to provide support by the various categories of support tools for detecting or migrating third-party libraries are the following:*

*'Identifier' tools use a variety of techniques for detecting third-party libraries. However, feature similarity and/or module decoupling techniques are more frequent. Almost all 'Migrator' tools used collaborative filtering and/or natural language processing techniques to recommend library migration. Almost all 'Controller' tools used API hooking techniques to control privileges/permissions related to third-party libraries.*

#### 6.2.4. RG2-RQ4: *How do existing support tools compare to one another in terms of the support they offer to practitioners for improving energy efficiency in Android apps?*

To answer RG2-RQ4, we first list all the support tools for code smell/energy bug detection/correction (see Table 6.17) and compare them in terms of input, output, user interface, integrated development environment (IDE) integration, availability, and code smell/energy bug coverage. Second, we list all the support tools for detecting/migrating third-party libraries (see Table 6.18) and compare them in terms of input, output, library coverage, user interface, availability, and IDE integration support. In Tables 6.17 and 6.18, the 'input' column provides information about what is the input for each tool. The 'output' column provides information about the support the tool offers based on the input. The 'UI' column provides information about the user interface of the tool. The 'open source' column provides information about tool availability for usage/extension. The 'IDE' column (see Table 6.17) provides information about the IDE integration capability of tools. The 'TPL Type' column (see Table 6.18) provides information about the third-party library(TPL) coverage of the tool.

**Support tools for code smell or energy bug detection or refactoring**

In Table 6.17, we provide a list of all the tools identified in 'Profiler', 'Detector', and 'Optimizer' categories. As a result of fine-tuning the search query 1, we were able to identify three new 'Optimizer' tools [P22, P23, P24] that were not included in our previous work [54], For all the 24 tools listed in Table 6.17 we provide additional information related to interface, availability and IDE integration that was not included in previous work [54].

Studies in the category 'Profiler' offer support to the practitioners by providing tools that can measure the energy consumed by whole/parts of an app or device sensors used in the apps. The measured information is usually presented to practitioners as graphs for energy consumption over time. Studies in the 'Profiler' category do not recommend when, where, and how practitioners can use the information from these graphs during development to improve the energy consumption

of their apps. Studies in the category ‘Detector’ offer support to practitioners by developing tools that present as output lists of energy bugs/code smells causing a change in energy consumption of apps. Studies in the category ‘Optimizer’ offer support to practitioners by developing tools that present as output refactored source code of apps optimized for energy. The studies in this category do not explicitly give the recommendation to the developers about how to optimize the source code for energy efficiency as the tools automatically refactor the code.

Table 6.17: List of support tools in ‘Profiler’, ‘Detector’, and ‘Optimizer’ categories along with information about their inputs and outputs, user interface, IDE support and availability (RG2-RQ4)

| Ct. | Tool | Input | Output | UI | IDE | Open Source | ID |
|---|---|---|---|---|---|---|---|
| CP | Orka | APK | ECG | GUI | No | No | P6 |
| | SEPIA | AE | ECG | GUI | No | No | P12 |
| | Mantis | PBC | Program CRC predictors | CMD | No | No | P13 |
| | AEP* | SL, PID via ADB | ECG | GUI | No | No | P14 |
| | E-Spector | SL, AL via ADB | ECG | GUI | No | No | P16 |
| | SEMA | PID, MVC | Log of EC | CMD | No | No | P20 |
| | Keong et. al | SC | ECG | GUI + CMD | No | No | P19 |
| CD | Wu et al. | SC | List of energy bugs | CMD | No | No | P1 |
| | Kim et al. | PBC | List of energy bugs | CMD | No | No | P3 |
| | Statedroid | APK | List of energy bugs | CMD | No | No | P5 |
| | PatBugs | SC | List of detected warnings | NS | No | No | P8 |
| | SAAD | APK | List of energy bugs | CMD | No | No | P9 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | aDoctor | SC | List of code smells | GUI + CMD | No | Yes | P4 |
| | GreenDroid | PBC,CF | List of energy bugs + severity level | CMD | No | Yes | P17 |
| | Paprika | APK, PM | List of code smells | CMD | No | Yes | P7 |
| CO | DelayDroid | APK | Refactored APK | NS | No | No | P2 |
| | HOT-PEPPER | APK | Most energy efficient APK, Refactored SC, and List of refactoring | CMD | No | Yes | P10 |
| | Asyncdroid | SC | Refactored SC | GUI | Eclipse | No | P11 |
| | EARMO | APK | Refactored APK | CMD | No | Yes | P15 |
| | EnergyPatch | APK | Refactored APK | GUI | Eclipse | No | P18 |
| | Nguyen et al. | SC | Refactored SC | GUI | Eclipse | No | P21 |
| | Chimera | SC | Refactored APK | CMD | Android Studio | No | P22 |
| | ServDroid | APK | Refactored APK | CMD | No | Yes | P23 |
| | Leafactor | SC | Refactored APK file | GUI | Eclipse | Yes | P24 |

(Ct.=Category, CP=Profiler, CD= Detector, CO= Optimizer, SC=Source Code, APK=Android Package Kit, PBC=Program Byte Code, SL= System Log files, AL= Application Log files, PID= Process ID, ADB=Android Debug Bridge, CRC= Computational Resource Consumption, AE= Application Events, CF= configuration Files, MVC= Measurements of Voltage and Current, ECG= Energy Consumption Graph, SM= Software Metrics values, PM=PlayStore Metadata, GUI = Graphical User Interface, CMD= Command Line, EC= Energy Consumption)

Out of 24 tools listed in Table 6.17, only seven are open source. Out of the seven open-source tools, three are ‘Detector’ tools, and four are ‘Optimizer’ tools. Most of the tools do not offer IDE integration. Four tools in ‘Optimizer’ category support integration with Eclipse IDE [P11, P18, P21, P24] while one tool [P22] supports integration with Android Studio IDE. Out of 24 tools, 12 offer command-line interface (CMD) [P1, P3, P5, P9, P7, P10, P13, P15, P17, P20, P22, P23], eight tools offer graphical user interface (GUI) [P6, P11, P12, P14, P16, P18, P21,

P24], two tools offer both [P4, P19], while for the rest of them information about interface is not specified in the publications.

See Annex B for detail about definitions of code smells/ energy bugs covered by tools in 'Detector' and 'Optimizer' categories. Figure 6.3 shows the Android energy bug coverage of tools in the 'Detector' and 'Optimizer' category. The Android energy bugs are shown on the horizontal axis. The percentage of tools in the 'Detector' and 'Optimizer' categories covering Android energy bugs is shown on the vertical axis. We can see that Android energy bugs 'TMV', 'TDL', 'UL', 'UP', 'VBS' are detected by 13% of the tools, whereas 'RL', 'WB' and 'NCD' are detected by 75%, 50% and 38% of tools, in the 'Detector' category respectively. None of the tools in the 'Optimizer' category covers. 'TMV', 'TDL', 'UL', and 'UP' energy bugs. On the other hand, energy bugs 'IB', 'OLP', 'VHB', 'EMC' are covered by tools in the 'Optimizer' category, whereas none of the tools in the'Detector' category covers them. 'RL' and 'VBS' energy bugs are detected by 44% of the tools in the 'Optimizer' category.

Figure 6.4 shows the Android code smell coverage of tools in the 'Detector' and 'Optimizer' category. The Android code smells are shown on the vertical axis. The percentage of tools in the 'Detector' and 'Optimizer' categories covering Android code smells is shown on the horizontal axis. We can see that Android code smell 'ERB' and 'VHP' is not detected by any tool in the 'Detector' category, whereas 'LWS', 'LC', 'RAM', 'PD', 'ISQLQ', 'IDFP', 'DW; 'DR', and 'DTWC' are not detected by any of tools in the 'Optimizer' category. Android code smells such as 'IOD', 'HBR', 'HSS', 'HAT', 'IWR', 'UIO', 'BFU', 'UHA', 'LWS', 'LC', 'SL', 'RAM', 'PD', 'NLMR', 'MIM', 'LT', 'IDS', 'IDFP', 'DW', 'DR', 'DTWC' are detected by 13-25% of the tools in the 'Detectors' category

*Typical support given by the various categories support tools for detecting and refactoring code smells/energy bugs are as follows:*

*'Profiler' tools support developers by visualizing the energy consumption of the whole app or parts of it. 'Detector' tools support developers with lists of energy bugs and code smells to be manually fixed by the developer for energy improvement. 'Optimizer' tools support developers by automatically refactoring APK/SC versions based on pre-defined rules.*

### Support tools for third-party library detection or migration

In Table 6.18, we provide a list of all the tools identified in 'Identifier', 'Migrator', and 'Controller' categories. Publications in the category 'Identifier' offer support to the practitioners by providing tools that detect third-party libraries present in the apps. The information is usually presented to practitioners as a list of detected libraries along with their version or similarity scores. Publications in the

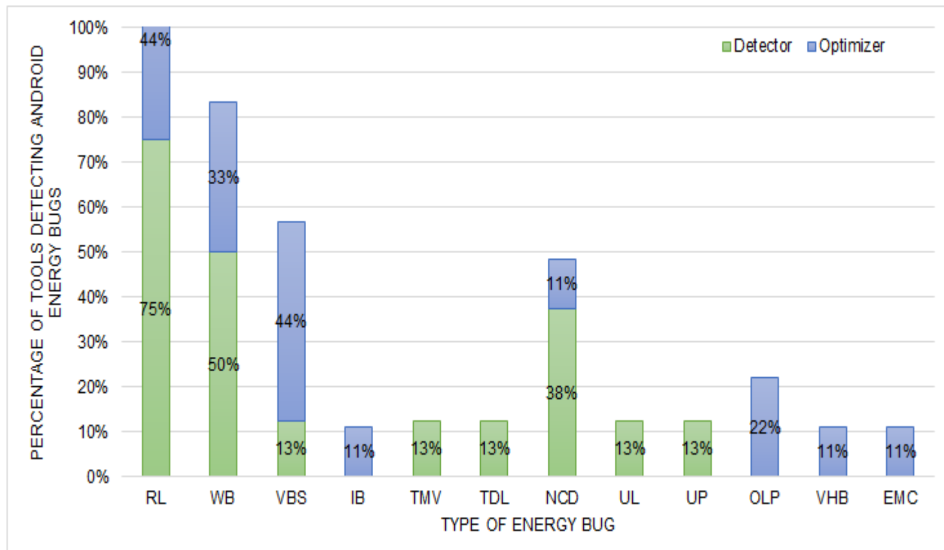


Figure 6.3: Percentage of the tools in 'Detector' and 'Optimizer' categories that can detect Android energy bugs.

*(RL=Resource Leak, WB=Wake-lock Bug, VBS=Vacuous Background Services, IB= Immortality Bug, TMV=Too Many Views, TDL= Too Deep Layout, NCD=Not Using Compound Drawables, UL= Useless Leaf, UP=Useless Parent, OLP=Obsolete Layout Parameter, VHB= View Holder Bug, EMC=Excessive Method Calls)*

category 'Migrator' offer support to practitioners by developing tools that present as output lists of recommended third-party libraries. Publications in the category 'Controller' offer support to practitioners by developing tools that present as output policy-based privilege or permission control over third-party libraries. Most tools in 'Identifier', 'Migrator', and 'Controller' categories provide coverage for all types (advertisement, social, network, billing, analytics etc) of Java-based third-party libraries. Some tools such as AdDetect (CI), or Pedal (CC) cover only the advertisement related third-party libraries. NativeGuard (CC) provides coverage for only native third-party libraries. Reaper (CC) and LibCage (CC) provide coverage for native and Java-based third-party libraries. For many tools listed in Table 6.18, interface type was not specified in publications, while others provide either a command-line interface (CMD) or a graphical user interface (GUI). Out of 27 tools listed in Table 6.18, only seven tools are open source. Out of the seven open-source tools, six tools [P30, P31, P33, P37, P42, P48] are 'Identifier' tools and one tool [P46] is a 'Controller' tool. None of the tools listed in Table 6.18 provides IDE integration support.

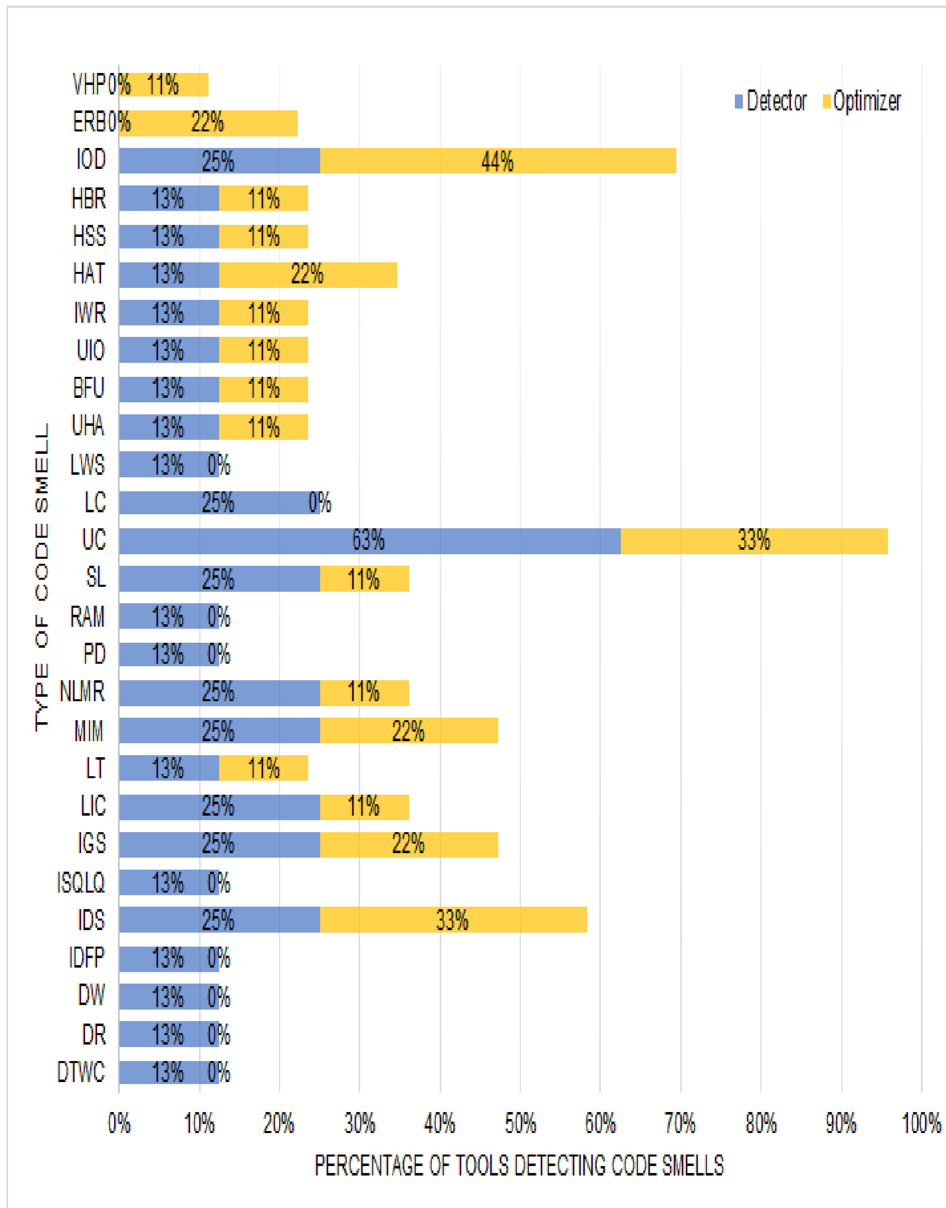


Figure 6.4: Percentage of code smells detected by each tools in 'Detector' and 'Optimizer' categories.

*(DTWC=Data Transmission Without Compression, DR=Debuggable Release, DW=Durable Wakelock, IDFP=Inefficient Data Format and Parser, IDS=Inefficient Data Structure, ISQLQ=Inefficient SQL Query, IGS=Internal Getter and Setter, LIC=Leaking Inner Class, LT=Leaking Thread, MIM=Member Ignoring Method, NLMR=No Low Memory Resolver, PD=Public Data, RAM=Rigid Alarm Manager, SL=Slow Loop, UC=Unclosed Closeable, LC=Lifetime Containment, LWS= Long Wait State, UHA=Unsupported Hardware Acceleration, BFU= Bitmap Format Usage, UIO=UI Overdraw, IWR=Invalidate Without Rect, HAT=Heavy AsyncTask, HSS=Heavy Service Start, HBR=Heavy Broadcast Receiver, IOD=Init ONDraw, ERB=Early Resource Binding, VHP= View Holder Pattern.)*

Table 6.18: List of support tools in ‘Identifier’, ‘Migrator’, and ‘Controller’ categories along with information about their inputs and outputs, library coverage, UI, and availability (RG2-RQ4)

| Ct. | Tool | Input | Output | TPL Type | UI | Open Source | Ref |
|---|---|---|---|---|---|---|---|
| CI | Duet | APK | Library integrity -pass/fail ratio | Java | NS | No | P26 |
| | AdDetect | APK | List of detected TPLs | Java-Ad | NS | No | P27 |
| | AnDarwin | APK | Detect and exclude TPLs + Set clone or rebranded apps | Java | NS | No | P29 |
| | LibScout | TPL .jar/.aar + APK | Presence of given TPL based on similarity score | Java | CMD | Yes | P30 |
| | DeGuard | APK | De-obfuscated APK (containing detected TPLs) | Java | GUI* | Yes | P31 |
| | LibSift | APK | List of detected TPLs | Java | NS | No | P32 |
| | LibRadar | APK | List of detected TPLs sorted by popularity + info about TPLs | Java | GUI* | Yes | P33 |
| | LibD | APK | List of detected TPLs | Java | CMD | Yes | P37 |
| | Ordol | APK | List of detected TPL versions + similarity score. | Java | NS | No | P40 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | LibPecker | TPL name+ APK | Presence of given TPL based on the similarity score | Java | NS | No | P41 |
| | Orlis | APK | List of detected TPLs | Java | NS | Yes | P42 |
| | PanGuard | APK | List of detected TPLs | Java | GUI* | No | P44 |
| | He et al. | APK | List of detected TPLs + risk assessment | Java | NS | No | P47 |
| | Feichtner et al. | APK/TPL | List of detected TPLs and versions + similarity score | Java | CMD*** | Yes | P48 |
| | DPAK | APK/ Android jar | List of detected TPLs | Java | CMD*** | No | P49 |
| CM | AppLibRec | SC | List of recommended TPLs | Java | NS | No | P35 |
| | Appcommune | APK | Tailored app without TPLs and updated/-customized TPLs | Java | GUI** | No | P45 |
| | SimilarTech | TPL name | List of recommended TPLs + information about usage | Java | GUI* | No | P50 |
| | LibSeek | APK | List of recommended TPLs | Java | NS | No | P51 |
| CC | NativeGuard | APK | Split original APK into Ser-vice APK and Client APK | Native | CMD | No | P25 |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | Pedal | APK | Repackaged APK with privilege de-escalated for detected TPLs. | Java-Ad | GUI** | No | P28 |
| | LibCage | SC + list of permissions required by TPLs | Deny unnecessary TPL permission on runtime | Java+ Native | NS | No | P34 |
| | Zhan et al. | SC+ Policy | Grant or deny permissions to TPLs based on policy | Java | NS | No | P36 |
| | SurgeScan | TPL byte-code + Android.jar + policy | Dex and jar files of TPL with the policy implemented | Java | NS | No | P39 |
| | AdCapsule | SC+ policy | Grant or deny permissions to TPLs based on policy | Java-Ad | NS | No | P43 |
| | Reaper | APK | Grant or deny permissions to TPLs based on user preference | Java + Native | GUI** | Yes | P46 |

(Ct.=Category, CI=Identifier, CM= Migrator, CC=Controller UI= User Interface, SC=Source Code, APK=Android Package Kit, TPL= Third-Party Libraries, GUI= Graphical User Interface, CMD= Command-line Interface, NS= Not Specified in publication, *Web service, **App on Android device, *** executable jar, NS= not specified in the publication.)

*The tools in ‘Identifier’, ‘Migrator’, and ‘Controller’ categories do not detect/update/control/migrate third-party libraries to optimize the source code of Android app for energy efficiency.*

*Typical support given by the various categories of support tools for detecting and migrating third-party libraries are as follows:*

*’Identifier’ tools support developers by detecting third-party libraries present in apps. ’Migrator’ tools support developers with lists of recommended third-party libraries along with the mapping information of these libraries for updating/migrating them. ’Controller’ tools support developers by separating third-party library privileges from the app privileges based on policy defined by developers.*

## 6.3. Discussion

In this section, we highlight what is lacking and make suggestions for future research directions regarding the two types of tool support.

### 6.3.1. Support Tools for Code Smell and Energy Bug Detection or Refactoring

Regarding support tools for code smell and energy bug detection and refactoring, we identified the following shortcomings.

- ***Lack of open-source tools:*** We observed that most of the support tools in ‘Profiler’, ‘Detector’ and ‘Optimizer’ categories are not open-source, making these tools inaccessible to many developers.
- ***Lack of IDE integration:*** Most of the support tools in ‘Profiler’, ‘Detector’ and ‘Optimizer’ categories do not support IDE integration. Due to the rapid development process of Android apps, developers are more likely to use tools that are integrated with the IDEs and share the same interface design.
- ***Limited code smell and energy bug coverage:*** Each tool in ‘Detector’ and ‘Optimizer’ categories provided a limited coverage over Android-specific code smells or energy bugs. In principle, if developers spend time and effort to learn one such tool, they still might not be able to identify many code smells and energy bugs in their code unless they use a combination of these tools to get complete coverage.
- ***Vague industry relevance:*** The industry relevance of the current state of the art support tools might not be obvious because they are not evaluated in industrial settings.

- ***Lack of hybrid analysis techniques:*** Most tools in 'Detector' and 'Optimizer' categories used static source code analysis, which indicates that these tools do not cover dynamic issues such as those related to asynchronous tasks.

Based on the above shortcomings we propose the following future research opportunities

- ***Open-source tools and IDE integration:*** There is a need for more open-source 'Profiler', 'Detector' and 'Optimizer' tools to be integrated with current IDEs for better accessibility. The process of setting up and controlling hardware-based energy measurements for apps is cumbersome. Open-source 'Profiler' tools for hardware-based measurements could be useful in conducting controlled experiments to generate accurate energy data in a specific context. IDE integration of such tools will help reduce the learning curve for the users and help with the adaption of the tools (we present one such tool (ARENA) in Chapter 7).

- ***Extension of existing support tools:*** The current state of the art tools could be extended to integrate with other industrially famous code analyzers like Android Lint, Check Style, Find Bugs and PMD. Work done by Goaër et al. [68] and Fatima et al. [53] are promising as they extend the Android Lint tool to provide coverage for various code smells and energy bugs.

- ***Tool development using hybrid techniques*** For the development of better support tools, hybrid techniques encompassing both dynamic and static analysis could be used. In addition, non-intrusive techniques could be used to collect software metrics for identifying code smells and energy bugs.

- ***Improving scope of primary studies***: The results from the selected publications could be expanded to include cross-project predictions and corrections for energy bugs. Analysis and inclusion of multi-threaded programming approaches in the experiments could be another direction for future researchers.

### 6.3.2. Support Tools for Third-party Library Detection or Migration

Regarding support tools for third-party library detection and migration, we identified the following shortcomings.

- ***Lack of IDE integration:*** We observed that none of the support tools in 'Identifier', 'Migrator' and 'Controller' provide support for IDE integration, and many of these tools are also not open source, making them inaccessible to developers.

- ***Lack of energy-related support and library coverage:*** We also observed that none of the support tools in 'Identifier', 'Migrator', and 'Controller'

categories offers any support to developers to aid the development of green Android apps. One possible reason could be that so far research related to third-party library identification is mostly used in clone detection, detection of rebranded or similar or malicious apps, detection of issues related to security, privacy or data leaks. Rasmussen et al. [134] showed that blocking advertisements in Android apps reduce energy consumption. However, these studies have only focused on a small subset of network and advertisement related libraries. Current state-of-the-art explore limited types and distribution of commonly used third-party libraries such as ads, billing, and social libraries.

- ***Lack of hybrid analysis techniques:*** Support tools in the 'Identifier' category roughly use two techniques a) whitelist-based b) similarity-based. Tools that use whitelist based approaches are fast due to smaller feature sets thus could perform better in large scale analysis. However, this technique cannot identify third-party libraries without prior knowledge. On the other hand, tools that use similarity-based approaches such as feature hashing use a larger feature set and can identify third-party libraries without prior knowledge. Due to the extended feature set, these tools might be more accurate but time-consuming.

- ***Lack of detection accuracy:*** Many tools in the 'Identifier' category (such as 'LibD', 'LibScout', 'LibRadar', 'AdDectect' etc.) consider code obfuscation during library detections to give accurate results. However, not many tools are resilient against code shrinking as they rely on package hierarchies.

Based on the above shortcomings we propose the following future research opportunities

- ***Candidates for tool extension:*** Support tools in 'Migrator' category are good candidates for an extension to offer energy-related support as collaborative filtering, and natural language processing techniques could supplement the data gathered from the energy reading of third-party libraries. Such information could be useful in mapping the function of one library to another alternative library for a smooth migration.

- ***Improving detection techniques:*** Support tools in 'Controller' categories rely on API hooking techniques that separate libraries from app code. Such tools could also benefit from using an access control list (ACL) to split privileges. Because current techniques require system-level changes, which makes the deployment of 'Controller' tools difficult.

- ***Recommend energy-efficient third-party libraries:*** Energy consumption of other libraries such as network, analytical, utility, etc., is not fully explored in literature and merits further research. Tool developers could use

data from such studies to recommend energy-efficient libraries to developers during development (we present one such tool (REHAB) in Chapter 8).

## 6.4. Threats to Validity

In this section, we discuss the possible threats to validity of this study and our strategy to mitigate them.

***Internal validity:*** The search queries and classification of selected publications could be biased by the researcher's knowledge. We mitigated this threat by following the method described by Petersen et al. [129] for conducting a systematic mapping study. We defined the inclusion, exclusion and quality criteria for the selection of the publications. We searched for publications in only four online repositories (IEEE Xplore, ACM digital library,Science Direct, and Springer), as these repositories cover most of the high quality publications in the domain of software engineering.

***Construct validity:*** In order to avoid false-positive and false negatives in the search results, we used the wildcard character (*) to maximize coverage and the keyword 'AND NOT' to remove irrelevant studies. We did not use the terms 'energy' or 'efficiency' in combination with 'Android' in the second search query, as we have already executed this combination in search query 1. The results of the search strings were manually checked and further refined. We already knew about many relevant studies and we recaptured almost 90% of them when we executed the search queries. On each online repository the search mechanism is slightly different we tried to keep the queries as consistent as possible, but there might be a slight difference due to the difference in search mechanism provided by different online repositories. We have excluded publications that did not focus on Android development yet still contributed a tool for detecting or recommending third-party libraries. Maven central repository contains a huge quantity of Java-based third-party libraries that could be used in any Java-based application. However, in this study, we focused particularly on the support tools for energy profiling, code optimization and refactoring of code smells or energy bugs, detection or migration of third-party libraries to help aid development of green Android apps. Other types of support tools, such as tools for style checking, interface optimization, test generation, requirement engineering, code obfuscation, etc., were not in the scope of this study. Therefore, while applying inclusion/exclusion criteria, we filtered support tools such as 'LibFinder', LibCPU, CrossRec and RAPIM [15, 113, 119, 146]. These tools could identify or recommend third-party libraries but they were not designed to be used specifically with Android apps. We plan to cover such tools in future work.

***External validity:*** We only covered publications from 2014 to June 2020. Online repositories continuously update their databases to include new publications,

therefore, executing the same queries might yield some additional results that were not included in this study. Some selected publications use the terms code smells and energy bugs interchangeably which could affect the classification. To mitigate this threat, we used the selected definitions (cf. section 6.1.4) for code smells and energy bugs to correctly classify the studies in the right category.

# 7. ARENA: A TOOL FOR MEASURING AND ANALYSING THE ENERGY EFFICIENCY OF ANDROID APPLICATIONS

In this chapter, we also focus on RG2 (defined in Section 1.1) to improve the tool support for energy efficient mobile app development. In the previous chapter, we provided an overview of state-of-the art and compared the current available support tools to aid green Android development. Recent studies [28, 128, 130] indicate that user acceptance of energy-draining apps is low. To make energy-efficient mobile apps, there is a need for tools that assist software practitioners in estimating the energy consumption of an app when it is running on a device. Therefore, in this chapter, we present an open-source tool **ARENA** (**A**nalyzing ene**R**gy **E**fficiency in a**N**droid **A**pps) to support the energy measurement and analysis process and to reduce the risks related to human errors. This tool integrates all the activities necessary to measure, statistically analyze and report (including result interpretations and visualization in the form of graphs) the energy consumption of Android apps.

The energy measurement and analysis process typically involves setting up an energy measurement environment, executing the app under test (AUT) on the mobile device, and recording current/voltage data, usually at the rate of 5KHz and above. Once the energy data is acquired, it needs to be cleaned from noise and aggregated over several samples to account for variations in energy consumption due to background processes in the mobile device. Further, data is visualized or statistically analyzed to discover significant variations in energy consumption. The energy measurements could be captured either via hardware-based approaches (e.g., using devices such as Monsoon power monitor[1]) or via software-based approaches (e.g., such as PowerAPI[2]). As compared to software-based approaches, hardware-based approaches are more accurate in capturing energy measurements but at the same time more cumbersome to implement. Several empirical studies exist [74, 85, 114, 144] in which either one or both of these approaches are used to measure energy consumption of mobile apps. In each of these studies, the authors employ their own methods for measuring energy consumption, and most of the work is done manually or via specialized scripts. Therefore, it is difficult to compare and reproduce their results. Estimating the energy consumption of an Android app is challenging and resource intensive. To overcome these problems, a systematic and fully/semi-automated process is needed to ensure that the measurements are performed consistently and reliably [101]. Previously, many tools have been developed to estimate energy consumption [75, 110, 117, 126, 167, 171] of

[1] https://www.monsoon.com/high-voltage-power-monitor

[2] https://github.com/powerapi-ng/powerapi-scala

apps. However, they target large-scale app store analysis after an app has been published, or they use outdated hardware for physical measurements. Few tools exist that help developers estimate the energy consumption of an app during development. Physalia is a python library[3] that helps software practitioners in taking hardware based energy measurements for mobile apps. However, it does not provide integration with Android Studio IDE. Android Profiler within the Android Studio IDE estimates energy consumption via a software-based approach but does not provide means to analyze and report the energy consumption between different apps, or different versions of the same app, via a hardware-based approach. As the process of recording hardware-based energy measurement is lengthy with a steep learning curve, we present a new support tool ARENA for energy measurement of Android apps.

In the rest of the chapter, we describe ARENA's architecture and explain how ARENA supports the energy measurement and analysis process. Then, we provide implementation details. Finally, we present ARENA in a typical usage scenario.

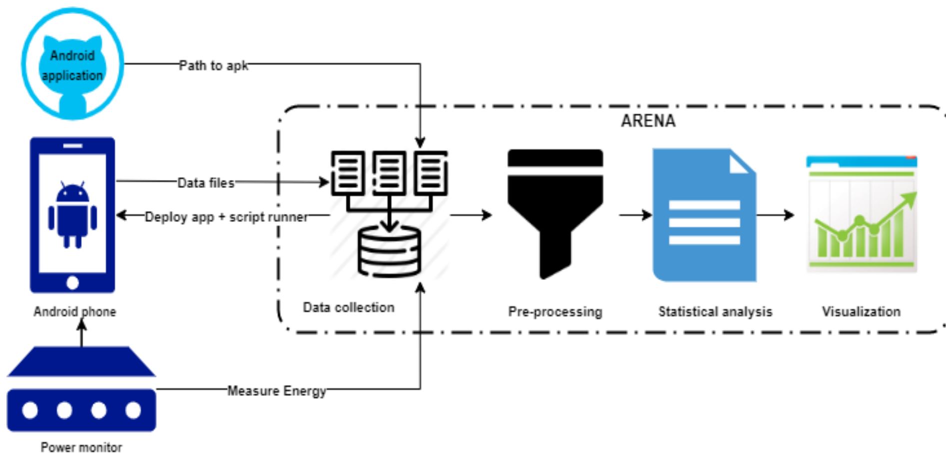


Figure 7.1: An overview of the energy measurement and analysis process that is supported by ARENA

## 7.1. ARENA Architecture

In this section, we describe how ARENA supports the energy measurement and analysis process. Typically energy measurement and analysis process consist of four steps: 1) Data collection, 2) Data pre-processing, 3) Statistical Analysis and 4) Visualization. For each of the main steps in this process, there exists a corresponding component in ARENA. Figure 7.1 gives an overview of the energy measurement and analysis process that is supported by ARENA.

[3]https://tqrg.github.io/physalia/

### 7.1.1. Component 1: ExperimentRunner

The ARENA component ‘ExperimentRunner’ supports the first step of the energy measurement and analysis process, i.e., ‘Data collection’. During data collection, energy measurements are recorded several times to account for the underlying variations in energy consumption due to background processes in the mobile device. The energy measurements are controlled from the host computer via ‘ExperimentRunner’, which helps ARENA user to control activities required to set up and execute the experiment for data collection. ‘ExperimentRunner’ initializes the callback object to display output in the tool window. It also creates a directory where customized shell scripts will be created.

*Experiment Setup.* Before using the ARENA tool, the scope of the experiment, measurement environment settings, an AUT, and test scripts are prepared by the ARENA user. R version 3.4.3 or above and Rtools needs to be installed on the host computer. To measure hardware-based energy readings, ARENA is designed to be used with the Monsoon Power Monitor, a popular physical measurement device that has been used in various studies [47, 96, 140]. Therefore related libraries and python packages[4] need to be installed as per the user manual of Monsoon Power Monitor. The mobile phone on which the AUT will be executed should have Android 5.0 and above. The mobile phone is connected to the host computer with a USB cable via Monsoon Power Monitor disabling the USB phone charging once the energy measurement starts. ARENA user should check that the screen brightness is set to a minimum and only essential Android services are running on the phone. ‘ExperimentRunner’ checks if the mobile device is successfully connected to the host computer. Additionally, a script ‘start_power_monitor.py’ is generated, which initializes Monsoon Power Monitors’ runtime current limits, and enables the USB channel on Monsoon Power Monitor hardware. The serial number of Monsoon Power Monitor hardware can be configured in the ‘start_power_monitor.py’ script in the ARENA source code. ‘ExperimentRunner’ works with only one Monsoon Power Monitor at a time.

*Experiment Execution.* ‘ExperimentRunner’ creates customized scripts for experiment execution. These scripts include commands to clear battery statistics, memory statistics, network statistics and adb log files before each iteration of the experiment. We consider an iteration to be the execution of a test case on a mobile device once. Based on the AUT selected by the ARENA user, commands in shell scripts are customized to install and run the app (for baseline readings, these commands are not included). The script ‘start_power_monitor.py’ creates an instance of the sample engine class from the Monsoon Power Monitor Python library. By default, the current/voltage samples are saved as a Python list that can be retrieved with the getSamples() function. At the end of each iteration, the Python list is converted into a CSV file and saved on the host computer. Monsoon Power Monitor

[4] https://figshare.com/s/9cdfc9f8b39411698afd

output files are checked for reliability based on the number of dropped samples. As Monsoon Power Monitor records samples at a rate of 5KHz, and assuming that AUT runs for more than one second, the ARENA user is given a warning to check current/voltage data if 1000 or more samples are dropped.

Once all scripts are ready, they are pushed to the mobile device along with the script-runner apk. Script-runner is a small app that comes with ARENA to automatically trigger AUT and related shell scripts on the mobile device. This app is a necessary overhead to save manual effort and to ensure that no problems are created during the experiment due to human error. 'ExperimentRunner' gives the options to the ARENA user to re-run the same iteration, run the next iteration, uninstall AUT from the mobile device, and clear data for AUT from the mobile device. The selected option is passed as a runtime argument to the script-runner app, which executes the relevant shell script on the mobile device.

After each iteration, data is retrieved from the device to the result folder on the host computer and files are renamed as per iteration number. e.g. for the first iteration, the adb log file "logcat.txt" is renamed to "Logcat_R1" and so on. During the next iterations, settings are updated in the shell scripts (if needed).

### 7.1.2. Component 2: CleanupRunner

The ARENA component 'CleanupRunner' supports the second step of the energy measurement and analysis process, i.e., 'Data pre-processing'. 'Cleanup-Runner' renders the raw data files in a list in the tool window and performs cleaning/filtering on the selected files. PID (process-ID) and UID (user-ID) are extracted from the adb log files for each iteration of the experiment. The UID is used to extract relevant data from network statistics files. CPU and memory statistic files are filtered by app package name. For cleaning adb logs, UID, PID, and user-specified tags are used. As the format of adb log and statistic files in different Android versions are slightly different, to produce cleaned output files with a consistent format, the API version of the mobile device is used to implement the correct parser on the log and statistic files. Once the adb log and statistic files are cleaned, the timestamps from the cleaned adb log file are used to extract relevant current/voltage data in each iteration. The cleaned current/voltage file is used to calculate energy consumption in joules (J) of AUT in each iteration. An average of baseline energy is subtracted from the calculated energy consumption of AUT (under the assumption that an increase in energy consumption from the baseline is due to the execution of AUT). A data file named 'data.csv' is created containing the package name of AUT, energy (J), memory %, CPU % and network statistics for each iteration. Another file named 'average_data.csv' is created with aggregated values for energy (J), memory %, CPU % and network statistics of all iterations of AUT.

### 7.1.3. Component 3: AnalysisRunner

The ARENA component 'AnalysisRunner' supports the third step of the energy measurement and analysis process, i.e., 'Statistical Analysis'. 'AnalysisRunner' populates the combo boxes for dependent, independent and filter variables in the tool window with column names from the selected CSV data files (the cleaned energy files produced by 'CleanupRunner' are used here). 'AnalysisRunner' provides detailed help text in the tool window in order to make it easier for the user to select a statistical analysis based on requirements and data type. Based on the ARENA user selection, the values of variables included in the analysis are updated in the relevant R scripts, which are then executed to produce a report containing the results of the selected statistical analysis and its interpretation.

### 7.1.4. Component 4: VisualizationRunner

The ARENA component 'VisualizationRunner' supports the fourth step of the energy measurement and analysis process, i.e., 'Visualization'. 'VisualizationRunner' populates the combo boxes for dependent, independent and filter variables in the tool window with column names from the selected CSV file. The type of the graph selected and the dependent, independent, and filter variables control how the data in the graph is displayed. 'VisualizationRunner' allows various graph configuration for each graph type in terms of label font, legend colours, graph title, graph size, sequence of label on x-axis etc.

## 7.2. ARENA Implementation

ARENA is built for integration with IntelliJ IDEA and Android Studio IDE as a plugin. Functionalities of the plugin are implemented on widgets of the tool window (from here onwards referred to as ARENA interface). The plugin is implemented in Java. Each component in ARENA's architecture corresponds to a tab on the ARENA interface. Based on the scope and requirements of the experiment, the ARENA user can set certain parameters on each tab to get the results. It is ideal to use the tabs in the ARENA interface iteratively as they are interrelated. However, if ARENA users wants to reuse data of a particular process step or skip a process step, that is also permitted.

The first tab in the ARENA interface is 'Data collection', which corresponds to the ARENA component 'ExperimentRunner'. Within this tab, ARENA users can perform two sets of activities, 1) configure experiment setup by selecting energy measurement mode, data collection phase and corresponding data files, and 2) control the experiment execution by configuring the experiment parameters such as number of iterations (a single iteration is the execution of test apk once, the choice of this value depends on the requirement of the experiment, however for the sake of sampling distribution a value between 10-30 is considered good), path to app apk, path to test apk, data path on mobile device, test class, test runner, re-run

configuration (i.e., re-install app or clear data), results folder etc. The main output of this tab is the raw timestamped current/voltage data from Monsoon Power Monitor and corresponding adb logs and statistics from the mobile device.

The second tab in the ARENA interface is 'Pre-processing', which corresponds to the ARENA component 'CleanupRunner'. The main outputs of this tab are the 1) filtered current/voltage data, adb logs and statistics[5], and 2) calculated and aggregated energy and statistics data[6].

The third tab in the ARENA interface is 'Analysis', which corresponds to the ARENA component 'AnalysisRunner'. The main output of this tab is a report(s) in .docx format with results of statistical analysis about the energy consumption of AUT (along with its interpretations).

The fourth tab in the ARENA interface is 'Visualization', which corresponds to the ARENA component 'VisualizationRunner'. The main output of this tab is the graph of the selected type.

In all the tabs hovering the mouse pointer on a widget of the interface shows a tool-tip with help text. Progress and error messages are shown either in the tool window terminal or via error labels.

The implementation of ARENA is available at our bitbuket repository[7].

## 7.3. ARENA in Practice

This section ties together all components described above and provide example usage. ARENA can be installed as an IntelliJ or Android Studio plugin using the package we provide on our bitbucket repository[8]. After installation, when a user opens a new or existing project, they can see the ARENA tab on the right side of the IDE.

### 7.3.1. Comprehensive Usage Scenario

In this section we present a comprehensive usage scenario in which all tabs in ARENA are used. It is ideal to use the tabs in the ARENA interface iteratively as they are interrelated. However, if ARENA users wants to reuse data of a particular process step or skip a process step, that is also permitted. The comprehensive usage scenario of ARENA is also shown in a YouTube video[9].

[5] Details of columns in filtered data file https://figshare.com/s/50c5732300315023b197

[6] Details of columns in aggregated data file https://figshare.com/s/cbc2fd529b413e4dcbf1

[7] ARENA source code, https://bitbucket.org/hinaanwar2003/arena

[8] ARENA is packaged as a plugin, which could be installed via zip file EnergyPlugin-1.0-SNAPSHOT.zip available at https://bitbucket.org/hinaanwar2003/arena/src/master/. See Annex C for installation instructions.

[9] ARENA comprehensive usage scenario YouTube video: https://www.youtube.com/watch?v=hgP5XL9SvRU

The comprehensive usage scenario of ARENA begins with the source code of the AUT. The developer writes automated Android user interface tests for AUT using tool such as Espresso[10]. Next, the developer wants to assess AUT's energy consumption to compare with the previous version of the same app or against a competitor app. The ARENA interface facilitates the developer to measure, aggregate, analyse and visualize the energy consumption of AUT. Using the 'Data collection' tab, the energy data collection process is initiated. The corresponding adb logs and additional statistics (if selected) such as CPU, memory, network statistics and trace files are recorded and extracted from the mobile device. The energy data from Monsoon Power Monitor is automatically saved as a CSV file on the host computer. Using the 'pre-processing' tab, the raw energy data is cleaned and aggregated by matching it against the start and end timestamps found in the adb log files. Using the 'Analysis' tab, various statistical analysis (such as Summary statistics, Kruskal-Wallis, Spearman Correlation, ANOVA etc.) could be performed on the data. After analysis is complete, a detailed report of the analysis and the interpretation of the results are generated (in .docx format). Using the 'Visualization' tab, the data could be visualized by creating various graphs (such as scatter plot, box plot). In Figure 7.2. we show the detailed workflow with included sub-steps supported by ARENA tool. Figures 7.3, 7.4, 7.5, and 7.6 shows the corresponding ARENA interfaces[11] in IntelliJ IDE.

### 7.3.2. Application Example

Application example used the comprehensive scenario described above. We used ARENA in a study [154] to evaluate the energy consumption of commonly used third-party network libraries in Android apps. We made 45 versions of a custom app using selected third-party network libraries in different use cases. We used ARENA to measure the energy consumption of each version of the custom app by executing it on an Android device ten times. We recorded 450 energy measurements along with corresponding adb logs. Next, the recorded data was cleaned, aggregated, analyzed and visualized using ARENA to identify the statistically significant changes in energy consumption of different third-party network libraries in different use cases. ARENA significantly reduced the time and effort required to measure and analyze the energy consumption of AUT. As the process described was controlled via ARENA, thus errors in measurement due to human error were also avoided.

[10]https://developer.android.com/training/testing/espresso

[11]See the detailed tool tutorial: https://figshare.com/s/4c4ec26fc0ec91fbad41

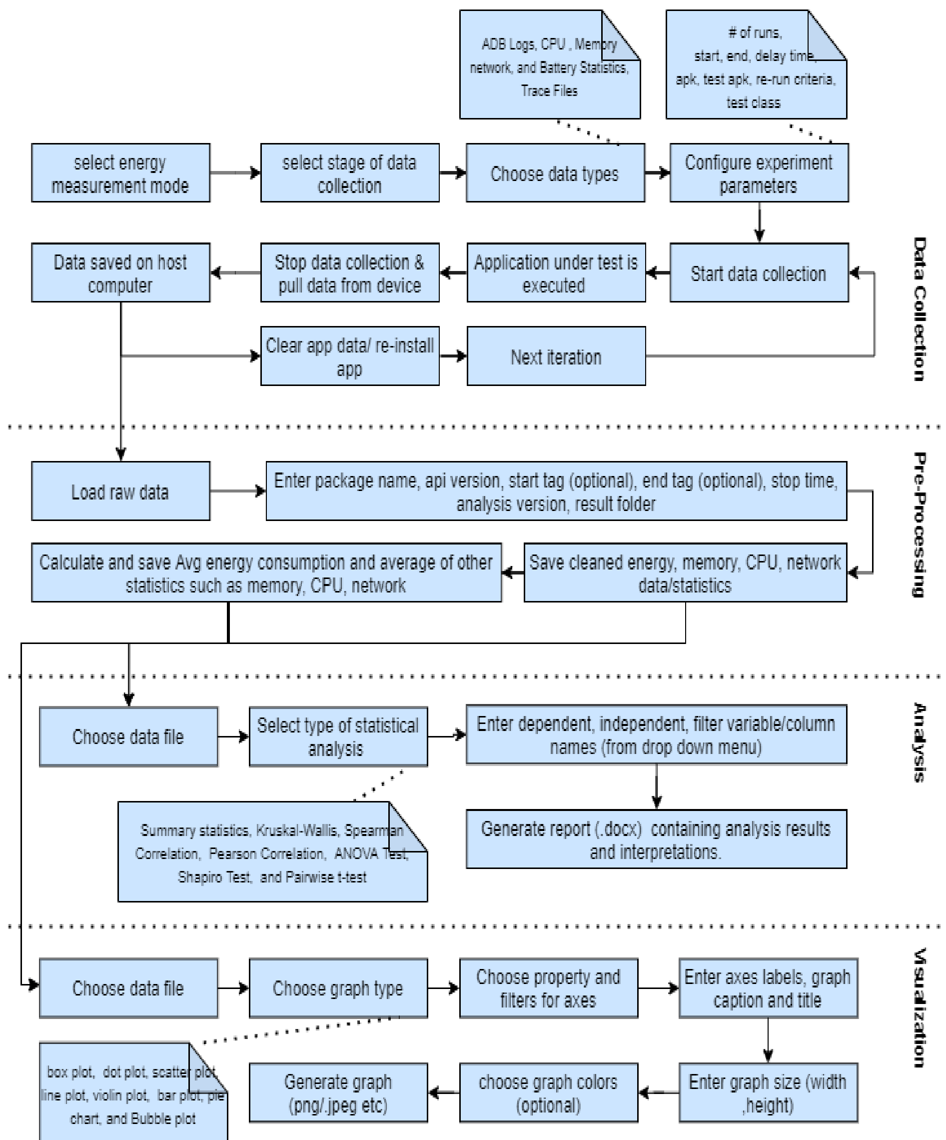


Figure 7.2: Detailed workflow supported by ARENA tool

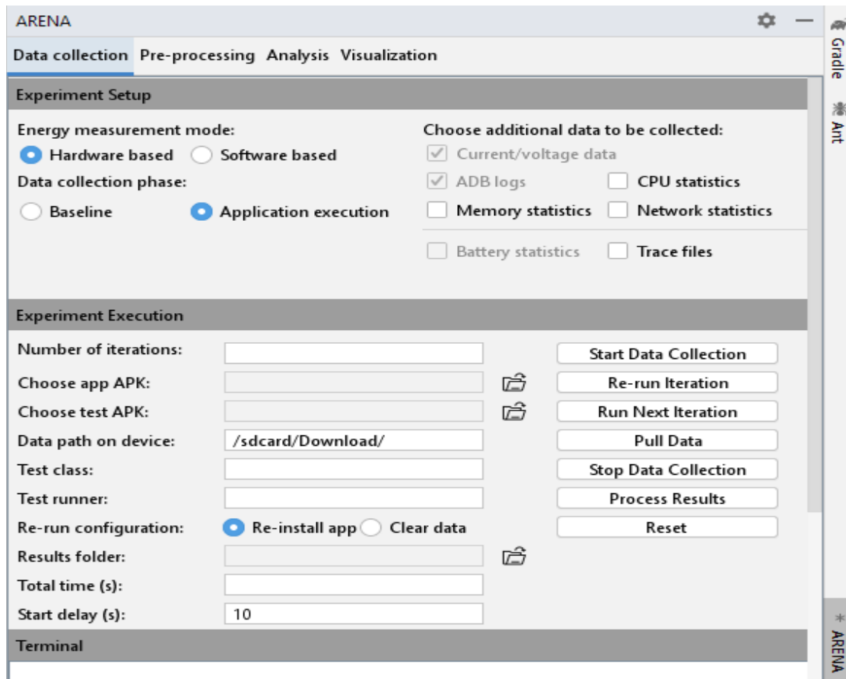


Figure 7.3: ARENA interface - Data collection tab

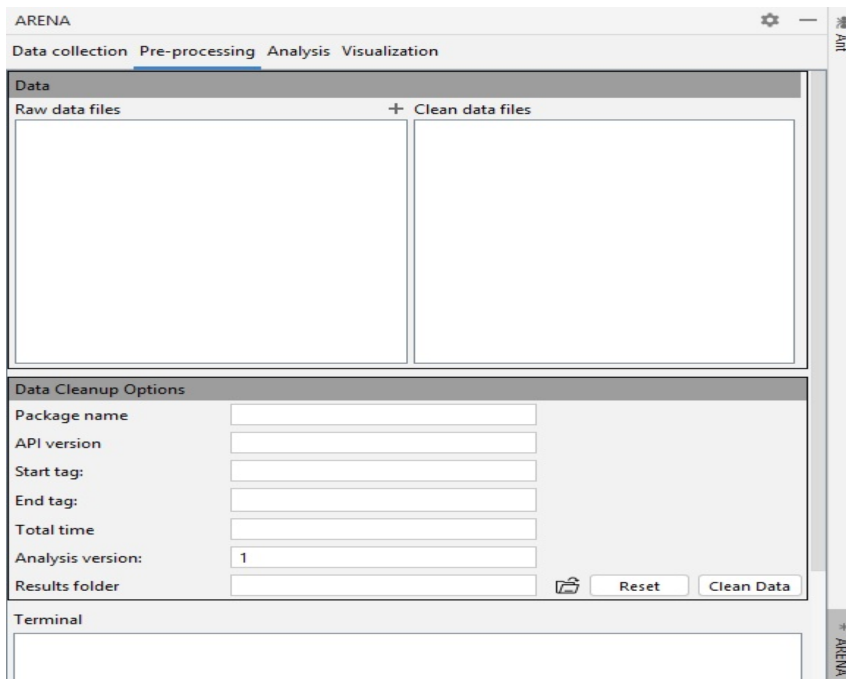


Figure 7.4: ARENA interface - Pre-processing tab

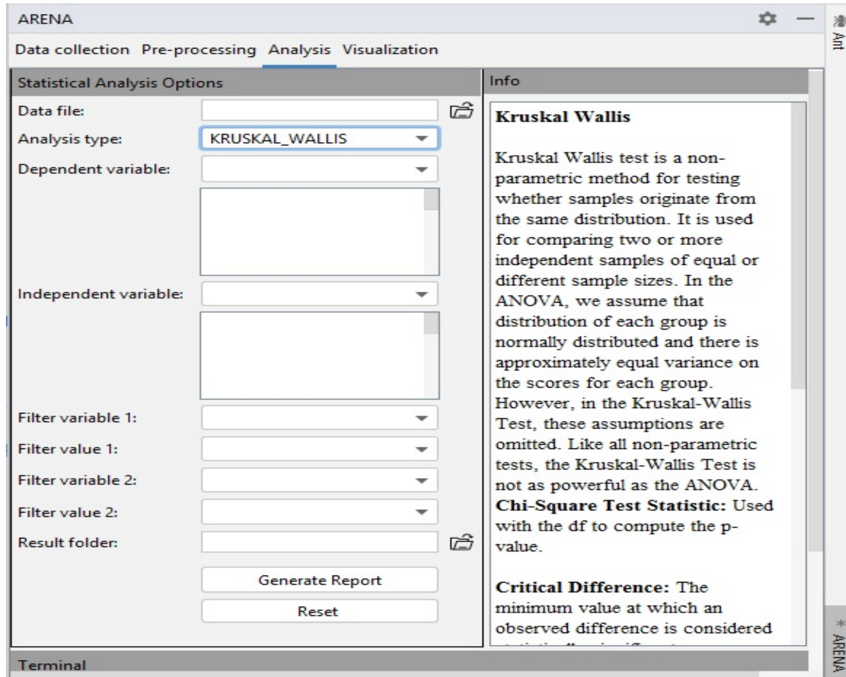


Figure 7.5: ARENA interface - Analysis tab

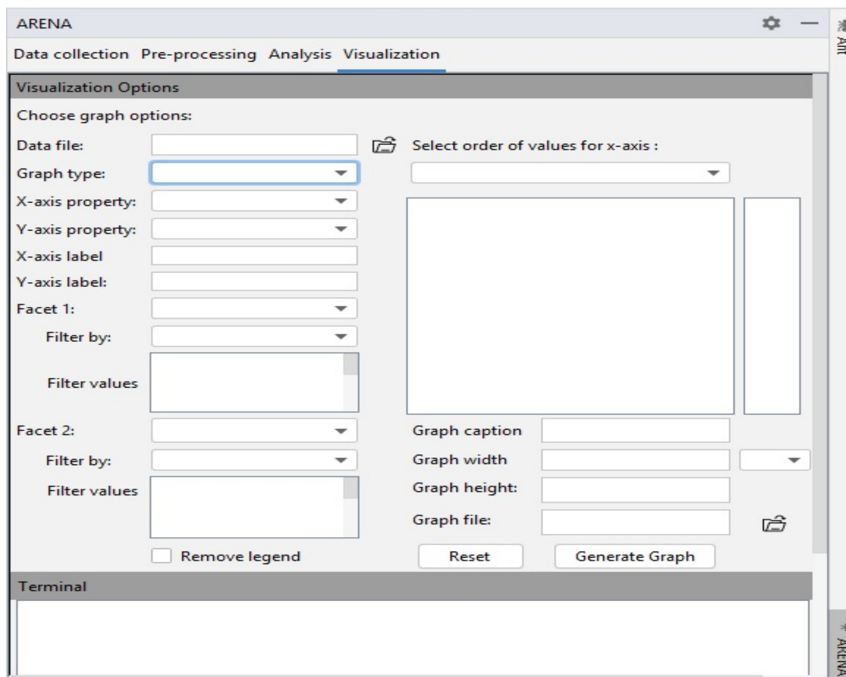


Figure 7.6: ARENA interface - Visualization tab

# 8. REHAB: A TOOL FOR RECOMMENDING ENERGY-EFFICIENT THIRD-PARTY LIBRARIES TO ANDROID DEVELOPERS

In this chapter, we focus on RG2 (defined in Section 1.1) as well to improve the tool support for energy efficient mobile app development. In chapter 6, we provided an overview of state-of-the art and compared the current available support tools to aid green Android development. In chapter 7, we presented a new support tool for measuring energy of Android apps. The work presented in this chapter is an extension of the study presented in chapter 5, in which we measured the energy consumption of commonly used third-party HTTP libraries for Android apps. We measured energy consumption for one version per library within each selected use case. Now, we present a recommender system **REHAB** (**R**ecommending **E**nergy-efficient t**H**ird-p**A**rty li**B**raries) to assist developers in choosing energy-efficient third party HTTP libraries in specific use cases. Further, we discuss how the scope of REHAB can be widened by conducting usage and change analysis on 8457 Android apps. The usage analysis quantifies the usage of selected third-party HTTP libraries and all their versions in a set of real Android apps. The change analysis quantifies what kind of changes are made in the consecutive versions.

Mobile users prefer longer battery life. If an app is causing battery drain it usually receive negative reviews from the users. Energy of mobile apps could be improved by minor code optimizations [30, 71, 90, 125]. Android developers use third-party libraries to speed up the development. Over 60% of Android apps' code is contributed by commonly used libraries [168]. The choice of using third-party libraries in the Android app is crucial as developers do not tend to update them frequently [147]. For a particular task usually many alternatives are available that offer similar functionalities. If the third-party library included in the app is not energy efficient it might drain the mobiles' battery. We seek to help Android developers by recommending energy efficient third-party HTTP libraries during development.

## 8.1. REHAB Architecture

In this section we describe how REHAB recommend energy efficient third-party HTTP libraries to the developers. We developed a rule based system that uses pre-defined rules to make recommendations to the user. Based on the energy consumption data of selected third-party HTTP libraries, a finite set of rule (conditions and related actions) are defined. REHAB does not automatically make changes in code. Based on conditions it recommends related action. REHAB consist of three main components: 1) knowledge base, 2) Code inspection module, 3)

Rule mapping module. Figure 8.1 gives an overview of the REHAB Architecture.

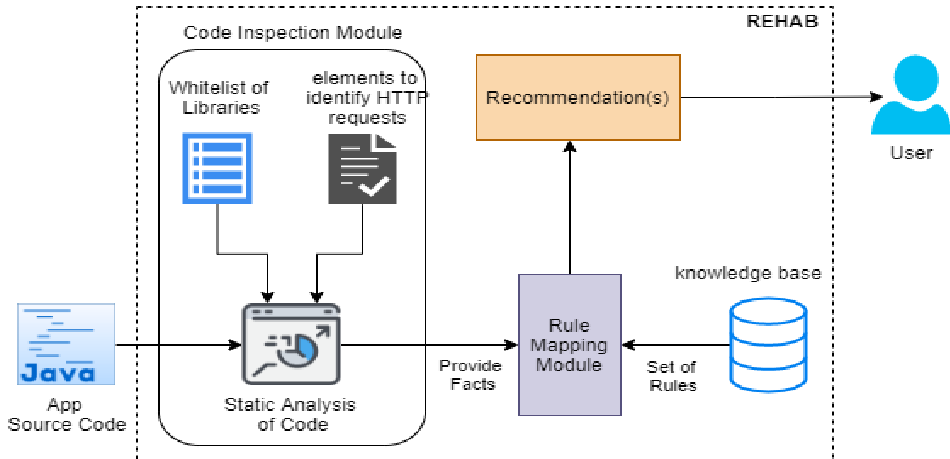


Figure 8.1: An overview of REHAB

### 8.1.1. Component 1: Knowledge Base

To define rules, we used energy data produced in chapter 5 [154]. The selected use-cases are as follows:

- **UC-GF:** Making an HTTP GET request to the server to download a file and logging the server's response.
- **UC-PF:** Making a multipart HTTP POST request to the server to upload a file and logging the server's response.
- **UC-PJO:** Making a multipart HTTP POST request for sending Java objects serialized as JSON to the server and logging the server's response.
- **UC-GJO:** Making an HTTP GET request to the server for receiving JSON Objects, de-serializing them and mapping them to the Java objects.
- **UC-GI:** Making a GET request to the server to load images and displaying them on screen.

The third-party selected libraries used within each use-case are summarized in the Table 8.1 (taken from chapter 5).

In chapter 5, we have calculated the energy consumption of libraries in individual use cases, therefore, we cannot tell how much energy efficient combined usage of libraries are in all possible combinations. We can make a recommendation if user wants to use a single library for more than one use case. At the moment, we cannot recommend combination of third-party libraries that requires additional energy data. We made 31 unique combinations of use-cases (see Table 8.2) in total. For each single/combination of use case we recommend library alternatives (assuming that the probability of inclusion of each library alternative in an app is

Table 8.1: Android third-party HTTP libraries used in each selected use case.

| ID | Library | UC | | | | |
|---|---|---|---|---|---|---|
| | | GF | PF | PJO | GJO | GI |
| vo | Volley | | x | | | x |
| vo(G) | Volley(G) | | | x | x | |
| vo(M) | Volley(M) | | | x | x | |
| vo(J) | Volley(J) | | | x | x | |
| re | Retrofit | x | x | | | |
| re(G) | Retrofit(G) | | | x | x | |
| re(M) | Retrofit(M) | | | x | x | |
| re(J) | Retrofit(J) | | | x | x | |
| ok | OkHttp | x | x | | | |
| ok(G) | OkHttp(G) | | | x | x | |
| ok(M) | OkHttp(M) | | | x | x | |
| ok(J) | OkHttp(J) | | | x | x | |
| async-h | Androidasynchttp | x | x | | | x |
| async-h(G) | Androidasynchttp(G) | | | x | x | |
| async-h(M) | Androidasynchttp(M) | | | x | x | |
| async-h(J) | Androidasynchttp(J) | | | x | x | |
| async | Androidasync | x | x | | | |
| async(G) | Androidasync(G) | | | x | x | |
| async(M) | Androidasync(M) | | | x | x | |
| async(J) | Androidasync(J) | | | x | x | |
| pic | Picasso | | | | | x |
| uil | UIL | | | | | x |
| gli | Glide | | | | | x |

equal). In Table 8.2 text in blue indicates the number of library choices available for that particular use case/combination of use cases.

The rules are based on mean energy consumption(joules) of each library and loss in energy calculated for each library within each use case using Minimax strategy. Minimax strategy is commonly used in game theory where the objective is to minimize loss and maximize the profit. In our case profit = most energy efficient. We select the libraries that are most energy efficient with minimum loss. To calculate loss, within each use case we select the library with smallest mean energy value (J) and we subtract it from all other alternatives. In each use case we have five choices of libraries. We select one third-party library in each use case that is most energy efficient ( with minimum loss within a use-case) for implementing that use case in an app. To recommend a library for a combination of use-cases, we rank the libraries based on the total minimum loss value for that combination.

Table 8.2: Unique combinations of use-cases.

| 1 UC | 2 UC | 3 UC | 4 UC | 5 UC |
|---|---|---|---|---|
| GF (5) | GF,PF (5) | GF,PF,PJO (15) | GF,PF,PJO,GJO (15) | GF,PF,PJO,GJO,GI (6) |
| PF (5) | GF,PJO (15) | GF,PF,GJO (15) | GF,PF,PJO,GI (6) | |
| PJO (15) | GF,GJO (15) | GF,PF,GI (2) | GF,PF,GJO,GI (6) | |
| GJO (15) | GF,GI (2) | GF,PJO,GJO (15) | GF,PJO,GJO,GI (6) | |
| GI (2+3) | PF,PJO (15) | GF,PJO,GI (6) | PF,PJO,GJO,GI (6) | |
| | PF,GJO (15) | GF,GJO,GI (6) | | |
| | PF,GI (2) | PF,PJO,GJO (15) | | |
| | PJO,GJO (15) | PF,PJO,GI (6) | | |
| | PJO,GI (6) | PF,GJO,GI (6) | | |
| | GJO,GI (6) | PJO,GJO,GI (6) | | |
| **45** | **96** | **86** | **39** | **6** |

Table 8.3 shows an example of the calculated minimum loss values indicating loss in energy for selected third-party HTTP libraries in UC-GF and UC-PF. The last column shows the ranking of third-party libraries based on the loss values of both UC-GF and UC-PF. For example, within a single use case such as UC-GF, based on minimum loss value AndroidAsync library will be recommended to the REHAB user. For a combination of use-cases, for each third-party library we add the minimum loss calculated for that library within each use-case, and rank (1 = most energy-efficient, 5 = least energy-efficient) the third-party libraries. The ranking is shown to REHAB user so that they can make an informed decision.

Table 8.3: Based on mean energy values the calculated minimum energy loss values are shown for selected third-party HTTP libraries in UC-GF and UC-PF

| | | UC | | |
|---|---|---|---|---|
| ID | Library | GF | PF | Ranking |
| vo | Volley | 15.57 | 43.53 | 5 |
| re | Retrofit | 6.18 | 0 | 2 |
| ok | OkHttp | 5.59 | 3.38 | 3 |
| async | AndroidAsync | 0 | 1.5 | 1 |
| async-h | AsyncHttp | 14.01 | 4.27 | 4 |

### 8.1.2. Component 2: Code Inspection

Code inspection module contains the whitelist of selected third-party HTTP libraries, elements for detecting different types of HTTP requests and *Action_DetectTPL* class. Static analysis is perfomed on the source by *Action_DetectTPL* class upon user action (i.e., when user click 'REHAB - Detect Third-party Libraries' option form 'Tools' menu in IDE). Code inspection module detect third-party HTTP libraries and their call expression and methods included in the project. Project refers to an Android Studio or IntelliJ Gradle based

project. Whitelist in the code inspection module contains the base package name and version number for all the select third-party HTTP libraries. Elements for detecting different types of HTTP requests are referred in the following sections as TypeRequestRules.

First, we collect all the libraries included in the project from Gradle file, that may be added to dependencies of the corresponding projects modules. We match the collected project-level libraries with all the selected third-party HTTP libraries in the whitelist. If a match is found,it is displayed in the tool window. Second, for each detected third-party HTTP library, we search for all the JAVA source files in project that are dependent on that third-party library. We match import statements in each source file with the base package of selected third-party HTTP libraries. Base package name of a third-party library in import statements is enough to find all the classes added in a source file from that library. Third, based on TypeRequestRules, we statically analyze the source files to detect methods definitions, method calls, fields, and parameter for each library. Detected results are shown to REHAB user in tool window.

### 8.1.3. Component 3: Rule Mapping Module

In rule mapping module the identified methods definitions, method calls and variables for each third-party library are saved as facts. These facts are then matched against the knowledge base. If a match is found i.e., the third-party library name and the corresponding HTTP request type are same in fact and in pre-defined rules, then the corresponding recommendation for single use case and combination of use cases is shown to REHAB user in the tool window.

## 8.2. REHAB Implementation

REHAB is implemented as an IntelliJ and Android Studio plugin. The implementation is done in JAVA. Within the knowledge base module, the *RecLibAlternatives* class reads the energy data from excel sheet and calculates the minimum energy loss value for each library with each use case and rank the libraries for combination of use cases. Within the code inspection module, *Action_DetectTPL* class detect third-party HTTP libraries and their call expression and methods included in the project. *Action_DetectTPL* class uses *JavaRecursiveElementVisitor* to analyze all the source files included in the project for patterns. *JavaRecursiveElementVisitor* belongs to the build in *PsiElementVisitor* class, which can be used to visit elements in PSI[1] (program structure interface) tree in a programming language.

[1]PSI is a root structure containing contents of a file as hierarchy. Static analysis could be performed on the source code by accessing individual PSI elements. For more details on the topic of PSI trees and files see https://plugins.jetbrains.com/docs/intellij/psi.html

To detect and highlight the source code lines related to different types of HTTP requests, we defined TypeRequestRules. Each TypeRequestRules consists of attributes (see Table 8.4) to identify a particular type of HTTP request such as GET, POST etc. Third-party libraries are used in source files via annotations, identifiers, method calls, client Builders and response callbacks and serialization methods. Not all attributes are required in every HTTP request.

We extract the PSI tree and elements (such as Psimethods, Psivariables, PsiMethodCallExpression etc.) for each source file and match it against the TypeRequestRules to detect and highlight the code relevant to HTTP requests. If a TypeRequestRule is satisfied than the third-party library name and HTTP request type within the TypeRequestRule is matched against the knowledge base and recommendation is shown to the REHAB user.

Table 8.4: Elements for detecting different types of HTTP requests

| **Elements** | **Description** |
|---|---|
| requestClass | Method call or variable for request e.g.com.loopj.android.http.RequestHandle |
| requestIdentifier | Keyword to identify method call depending on the library e.g. "GET", "POST" for volley. |
| typeAnnotations | Annotations that must be present in an HTTP request e.g. retrofit2.GET, retrofit2.POST etc. |
| notTypeAnnotations | Annotations that must not be present in an HTTP request. |
| bodyClass | The request body object that must be present such as to differentiate between multipart body object from simple body object. |
| notbodyClass | The request body object that must not be present in HTTP request. |
| clientBuilderClass | Client builder class used in HTTP request e.g. retrofit2.Retrofit |
| builderIdentifier | Identifier to specify that client is created e.g. "build" or "getInstance" etc. |
| builderAttributes | List of strings that must be present in client builder definition e.g. GsonConverterFactory |
| notbuilderAttributes | List of strings that must not be present in client builder definition e.g. JsonConverterFactory |
| serializerClass | Serialization builder required for HTTP request e.g. com.squareup.moshi.Moshi |
| notSerializerClass | Serialization builder that must not be present in an HTTP request e.g. com.squareup.moshi.Moshi |
| responseCallbackClass | The Response Callback class e.g. retrofit2.Callback |

### 8.2.1. Usage Overview

REHAB can be installed as an Intellij or Android Studio plugin using the package provided in our bitbucket repository [2]. After installation, when a user opens a new or existing project, they can see the REHAB tab on the right side of the IntelliJ or Android Studio IDE. Figure 8.2 shows the tool window that will appear when user click on the REHAB tab. In this figure we have assigned numbers to main components in the tool window for easy reference. The REHAB user can hover the mouse pointer on any of the tool window components to get a short description about them. As third-party libraries are not so frequently updated in source code therefore to avoid unnecessary computational overhead we do not run REHAB constantly in the background. Instead, to detect third-party HTTP libraries included in the whitelist of REHAB and see the related recommendation, REHAB user can click on 'REHAB - Detect Third-party Libraries' from the 'Tools' drop down menu in IntelliJ/Android Studio IDE. The first part of the tool window (marked by number 1 in Figure 8.2) shows the detected third-party HTTP libraries in the source code. For each of the detected third-party libraries the second part of the tool window (marked by number 2 in Figure 8.2) shows related method definitions (if any). Similarly, for each of the detected third-party libraries the third part of the tool window (marked by number 3 in Figure 8.2) shows related method calls (if any). Clicking any of the identified method definitions/method call

- highlights the related source code and the name of the source file containing the highlighted code is displayed in fifth part of the tool window (marked by number 5 in Figure 8.2),
- the recommendation for alternative energy-efficient third-party Library that can make similar HTTP request is also shown in the fifth part of the tool window,
- a ranking of alternative energy-efficient third-party libraries for the detected HTTP request(s) is shown in part sixth part of the tool window (marked by number 6 in Figure 8.2).

Figure 8.3 shows an example of a recommendation made by REHAB. Based on the provided information in the tool window, REHAB users can make an informed decision on whether or not they would like to change the current third-party HTTP library. The fourth part of the tool window (marked by number 4 in Figure 8.2) can used to export the results shown in REHAB tool window to a CVS file in USER.HOME.

[2] REHAB is published for others to use at https://bitbucket.org/hinaanwar2003/rehab/src/master/

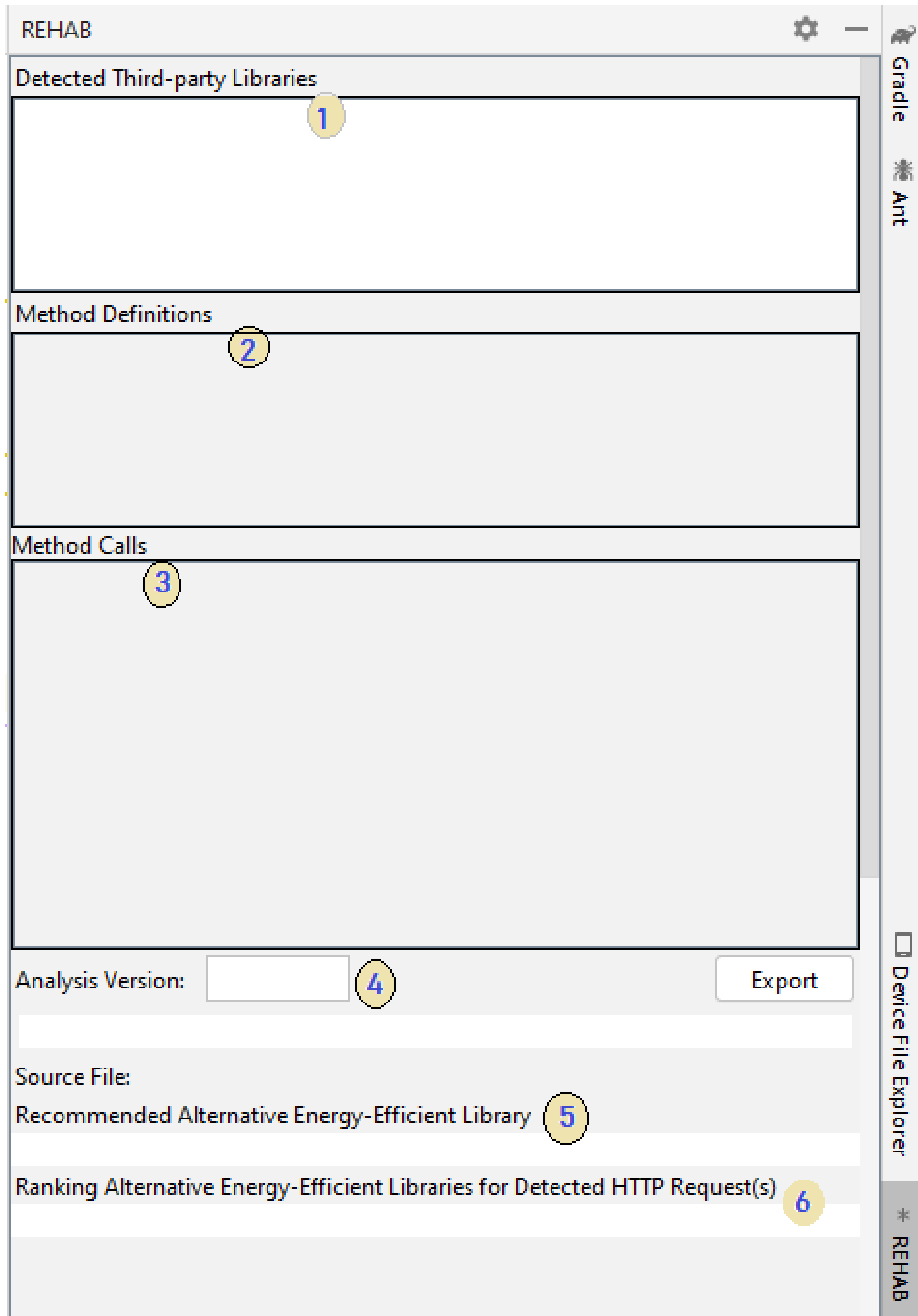

Figure 8.2: REHAB tool window

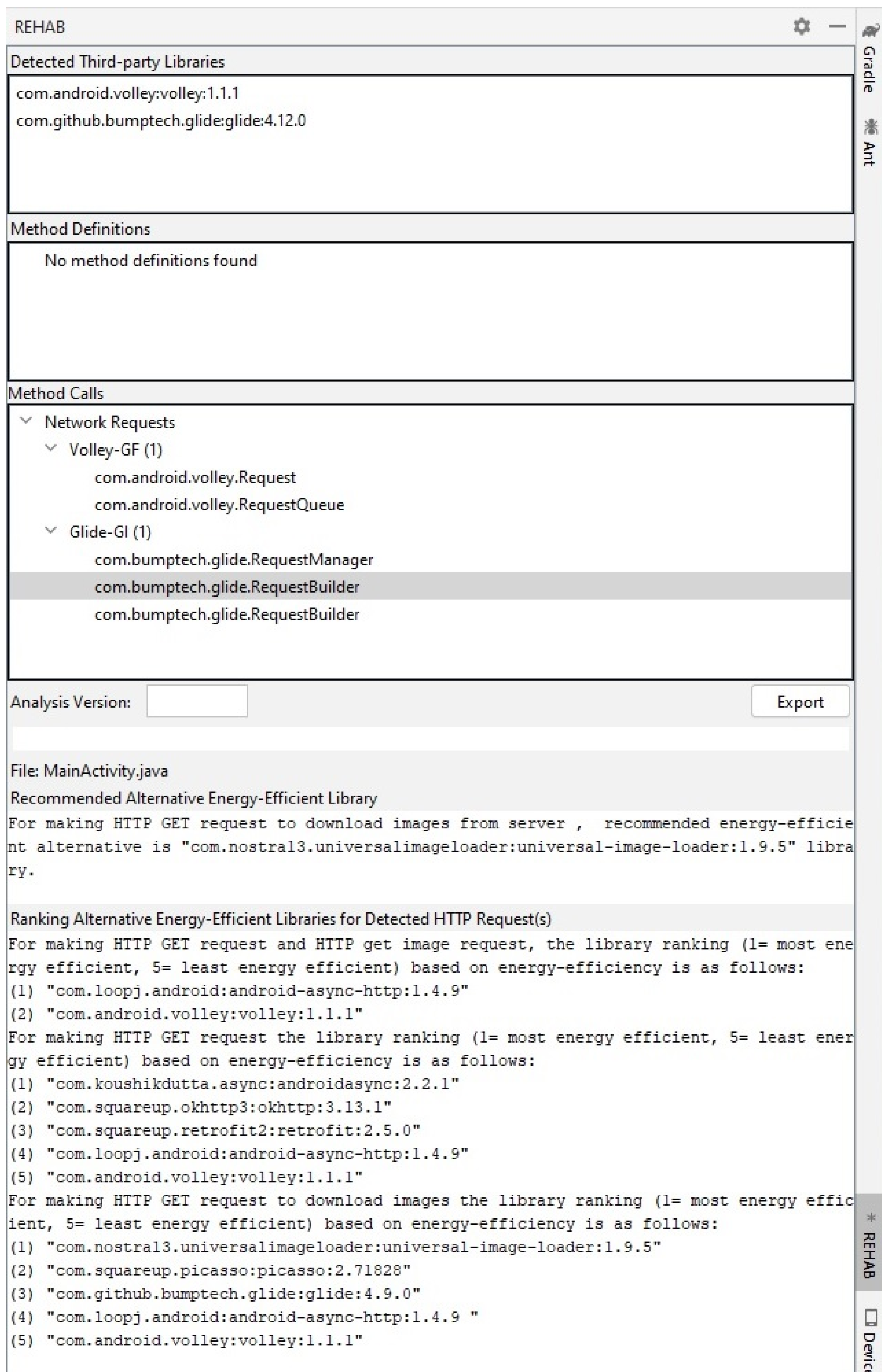


Figure 8.3: REHAB usage example

## 8.3. Scope Extension of REHAB

In this section, we discuss if the recommendations shown by REHAB for particular version of selected third-party HTTP libraries could also be applied to other versions of the same library. For this purpose, we conducted usage and change analyses on Android apps. Each selected third-party HTTP library has many versions available in MVN-central library. The usage analysis quantifies the usage of selected third-party libraries and all their versions in a set of real Android apps. The change analysis quantifies what kind of changes are made in the between consecutive versions.

Collecting Android APKs from online markets is a time consuming process, therefore, we choose Androzoo dataset[3] which contains 15,730,282 APKs (as of june 2021) collected from different app stores. For each APK, Androzoo dataset contains its SHA256, SHA1, md5, apk_size dex_size,pkg_name, market etc. App APK could be downloaded from a remote server via a combination of SHA256 key and API key. From Androzoo dataset we filtered out APKs that belonged to F-Droid[4] and Google Playstore[5]. We selected F-Droid because it contains free open source Android apps. We selected Google Playstore because it the official online market for publishing Android apps. We further filtered the apps using SHA256 key to ensure that for each app only the latest version of that app is included. As manually downloading such a huge number of apps one by one is a time consuming task, therefore we wrote python scripts to automate the process. After filtering apps via SHA25 keys we had 3457 unique SHA256 keys for apps from F-Droid repository, and 4388689 unique SHA256 keys for apps from Google play store. We downloaded 3457 app APKs belonging to F-Droid and 5000 app APKs belonging to google play.

For the usage and change analysis, we used LibScout[6] (version 2.3.2), a static analysis tool, to detect selected third-party libraries in Android apps APK. LibScout requires the original library SDKs (compiled .jar/.aar files) to extract library profiles that can be used for detection on Android apps. LibScout can detect library versions and its APIs based on semantic versioning. For each app APK a JSON report was made containing app name, package name, used libraries and versions. Using python scripts we parsed the JSON files to CSV for further processing in Excel.

For each third-party library, there are several versions available in the MVN-central repository. Table 8.5 gives an overview of the number of versions available for each third-party library. Figure 8.4 shows number of versions per change type

[3]https://androzoo.uni.lu/

[4]https://www.f-droid.org/

[5]https://play.google.com/store/apps

[6]https://github.com/reddr/LibScout

per library. The type of change in versions of a library were detected using LibScout tool.

Table 8.5: Number of versions per library available in MVN-central repository (as of January 2021).

| ID | Library | Number of versions |
|---|---|---|
| vo | Volley | 3 |
| re | Retrofit | 20 |
| ok | OkHttp | 64 |
| async | AndroidAsync | 65 |
| async-h | AsyncHttp | 9 |
| gli | Glide | 26 |
| pic | Picasso | 22 |
| Uil | UIL | 15 |

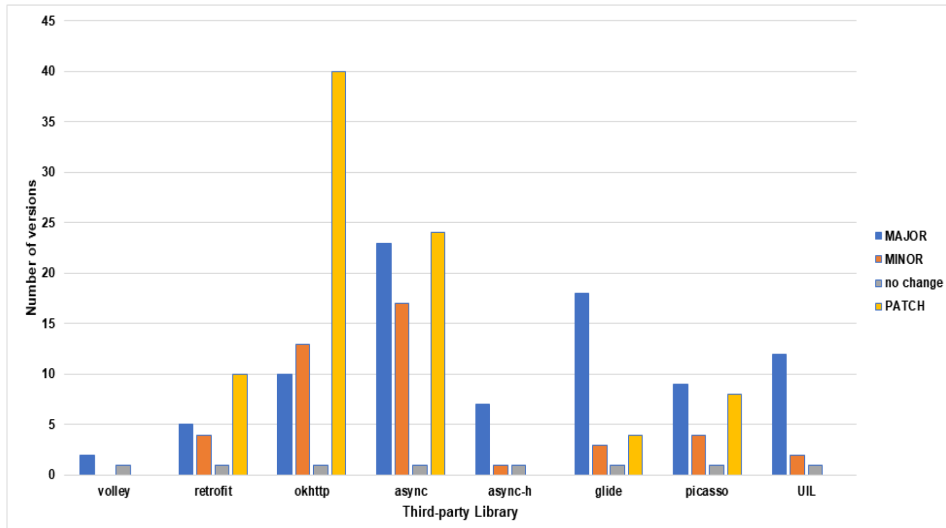


Figure 8.4: Number of versions per change type per library

At the moment we have energy data for only one version per third-party library. Therefore, we grouped the versions (see Table 8.6) based on the changes made between two consecutive versions: major, minor, or a patch. Major version change include incompatible API changes, minor version change include the addition in functionality with backward-compatible and a patch version change includes bug fixes with backward-compatible. We assume that in the case of minor and patch level changes, the difference in consecutive versions are small and might not cause a significant change in energy consumption. Therefore, to cover more library versions via REHAB, we group such similar library versions based on the type of change. The consecutive versions with minor changes and patches are kept in

a single group until a major change occurs. Upon encountering a major change between consecutive versions, we place the version with major change and its subsequent versions with minor and patch changes in the next group, and so on. From here onward, the groups containing the library versions for which we have energy data are referred as target groups.

Table 8.6: Number of groups created per library in which versions were divided.

| ID | Library | Number of groups |
|---|---|---|
| vo | Volley | 1 |
| re | Retrofit | 3 |
| ok | OkHttp | 6 |
| async | AndroidAsync | 14 |
| async-h | AsyncHttp | 5 |
| gli | Glide | 15 |
| pic | Picasso | 5 |
| uil | UIL | 9 |

Out of 3457 apps belonging to F-Droid repository, selected third-party HTTP libraries were detected in 789 apps i.e. 21%. Out of 5000 apps belonging to Google playstore, selected third-party HTTP libraries were detected in 2942 app i.e. 78% (see Table 8.7). Out of 789 apps belonging to F-Droid repository, versions from the target group were detected in 395 app i.e. 50%. Out of 2942 apps belonging to Google playstore, versions from the target group were detected in 1636 app i.e. 55%. (see Annex D for detailed data on versions, groups, number of apps etc.).

Table 8.7: Number of apps in which selected third-party HTTP libraries were detected (irrespective of the version).

| ID | Library | Number of Google playstore apps | Number of F-Droid apps | TOTAL |
|---|---|---|---|---|
| vo | Volley | 101 | 16 | 117 |
| re | Retrofit | 637 | 172 | 809 |
| ok | OkHttp | 955 | 366 | 1321 |
| async | AndroidAsync | 11 | 13 | 24 |
| async-h | AsyncHttp | 48 | 8 | 56 |
| gli | Glide | 461 | 109 | 570 |
| pic | Picasso | 619 | 80 | 699 |
| uil | UIL | 110 | 25 | 135 |
| | **TOTAL** | **2942** | **789** | **3731** |

Tables 8.8 to 8.15 shows group of versions created per library along with number of apps in which these groups were detected. The version highlight in the red are the ones for which we measured energy data in chapter 5.

Third-party libraries have multiple versions and it is a time consuming task to measure energy consumption of each version in every possible use-case. If library

versions are released frequently then energy data also needs to be updated frequently. As we grouped the similar versions together, therefore, we can say with some confidence that the recommendation that REHAB made for the selected version could be applied to the other versions that belong to the same group. One can argue, that in minor and patch changes, even though the difference in code or functionality is small but it could still cause significant changes in energy consumption. However, these changes are done in backward compatible manner which means that apps consuming the APIs from such versions could switch between versions without changing their code. Minor and patch level changes in library versions are usually quick fixes by the creators of third-party libraries until the version with major change is released. In the versions with major changes APIs are deleted, added or changed in a way that require developers to change their app code. We suggest that the software practitioners could adopt a similar scheme and group the versions together. Energy consumption could be measured first for the version with major changes, with the assumption that the subsequent versions with minor or patch level changes are not too different. Then if needed the energy consumption for the version with minor and patch changes could be measured respectively. Measuring energy consumption of third-party libraries is lengthy process, our tool ARENA (presented in Chapter 7) could help the researchers to speed up the energy measurement process without human errors.

Table 8.8: Group of versions created for Volley library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 1.0.0<br>1.1.0<br>1.1.1 | 110 | 100% | 25 | 100% | 1 | no change<br>MINOR<br>MINOR |
| TOTAL | 110 | | 25 | | | |

Table 8.9: Group of versions created for Retrofit library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 2.0.0<br>2.0.1<br>2.0.2<br>2.1.0 | 139 | 22% | 22 | 13% | 1 | no change<br>PATCH<br>MINOR<br>PATCH |
| 2.2.0<br>2.3.0<br>2.4.0<br>2.5.0 | 444 | 70% | 26 | 15% | 2 | MAJOR<br>PATCH<br>MINOR<br>MINOR |
| 2.6.0<br>2.6.1<br>2.6.2<br>2.6.3<br>2.6.4<br>2.7.0<br>2.7.1<br>2.7.2 | 54 | 8% | 124 | 72% | 3 | MAJOR<br>MINOR<br>PATCH<br>MINOR<br>PATCH<br>PATCH<br>PATCH<br>PATCH |

| 2.8.0 | | | | | | MINOR |
|---|---|---|---|---|---|---|
| 2.8.1 | | | | | | PATCH |
| 2.8.2 | | | | | | MINOR |
| 2.9.0 | | | | | | PATCH |
| **TOTAL** | **637** | | **172** | | | |

Table 8.10: Group of versions created for OkHttp library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 3.0.0 | 42 | 4% | 8 | 2% | 1 | no change |
| 3.0.1 | | | | | | PATCH |
| 3.1.0 | | | | | | MINOR |
| 3.1.1 | | | | | | MINOR |
| 3.1.2 | | | | | | PATCH |
| 3.2.0 | | | | | | MINOR |
| 3.3.0 | | | | | | MINOR |
| 3.3.1 | | | | | | PATCH |
| 3.4.0 | 179 | 19% | 50 | 14% | 2 | MAJOR |
| 3.4.1 | | | | | | PATCH |
| 3.4.2 | | | | | | PATCH |
| 3.5.0 | | | | | | MINOR |
| 3.6.0 | | | | | | MINOR |
| 3.7.0 | | | | | | MINOR |
| 3.8.0 | | | | | | MINOR |
| 3.8.1 | | | | | | PATCH |
| 3.9.0 | 734 | 77% | 277 | 76% | 3 | MAJOR |
| 3.9.1 | | | | | | PATCH |
| 3.10.0 | | | | | | PATCH |
| 3.11.0 | | | | | | MINOR |
| 3.12.0 | | | | | | MINOR |
| 3.12.1 | | | | | | PATCH |
| 3.12.2 | | | | | | PATCH |
| 3.12.3 | | | | | | PATCH |
| 3.12.4 | | | | | | PATCH |
| 3.12.5 | | | | | | PATCH |
| 3.12.6 | | | | | | PATCH |
| 3.12.7 | | | | | | PATCH |
| 3.12.8 | | | | | | PATCH |
| 3.12.9 | | | | | | PATCH |
| 3.12.10 | | | | | | PATCH |
| 3.12.11 | | | | | | PATCH |
| 3.12.12 | | | | | | PATCH |
| 3.13.0 | | | | | | MINOR |
| 3.13.1 | | | | | | PATCH |
| 3.14.0 | | | | | | MINOR |
| 3.14.1 | | | | | | PATCH |
| 3.14.2 | | | | | | PATCH |
| 3.14.3 | | | | | | PATCH |
| 3.14.4 | | | | | | PATCH |
| 3.14.5 | | | | | | PATCH |
| 3.14.6 | | | | | | PATCH |
| 3.14.7 | | | | | | PATCH |
| 3.14.8 | | | | | | PATCH |
| 3.14.9 | | | | | | PATCH |
| 4.0.0 | | | | | | MINOR |
| 4.0.1 | | | | | | PATCH |
| 4.1.0 | | | | | | MINOR |

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 4.1.1 | | | | | | PATCH |
| 4.2.0 | | | | | | MINOR |
| 4.2.1 | | | | | | PATCH |
| 4.2.2 | | | | | | PATCH |
| 4.3.0 | 0 | 0% | 6 | 2% | 4 | MAJOR |
| 4.3.1 | | | | | | PATCH |
| 4.4.0 | 0 | 0% | 4 | 1% | 5 | MAJOR |
| 4.4.1 | | | | | | PATCH |
| 4.5.0 | | | | | | MINOR |
| 4.6.0 | 0 | 0% | 21 | 6% | 6 | MAJOR |
| 4.7.0 | | | | | | MINOR |
| 4.7.1 | | | | | | PATCH |
| 4.7.2 | | | | | | PATCH |
| 4.8.0 | | | | | | PATCH |
| 4.8.1 | | | | | | PATCH |
| 4.9.0 | | | | | | MINOR |
| **TOTAL** | **955** | | **366** | | | |

Table 8.11: Group of versions created for AndroidAsync library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 1.0.0 | 0 | 0% | 0 | 0% | 1 | no change |
| 1.0.2 | | | | | | PATCH |
| 1.0.3 | | | | | | PATCH |
| 1.0.5 | | | | | | PATCH |
| 1.0.6 | | | | | | MINOR |
| 1.0.7 | | | | | | PATCH |
| 1.0.8 | | | | | | MINOR |
| 1.0.9 | 0 | 0% | 0 | 0% | 2 | MAJOR |
| 1.1.0 | | | | | | PATCH |
| 1.1.1 | 0 | 0% | 0 | 0% | 3 | MAJOR |
| 1.1.2 | | | | | | PATCH |
| 1.1.3 | 0 | 0% | 0 | 0% | 4 | MAJOR |
| 1.1.4 | 0 | 0% | 0 | 0% | 5 | MAJOR |
| 1.1.5 | | | | | | MINOR |
| 1.1.6 | | | | | | MINOR |
| 1.1.8 | 0 | 0% | 0 | 0% | 6 | MAJOR |
| 1.1.9 | | | | | | MINOR |
| 1.2.0 | | | | | | MINOR |
| 1.2.1 | 0 | 0% | 0 | 0% | 7 | MAJOR |
| 1.2.2 | | | | | | MINOR |
| 1.2.3 | | | | | | MINOR |
| 1.2.4 | | | | | | MINOR |
| 1.2.5 | 0 | 0% | 0 | 0% | 8 | MAJOR |
| 1.2.6 | 0 | 0% | 0 | 0% | 9 | MAJOR |
| 1.2.8 | 0 | 0% | 1 | 9% | 10 | MAJOR |
| 1.2.9 | | | | | | PATCH |
| 1.3.0 | | | | | | PATCH |
| 1.3.1 | | | | | | PATCH |
| 1.3.2 | | | | | | MINOR |
| 1.3.3 | | | | | | PATCH |
| 1.3.4 | | | | | | PATCH |
| 1.3.5 | | | | | | PATCH |
| 1.3.6 | | | | | | PATCH |
| 1.3.7 | | | | | | PATCH |
| 1.3.8 | | | | | | MINOR |
| 1.3.9 | | | | | | MINOR |

| 1.4.0 | | | | | | PATCH |
|---|---|---|---|---|---|---|
| 1.4.1 | | | | | | MINOR |
| 2.0.0 | | | | | | MAJOR |
| 2.0.1 | | | | | | MINOR |
| 2.0.2 | | | | | | MINOR |
| 2.0.3 | | | | | | MINOR |
| 2.0.4 | | | | | | PATCH |
| 2.0.5 | | | | | | PATCH |
| 2.0.6 | | | | | | MINOR |
| 2.0.7 | | | | | | MINOR |
| 2.0.8 | 4 | 36% | 7 | 64% | 11 | PATCH |
| 2.0.9 | | | | | | MINOR |
| 2.1.0 | | | | | | MINOR |
| 2.1.1 | | | | | | PATCH |
| 2.1.2 | | | | | | MINOR |
| 2.1.3 | | | | | | PATCH |
| 2.1.4 | | | | | | MINOR |
| 2.1.5 | | | | | | MINOR |
| 2.1.6 | | | | | | PATCH |
| 2.1.7 | | | | | | MAJOR |
| 2.1.8 | | | | | | MINOR |
| 2.1.9 | 7 | 64% | 5 | 45% | 12 | MINOR |
| 2.2.0 | | | | | | MINOR |
| 2.2.1 | | | | | | PATCH |
| 3.0.0 | 0 | 0% | 0 | 0% | 13 | MAJOR |
| 3.0.8 | | | | | | PATCH |
| 3.0.9 | 0 | 0% | 0 | 0% | 14 | MAJOR |
| 3.1.0 | | | | | | PATCH |
| **TOTAL** | **11** | | **13** | | | |

Table 8.12: Group of versions created for AsyncHttp library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 1.4.3 | 4 | 8% | 1 | 13% | 1 | no change |
| 1.4.4 | 12 | 25% | 0 | 0% | 2 | MAJOR |
| 1.4.5 | 6 | 13% | 1 | 13% | 3 | MAJOR |
| 1.4.6 | | | | | | MINOR |
| 1.4.7 | 6 | 13% | 0 | 0% | 4 | MAJOR |
| 1.4.8 | | | | | | MINOR |
| 1.4.9 | | | | | | MAJOR |
| 1.4.10 | 20 | 42% | 6 | 75% | 5 | MINOR |
| 1.4.11 | | | | | | MINOR |
| **TOTAL** | **48** | | **8** | | | |

Table 8.13: Group of versions created for Glide library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 3.3.0 | 0 | 0% | 0 | 0% | 1 | no change |
| 3.3.1 | 1 | 0% | 0 | 0% | 2 | MAJOR |
| 3.4.0 | 0 | 0% | 0 | 0% | 3 | MAJOR |
| 3.5.0 | | | | | | MAJOR |
| 3.5.1 | 4 | 1% | 1 | 1% | 4 | MINOR |
| 3.5.2 | | | | | | MINOR |

| | | | | | | |
|---|---|---|---|---|---|---|
| 3.6.0<br>3.6.1<br>3.7.0<br>3.8.0 | 267 | 58% | 51 | 47% | 5 | MAJOR<br>MINOR<br>PATCH<br>MINOR |
| 4.0.0 | 0 | 0% | 0 | 0% | 6 | MAJOR |
| 4.1.0<br>4.1.1 | 2 | 0% | 0 | 0% | 7 | MAJOR<br>PATCH |
| 4.2.0 | 3 | 1% | 0 | 0% | 8 | MAJOR |
| 4.3.0<br>4.3.1 | 8 | 2% | 0 | 0% | 9 | MAJOR<br>MINOR |
| 4.4.0 | 0 | 0% | 0 | 0% | 10 | MAJOR |
| 4.5.0<br>4.6.0<br>4.6.1 | 18 | 4% | 6 | 6% | 11 | MAJOR<br>MINOR<br>PATCH |
| 4.7.0<br>4.7.1 | 42 | 9% | 2 | 2% | 12 | MAJOR<br>PATCH |
| 4.8.0 | 16 | 3% | 8 | 7% | 13 | MAJOR |
| 4.9.0 | 77 | 17% | 11 | 10% | 14 | MAJOR |
| 4.10.0<br>4.11.0 | 23 | 5% | 30 | 28% | 15 | MAJOR<br>MINOR |
| **TOTAL** | 461 | | 109 | | | |

Table 8.14: Group of versions created for Picasso library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 1.0.0<br>1.0.1<br>1.0.2 | 0 | 0% | 0 | 0% | 1 | no change<br>MINOR<br>MINOR |
| 1.1.0<br>1.1.1 | 4 | 1% | 0 | 0% | 2 | MAJOR<br>PATCH |
| 2.0.0<br>2.0.1<br>2.0.2<br>2.1.0<br>2.1.1<br>2.2.0<br>2.3.0<br>2.3.1<br>2.3.2<br>2.3.3<br>2.3.4<br>2.4.0 | 73 | 12% | 8 | 10% | 3 | MAJOR<br>PATCH<br>PATCH<br>PATCH<br>MINOR<br>MINOR<br>MINOR<br>PATCH<br>PATCH<br>PATCH<br>MINOR<br>MINOR |
| 2.5.0<br>2.5.1<br>2.5.2 | 330 | 53% | 34 | 43% | 4 | MAJOR<br>MINOR<br>MINOR |
| 2.8.0<br>2.71828.0 | 212 | 34% | 38 | 48% | 5 | MAJOR<br>PATCH |
| **TOTAL** | **619** | | **80** | | | |

Table 8.15: Group of versions created for UIL (Universal Image Loader) library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 1.7.0<br>1.7.1 | 30 | 30% | 0 | 0% | 1 | no change<br>MINOR |
| 1.8.0<br>1.8.1 | 1 | 1% | 0 | 0% | 2 | MAJOR<br>MINOR |
| 1.8.2<br>1.8.3<br>1.8.4 | 2 | 2% | 2 | 13% | 3 | MAJOR<br>MINOR<br>MINOR |
| 1.8.5<br>1.8.6 | 12 | 12% | 1 | 6% | 4 | MAJOR<br>MINOR |
| 1.9.0 | 2 | 2% | 0 | 0% | 5 | MAJOR |
| 1.9.1 | 11 | 11% | 0 | 0% | 6 | MAJOR |
| 1.9.2 | 1 | 1% | 2 | 13% | 7 | MAJOR |
| 1.9.3 | 10 | 10% | 4 | 25% | 8 | MAJOR |
| 1.9.4<br>1.9.5 | 32 | 32% | 7 | 44% | 9 | MAJOR<br>MINOR |
| **TOTAL** | **101** | | **16** | | | |

## 8.4. REHAB Evaluation

We report the accuracy of REHAB when used on Android apps within Android Studio IDE. The purpose of this evaluation was to check 1) if REHAB can correctly identify the use-cases and the related HTTP requests in the source code, 2) if REHAB can correctly map the detected use cases to the pre-defined rules in the knowledge base and show recommendations.

The correctness of recommendations made by REHAB is exclusively dependent on the energy measurements used in the start to create the knowledge base. Once we assume that the energy measurements are correct, the recommendation made by REHAB is also considered correct. Currently, the knowledge base (cf. Section 8.1.1) contains a finite set of pre-defined rules. REHAB lookup the use cases and HTTP requests in the source code and match them to rules. A related recommendation based on the energy measurements is shown to the user if a match is found.

First, we evaluated REHAB on 45 custom app versions (we call it app-set 1[7]). For each of these app versions, we already knew the libraries included and request types used. Therefore, it was easier to spot false positives or false negatives. Then from the F-droid repository, we selected four open-source Android apps (we call it app-set 2 [8]). The criteria for selecting apps was 1) one or more selected third-party HTTP libraries were present in the app, 2) the size of the project should be

[7] The custom app versions used in the evaluation are available at https://figshare.com/s/8e65154ab9821eba96af

[8] The open source apps used in evaluation are available at https://figshare.com/s/5facc180f9cbd36ef114

small (less than 2K), as we wanted to confirm the results afterwards manually, and 3) the source code of the app is available.

For app-set 1, REHAB correctly identified the selected third-party HTTP libraries and HTTP requests in all 45 custom app versions. For each detected use case, two types of recommendations were shown to the user 1) an alternative energy-efficient third-party library for each use case, 2) ranking of alternative third-party libraries for the individual and a combination of use cases. The recommendations shown by REHAB for each app version were manually checked for correctness. Table 8.16 shows the detected use cases, the number of detected HTTP requests, detected libraries, false positives (FP), false negatives (FN), and the number of recommendations shown.

For app-set 2, REHAB correctly identified selected third-party HTTP libraries and related HTTP requests. Within each app, multiple use cases and HTTP requests were detected. For each detected use case, two types of recommendations were shown to the user 1) an alternative energy-efficient third-party library for each use case, 2) ranking of alternative third-party libraries for the individual and a combination of use cases. Table 8.17 shows the detected use cases, the number of detected HTTP requests, detected libraries, false positives (FP), false negatives (FN), and the number of recommendations shown. For the first app, 'Android-Volley-Master', four HTTP requests were detected. Out of four detected HTTP requests, two were GET requests (UC-GF), and two were GET image requests (UC-GI). For the second app, 'android-http-app-master', one POST request was detected. For the third app, 'Memesharing', two request types were detected, i.e. one GET request (UC-GF) to retrieve a file from a server and one GET request (UC-GI) to download an image from the server. For the fourth app, 'Weatherapp', four HTTP requests were detected. Out of four detected HTTP requests, one was GET requests (UC-GF), and three were GET image requests (UC-GI).

For app-set 1 and 2, REHAB correctly detected and highlighted the relevant code in source code files. Figure 8.3 in Section 8.2.1 shows an example of the recommendation shown by REHAB in the 'Memesharing' app in app-set 2. If energy measurement is correct, then the recommendation is correct. The 'Memesharing' app used third-party HTTP libraries Volley and Glide for making GET requests to extract images from the server. REHAB recommended the library UIL, which has a better energy value. Two types of recommendations are shown to the user 1) an alternative energy-efficient third-party library for each use case, 2) ranking of alternative third-party libraries for the individual and a combination of use cases.

Table 8.16: REHAB evaluation results (app-set 1)

| App version | Use Case | Detected Library | # of HTTP requests | FP | FN | # of recomm. |
|---|---|---|---|---|---|---|
| 1 | GF | Volley | 1 | 0 | 0 | 2 |
| 2 | GF | Retrofit | 1 | 0 | 0 | 2 |

| 38 | GF | OKHttp | 1 | 0 | 0 | 2 |
|---|---|---|---|---|---|---|
| 4 | GF | AsyncHttp | 1 | 0 | 0 | 2 |
| 5 | GF | AndroidAsync | 1 | 0 | 0 | 2 |
| 6 | PF | Volley | 1 | 0 | 0 | 2 |
| 7 | PF | Retrofit | 1 | 0 | 0 | 2 |
| 8 | PF | OKHttp | 1 | 0 | 0 | 2 |
| 9 | PF | AsyncHttp | 1 | 0 | 0 | 2 |
| 10 | PF | AndroidAsync | 1 | 0 | 0 | 2 |
| 11 | PJO | Volley, Gson | 1 | 0 | 0 | 2 |
| 12 | PJO | Volley, Jackson | 1 | 0 | 0 | 2 |
| 13 | PJO | Volley, Moshi | 1 | 0 | 0 | 2 |
| 14 | PJO | Retrofit, Gson | 1 | 0 | 0 | 2 |
| 15 | PJO | Retrofit, Jackson | 1 | 0 | 0 | 2 |
| 16 | PJO | Retrofit, Moshi | 1 | 0 | 0 | 2 |
| 17 | PJO | OKHttp, Gson | 1 | 0 | 0 | 2 |
| 18 | PJO | OKHttp, Jackson | 1 | 0 | 0 | 2 |
| 19 | PJO | OKHttp , Moshi | 1 | 0 | 0 | 2 |
| 20 | PJO | AsyncHttp, Gson | 1 | 0 | 0 | 2 |
| 21 | PJO | AsyncHttp, Jackson | 1 | 0 | 0 | 2 |
| 22 | PJO | AsyncHttp, Moshi | 1 | 0 | 0 | 2 |
| 23 | PJO | AndroidAsync, Gson | 1 | 0 | 0 | 2 |
| 24 | PJO | AndroidAsync, Jackson | 1 | 0 | 0 | 2 |
| 25 | PJO | AndroidAsync, Moshi | 1 | 0 | 0 | 2 |
| 26 | GI | Volley | 1 | 0 | 0 | 2 |
| 27 | GI | AsyncHttp | 1 | 0 | 0 | 2 |
| 28 | GI | Glide | 1 | 0 | 0 | 2 |
| 29 | GI | Picasso | 1 | 0 | 0 | 2 |
| 30 | GI | UIL | 1 | 0 | 0 | 2 |
| 31 | GJO | Volley, Gson | 1 | 0 | 0 | 2 |
| 32 | GJO | Volley, Jackson | 1 | 0 | 0 | 2 |
| 33 | GJO | Volley, Moshi | 1 | 0 | 0 | 2 |
| 34 | GJO | Retrofit, Gson | 1 | 0 | 0 | 2 |
| 35 | GJO | Retrofit, Jackson | 1 | 0 | 0 | 2 |
| 36 | GJO | Retrofit, Moshi | 1 | 0 | 0 | 2 |
| 37 | GJO | OKHttp, Gson | 1 | 0 | 0 | 2 |
| 38 | GJO | OKHttp , Jackson | 1 | 0 | 0 | 2 |
| 39 | GJO | OKHttp , Moshi | 1 | 0 | 0 | 2 |
| 40 | GJO | AsyncHttp, Gson | 1 | 0 | 0 | 2 |
| 41 | GJO | AsyncHttp, Jackson | 1 | 0 | 0 | 2 |
| 42 | GJO | AsyncHttp, Moshi | 1 | 0 | 0 | 2 |
| 43 | GJO | AndroidAsync, Gson | 1 | 0 | 0 | 2 |
| 44 | GJO | AndroidAsync, Jackson | 1 | 0 | 0 | 2 |
| 45 | GJO | AndroidAsync, Moshi | 1 | 0 | 0 | 2 |

Table 8.17: REHAB evaluation results (app-set 2).

| App | Use Case | Detected Library | # of HTTP requests | FP | FN | # of recomm. |
|---|---|---|---|---|---|---|
| Android-volley-Master | GF, GI | Volley, Picasso | 4 | 0 | 0 | 5 |
| android-http-app-master | PF | AsyncHttp | 1 | 0 | 0 | 2 |
| Memesharing | GF, GI | volley, Glide | 2 | 0 | 0 | 5 |
| Weatherapp | GF, GI | Retrofit, Picasso | 4 | 0 | 0 | 5 |

# 9. CONCLUSION AND FUTURE WORK

Sustainability in software engineering is a relatively new and fast growing field of research. Green software engineering aims to produce sustainable software products with minimum negative impact on the environment. In order to make greener software products, software practitioners need actionable timely information, to make useful trade-offs between energy efficiency and other quality attributes, like performance, during development. Software analytics could be used to provide this support, as it combines information from different software artifacts and converts it into useful information. In this thesis, we have evaluated alternative coding patterns to save energy in mobile apps in two contexts: 1) custom app code written by the developers; 2) library modules, i.e., reusable services added to the project. In addition, we explore the state of the art support tools for energy efficient app development and investigate their strengths and weaknesses. Based on this analysis, we provide support tools that overcomes some of the limitations of the existing tools.

## 9.1. Contributions and Findings

In this section we summarize our contributions and findings in the area of green software engineering.

### 9.1.1. Code Smell Refactorings

In this thesis, we extend previous studies by investigating the energy impact of refactoring five code smell types (first individually per type and then in permutations) of native Android open source apps. We also study the impact of using the code smell refactorings on the execution time of native Android open source apps. Our results indicate that the maximum energy reductions recorded are 10.8% and 10.5% for refactoring code smell ‘Duplicated code’ and ‘Type Checking’ respectively. Specific permutations of code smell refactorings should be used with caution, as their energy consumption impact might differ strongly between the selected Android apps. We observe neither significant increase nor decrease of the execution time in selected Android apps.

### 9.1.2. Third-party Libraries

We investigate whether popular Android third-party HTTP libraries vary in energy consumption. In addition, we checked whether there is a correlation between performance and energy consumption. To achieve this goal, we performed a controlled experiment. We created 45 different versions of a custom app and explored the energy consumption and performance of eight popular Android third-party HTTP libraries in five typical use cases. We found that there is a significant

variance in energy consumption between the selected Android third-party HTTP libraries. Based on our discussion, we hypothesize that the energy drivers are related to the internal structure of the Android third-party HTTP libraries, in particular with the handling of asynchronous tasks and the creation of multiple threads in the background. This is something that should be investigated in follow-up studies. We did not find any significant correlation between performance and energy consumption. Our results will help app developers make better choices when selecting Android third-party HTTP libraries.

### 9.1.3. Tool Support for Developing Green Android Apps

We provide an overview of the state-of-the-art and highlight research opportunities with respect to support tools available for green Android development. Based on our analysis we identified tools for detecting/refactoring code smells/energy bugs, which were classified into three categories 1) 'Profiler', 2) 'Detector', 3) 'Optimizer'. Additionally, we identified tools for detecting/migrating third-party libraries in Android apps, which were classified into 1) Identifier, 2) Migrator, 3) Controller categories. The main findings of this study are that most 'Profiler' tools provide a graphical representation of energy consumption over time. Most "Detector" tools provide a list of energy bugs/code smells to be manually corrected by a developer for the improvement of energy. Most "Optimizer" automatically convert original APK/SC to a refactored version(s) of APK/SC. Tools in 'Identifier', 'Migrator' or 'Controller' categories do not provide supports to developers to optimize code w.r.t energy consumption. The most typical technique in "Detector" and 'Optimizer 'category was static source code analysis using a predefined set of code smells and rules. The most typical techniques in the 'Identifier' category were module decoupling and feature similarity. While in 'Migrator' and 'Controller categories, API hooking and collaborative filtering in combination with natural-language processing were used, respectively.

We present the support tool ARENA to help automate the energy measurement process and to reduce the risks related to human errors during energy measurement. Energy consumption of app could be measured via software or hardware-based approaches. Compared to software-based approaches, hardware-based approaches for collecting energy data are more accurate but difficult to apply. ARENA connects with one of the most widely used physical measurement devices (Monsoon Power Monitor) to capture energy data. ARENA provides an interface that is consistent with the IntelliJ/Android Studio IDEs and enables developers and researchers to compare energy consumption between versions of the apps. Further, ARENA helps in aggregating, statistically analyzing, reporting and visualizing the energy data.

We also present a support tool REHAB for recommending energy-efficient third-party libraries to the developers. For a particular task usually alternative third-

party libraries are available that offer similar functionalities. If the third-party library included in the app is not energy efficient it might drain the mobiles' battery. REHAB helps Android developers by recommending energy efficient third-party HTTP libraries during development. Additionally, we discuss how the scope of REHAB can be widened by conducting usage and change analysis on 8457 Android apps. The usage analysis quantifies the usage of selected third-party HTTP libraries and all their versions in a set of real Android apps. The change analysis quantifies how what kind of changes are made in the consecutive versions. Third-party libraries have multiple versions and it is a time consuming task to measure energy consumption of each version in every possible use-case. We suggest that the software practitioners group the versions together. Energy consumption could be measured first for the version with major changes, with the assumption that the subsequent versions with minor or patch level changes are not too different. Then if needed the energy consumption for the version with minor and patch changes could be measured respectively.

## 9.2. Opportunities for Future Work

This work opens multiple opportunities for future work, which we outline below.

### 9.2.1. Contextual Data for Refactoring

From previous research (cf. Chapter 3) and the research conducted in this thesis (contribution 1) it is clear that code smell refactorings can impact energy consumption of apps. However, more context based data is needed in order to provide actionable recommendations to developers. In future, one could perform similar studies with wider scope to measure the impact of applying combinations and permutations of code smell refactorings on different underlying platforms (such as Java virtual machine vs Java runtime environment) and types (native vs hybrid) of mobile apps. Similarly, one could also explore how many times a single refactoring could be applied before it starts to negatively impact energy consumption. As manual detection and refactoring of code smells in apps is a human intensive task, and usually hinders such experiment, support tools such as xAL (extended Android Lint) [53] could be used. Similarly support tool such as ARENA could be used to automate the energy measurement process.

### 9.2.2. Trade-off Analysis

A third-party library chosen for a scenario for one quality aspect could have a positive or negative correlation with other quality aspects such as usability and resource consumption. Trade-off analysis is needed to evaluate how alternative third-party libraries for a scenario affect certain quality requirements and constraints. The benefits of such a trade-off analysis are; 1) helpful in better understanding the trade-off between the implementation choices regarding certain

quality aspects, 2) helpful in documenting all the trade-off decisions for future use. For example, third-party library 'A' is energy efficient but difficult to implement and requires developers to write additional code or third-party library 'B' is more secure but not energy efficient. Trade-off analysis could help in choosing which third-party library is better for a given task in a certain context.

### 9.2.3. Energy Data for Creating Software Based Energy Prediction Models

Developers want software based models that could predict energy consumption of app during development. To produce such models accurate energy data is required. There is lack of contextual energy data because it is costly in terms of efforts to produce such data. Not only we need hardware devices for accurate energy measurements but also specialized tests need to be written for every app. On top of that energy measurements could be taken at different level such as system, method, source line of codes. Better tools are needed to automate energy measurement process. Support tools such as ARENA could be improved to connect to more hardware devices with option to select the granularity (system level, method level etc.) of energy measurements.

# BIBLIOGRAPHY

[1] Android asynchronous http client. [Online; accessed 2019-11-04].

[2] Android sdk: Working with picasso. [Online; accessed 2019-04-25].

[3] Api reference | youtube data api | google developers. [Online; accessed 2019-10-21].

[4] Developer workflow basics | android developers. [Online; accessed 2021-10-15].

[5] Distribution dashboard | android developers. [Online; accessed 2019-11-04].

[6] Glide android library.

[7] Image loader andriod library.

[8] Picasso. [Online; accessed 2019-04-25].

[9] Smartphone ap market share 2014-2018 | statista. [Online; accessed 2019-11-04].

[10] What percentage of internet traffic is mobile in 2019? [Online; accessed 2019-11-04].

[11] Energy-efficient ICT in practice : Planning and implementation of GreenIT measures in data centres and the office. Technical report, 2014.

[12] Mobile device power monitor manual, 2017. [Online; accessed 2018-06-04].

[13] Hayri Acar. *Software development methodology in a Green IT environment*. PhD thesis, Université de Lyon, 2017.

[14] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 468–471. IEEE, 2016.

[15] Hussein Alrubaye, Mohamed Wiem Mkaouer, Igor Khokhlov, Leon Reznik, Ali Ouni, and Jason Mcgoff. Learning to recommend third-party library migration opportunities at the api level. *Applied Soft Computing*, 90:106140, 2020.

[16] Luca Ardito, Giuseppe Procaccianti, Marco Torchiano, and Giuseppe Migliore. Profiling power consumption on mobile devices. *ENERGY*, pages 101–106, 2013.

[17] Larissa Azevedo, Altino Dantas, and Celso G Camilo-Junior. Droidbugs: An android benchmark for automated program repair. *arXiv preprint arXiv:1809.07353*, 2018.

[18] Michael Backes, Sven Bugiel, and Erik Derr. Reliable third-party library detection in android and its security applications. pages 356–367. ACM Press, 10 2016.

[19] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 588–598, 2014.

[20] Abhijeet Banerjee and Abhik Roychoudhury. Automated re-factoring of android apps to enhance energy-efficiency. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, pages 139–150, 2016.

[21] Reuben Binns, Jun Zhao, Max Van Kleek, and Nigel Shadbolt. Measuring third-party tracker power across web and mobile. *ACM Transactions on Internet Technology (TOIT)*, 18(4):1–22, 2018.

[22] Coral Calero, Mª Ángeles Moraga, and Mario Piattini. *Introduction to Software Sustainability*, pages 1–15. Springer International Publishing, Cham, 2021.

[23] Coral Calero and Mario Piattini. *Green in software engineering*, volume 3. Springer, 2015.

[24] Coral Calero and Mario Piattini. Introduction to green in software engineering. In *Green in Software Engineering*, pages 3–27. Springer, 2015.

[25] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. Investigating the energy impact of android smells. pages 115–126. IEEE, 2 2017.

[26] Nitin Singh Chauhan and Ashutosh Saxena. A green software development life cycle for cloud computing. *It Professional*, 15(1):28–34, 2013.

[27] Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186, 2014.

[28] Shaiful Chowdhury, Stephanie Borle, Stephen Romansky, and Abram Hindle. GreenScaler: training software energy models with automatic test generation. *Empirical Software Engineering*, 24(4):1649–1692, August 2019.

[29] Shaiful Chowdhury, Silvia Di Nardo, Abram Hindle, and Zhen Ming Jiang. An exploratory study on assessing the energy impact of logging on android applications. *Empirical Softw. Engg.*, 23(3):1422–1456, June 2018.

[30] Shaiful Alam Chowdhury and Abram Hindle. Greenoracle: Estimating software energy consumption with energy measurement corpora. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 49–60. IEEE, 2016.

[31] Shaiful Alam Chowdhury, Abram Hindle, Rick Kazman, Takumi Shuto, Ken Matsui, and Yasutaka Kamei. Greenbundle: An empirical study on the energy impact of bundled processing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1107–1118, 2019.

[32] Yi-Fan Chung, Chun-Yu Lin, and Chung-Ta King. Aneprof: Energy profiling for android java virtual machine and applications. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 372–379. IEEE, 2011.

[33] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, 114:494–509, 1993.

[34] Vanessa N. Cooper, Hossain Shahriar, and Hisham M. Haddad. A survey of android malware characterisitics and mitigation techniques. pages 327–332. IEEE, 4 2014.

[35] Claudio Corrodi, Timo Spring, Mohammad Ghafari, and Oscar Nierstrasz. Idea: Benchmarking android data leak detection tools. volume 10953 LNCS, pages 116–123. Springer Verlag, 6 2018.

[36] Luis Cruz and Rui Abreu. Catalog of energy patterns for mobile applications. *Empirical Softw. Engg.*, 24(4):2209–2235, August 2019.

[37] Luis Cruz and Rui Abreu. Improving energy efficiency through automatic refactoring. *J. Softw. Eng. Res. Dev.*, 7:2, 2019.

[38] Luis Cruz and Rui Abreu. On the energy footprint of mobile testing frameworks. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[39] Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia. Do energy-oriented changes hinder maintainability? In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 29–40, 2019.

[40] Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: Memory power estimation and capping. In *2010*

*ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, pages 189–194. IEEE, 2010.

[41] Fernando López De La Mora and Sarah Nadi. Which library should i use? a metric-based comparison of software libraries. pages 37–40. IEEE Computer Society, 5 2018.

[42] Abebaw Degu. Android application memory and energy performance: Systematic literature review. *IOSR J. of Comp. Eng.*, 21(3):20–32, 2019.

[43] Marin Delchev and Muhammad Firdaus Harun. Investigation of code smells in different software domains. 2015.

[44] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200, 2017.

[45] Google Developers. Volley overview | android developers. [Online; accessed 2019-04-25].

[46] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Petra: A software-based tool for estimating the energy profile of android applications. pages 3–6. IEEE, 5 2017.

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

[48] Markus Dick, Stefan Naumann, and Norbert Kuhn. A model and selected instances of green and sustainable software. In *What kind of information society? Governance, virtuality, surveillance, sustainability, resilience*, pages 248–259. Springer, 2010.

[49] Koushik Dutta. Androidasync. [Online; accessed 2019-11-04].

[50] Egham. Gartner Says Worldwide End-User Device Spending Set to Increase 7 Percent in 2018; Global Device Shipments Are Forecast to Return to Growth, 2018.

[51] Cheng Fang, Jun Liu, and Zhenming Lei. Fine-grained http web traffic analysis based on large-scale mobile datasets. *IEEE Access*, 4:4364–4373, 2016.

[52] Keith I Farkas, Jason Flinn, Godmar Back, Dirk Grunwald, and Jennifer M Anderson. Quantifying the energy consumption of a pocket computer and a java virtual machine. In *Proceedings of the 2000 ACM SIGMETRICS*

*international conference on Measurement and modeling of computer systems*, pages 252–263, 2000.

[53] Iffat Fatima, **Anwar, Hina**, Dietmar Pfahl, and Usman Qamar. Detection and correction of android-specific code smells and energy bugs: An android lint extension. In *8th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2020)*, 2020.

[54] Iffat Fatima, **Anwar, Hina**, Dietmar Pfahl, and Usman Qamar. Tool support for green android development: A systematic mapping study. In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT)*, number 1, pages 409–417, 2020.

[55] Thiago Soares Fernandes, Erika Cota, and Alvaro Freitas Moreira. Performance evaluation of android applications: a case study. In *2014 Brazilian Symposium on Computing Systems Engineering*, pages 79–84. IEEE, 2014.

[56] Ronald Aylmer Fisher and Frank Yates. *Statistical tables for biological, agricultural and medical research*. Edinburgh: Oliver and Boyd, 6th edition, 1963.

[57] Francesca Arcelli Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. On experimenting refactoring tools to remove code smells. New York, New York, USA, 2015. ACM Press.

[58] Francesca Arcelli Fontana, Elia Mariani, Andrea Mornioli, Raul Sormani, and Alberto Tonello. An experience report on using code smells detection tools. pages 450–457. IEEE, 3 2011.

[59] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[60] Jiaojiao Fu, Yangfan Zhou, Huan Liu, Yu Kang, and Xin Wang. Perman: fine-grained permission management for android applications. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 250–259. IEEE, 2017.

[61] Xing Gao, Dachuan Liu, Haining Wang, and Kun Sun. Pmdroid: Permission supervision for android advertising. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 120–129. IEEE, 2015.

[62] Saurabh Kumar Garg and Rajkumar Buyya. Green cloud computing and environmental sustainability. *Harnessing Green IT: Principles and Practices*, 2012:315–340, 2012.

[63] Franz-Xaver Geiger and Ivano Malavolta. Datasets of android applications: a literature review. *ArXiv*, 1809.10069, 9 2018.

[64] GeSI. SMARTer2030 ICT Solutions for 21st Century Challenges. Technical report, 2015.

[65] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. A double-edged sword? software reuse and potential security vulnerabilities. In *International Conference on Software and Systems Reuse*, pages 187–203. Springer, 2019.

[66] vogella GmbH. Using the okhttp library for http requests - tutorial - tutorial. [Online; accessed 2019-04-25].

[67] Vogella GmbH. Using retrofit 2.x as rest client - tutorial, 2018. [Online; accessed 2019-04-25].

[68] Olivier Le Goaër. Enforcing green code with android lint. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering Workshops*, ASE '20, page 85–90, New York, NY, USA, 2020. Association for Computing Machinery.

[69] Marion Gottschalk, Jan Jelschen, and Andreas Winter. Saving energy on mobile devices by refactoring. pages 437–444, 2014.

[70] PK Gupta et al. Minimizing power consumption by personal computers: A technical survey. *IJ Information Technology and Computer Science*, 2012.

[71] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 225–236, 2016.

[72] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of android code smells. In *Proceedings of the International Conference on Mobile Software Engineering and Systems*, MOBILESoft '16, page 59–69, New York, NY, USA, 2016. Association for Computing Machinery.

[73] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting antipatterns in android apps. In *2015 2nd ACM international conference on mobile software engineering and systems*, pages 148–149. IEEE, 2015.

[74] Abram Hindle. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20(2):374–409, April 2015.

[75] Abram Hindle, Alex Wilson, Kent Rasmussen, E. Jed Barlow, Joshua Charles Campbell, and Stephen Romansky. Greenminer: A hardware based mining software repositories software energy consumption

framework. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, page 12–21, New York, NY, USA, 2014. Association for Computing Machinery.

[76] Myles. Hollander, Douglas A. Wolfe, and Eric Chicken. *Nonparametric statistical methods.* WILEY, 3rd edition, 2014.

[77] Muhammad Ikram, Narseo Vallina-Rodriguez, Suranga Seneviratne, Mohamed Ali Kaafar, and Vern Paxson. An analysis of the privacy and security risks of android vpn permission-enabled apps. In *Proceedings of the 2016 Internet Measurement Conference*, pages 349–364, 2016.

[78] Square Inc. Retrofit. [Online; accessed 2019-04-25].

[79] Erik Jagroep, Jan Martijn van der Werf, Sjaak Brinkkemper, Leen Blom, and Rob van Vliet. Extending software architecture views with an energy consumption perspective. *Computing*, 99(6):553–573, 2017.

[80] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I Hong. Why are they collecting my data? inferring the purposes of network traffic in mobile apps. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(4):1–27, 2018.

[81] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Dongwon Kim, and Hojung Cha. Devscope: a nonintrusive and online power analysis tool for smartphone hardware components. In *CODES+ISSS '12*, 2012.

[82] Aman Kansal and Feng Zhao. Fine-grained energy profiling for power-aware application design. *ACM SIGMETRICS Performance Evaluation Review*, 36(2):26–31, 2008.

[83] Amandeep Kaur and Gaurav Dhiman. A review on search-based tools and techniques to identify bad code smells in object-oriented systems. volume 741, pages 909–921. Springer Verlag, 2019.

[84] Eva Kern, Lorenz M Hilty, Achim Guldner, Yuliyan V Maksimov, Andreas Filler, Jens Gröger, and Stefan Naumann. Sustainable software products—towards assessment criteria for resource and energy efficiency. *Future Generation Computer Systems*, 86:199–210, 2018.

[85] Eva Kern, Lorenz M. Hilty, Achim Guldner, Yuliyan V. Maksimov, Andreas Filler, Jens Gröger, and Stefan Naumann. Sustainable software products—towards assessment criteria for resource and energy efficiency. *Future Generation Computer Systems*, 86:199–210, 2018.

[86] Fauzia Khan, **Anwar, Hina**, Dietmar Pfahl, and Satish Srirama. Software techniques for making cloud data centers energy-efficient: A systematic

mapping study. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2020)*, 2020.

[87] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[88] Young-Woo Kwon and Eli Tilevich. Reducing the energy consumption of mobile applications behind the scenes. In *2013 IEEE International Conference on Software Maintenance*, pages 170–179. IEEE, 2013.

[89] Seokjun Lee, Minyoung Go, Rhan Ha, and Hojung Cha. Provisioning of energy consumption information for mobile ads. *Pervasive and Mobile Computing*, 53:49–61, 2019.

[90] Ding Li and William G. J. Halfond. Optimizing energy of http requests in android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile - DeMobile 2015*, pages 25–28. ACM Press, 2015.

[91] Ding Li and William GJ Halfond. An investigation into energy-saving programming practices for android smartphone app development. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, pages 46–53, 2014.

[92] Ding Li, Yingjun Lyu, Jiaping Gui, and William G. J. Halfond. Automated energy optimization of http requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*, pages 249–260. ACM Press, 2016.

[93] Li Li, Tegawende F. Bissyande, and Jacques Klein. Rebooting research on detecting repackaged android apps: Literature review and benchmark. *IEEE Transactions on Software Engineering*, pages 1–1, 2 2019.

[94] Li Li, Tegawendé F Bissyandé, Hao-Yu Wang, and Jacques Klein. On identifying and explaining similarities in android apps. *Journal of Computer Science and Technology*, 34(2):437–455, 2019.

[95] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. Static analysis of android apps: A systematic literature review. *Information and Software Technology*, 88:67–95, 8 2017.

[96] Mario Linares-Vásquez, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Reposito-*

*ries*, MSR 2014, page 2–11, New York, NY, USA, 2014. Association for Computing Machinery.

[97] Mario Linares-vásquez, Gabriele Bavota, Carlos Bernal-cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Optimizing Energy Consumption of GUIs in Android Apps : A Multi-objective Approach. *FSE'15 Proceeding of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*, pages 143–154, 2015.

[98] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. Revisiting android reuse studies in the context of code obfuscation and library usages. pages 242–251. ACM Press, 2014.

[99] Tomi Lämsä. *Comparison of GUI testing tools for Android applications*. PhD thesis, 2017.

[100] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. Libradar: Fast and accurate detection of third-party libraries in android apps. pages 653–656. ACM Press, 5 2016.

[101] Javier Mancebo, Félix García, and Coral Calero. A process for analysing the energy efficiency of software. *Information and Software Technology*, 134:106560, 2021.

[102] Irene Manotas, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause. An empirical study of practitioners' perspectives on green software engineering. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 237–248. IEEE, 2016.

[103] William Martin, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman. A survey of app store analysis for software engineering. *IEEE transactions on software engineering*, 43(9):817–847, 2016.

[104] Alejandro Mazuera-Rozo, Jairo Bautista-Mora, Mario Linares-Vásquez, Sandra Rueda, and Gabriele Bavota. The android os stack and its vulnerabilities: an empirical study. *Empirical Software Engineering*, 24(4):2056–2101, 2019.

[105] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 2004.

[106] Yana Momchilova Mileva, Valentin Dallmeier, Martin Burger, and Andreas Zeller. Mining trends of library usage. pages 57–61, 2009.

[107] Yana Momchilova Mileva, Valentin Dallmeier, and Andreas Zeller. Mining api popularity. volume 6303 LNCS, pages 173–180, 2010.

[108] Rodrigo Morales, Ruben Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. Earmo: An energy-aware refactoring approach for mobile apps. *IEEE Trans. Softw. Eng.*, 44(12):1176 – 1206, 2018.

[109] Irineu Moura, Gustavo Pinto, Felipe Ebert, and Fernando Castor. Mining energy-aware commits. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 56–67. IEEE, 2015.

[110] S. Murugesan and G. R. Gangadharan. *Green Cloud Computing and Environmental Sustainability*, pages 315–339. 2012.

[111] San Murugesan and GR Gangadharan. Green it: an overview. 2012.

[112] Joseph Yisa Ndagi and John K. Alhassan. Machine learning classification algorithms for adware in android devices: A comparative evaluation and analysis. pages 1–6. IEEE, 12 2019.

[113] Phuong T. Nguyen, Juri Di Rocco, Davide Di Ruscio, and Massimiliano Di Penta. Crossrec: Supporting software developers by recommending third-party libraries. *Journal of Systems and Software*, 161:110460, 2020.

[114] A. Noureddine, A. Bourdon, R. Rouvoy, and L. Seinturier. A preliminary study of the impact of software engineering on greenit. pages 21–27, 2012. cited By 56.

[115] Hiroki Ogawa, Eiji Takimoto, Koichi Mouri, and Shoichi Saito. User-side updating of third-party libraries for android applications. In *2018 Sixth International Symposium on Computing and Networking Workshops (CANDARW)*, pages 452–458. IEEE, 2018.

[116] Steffen Olbrich, Daniela S. Cruzes, Victor Basili, and Nico Zazworka. The evolution and impact of code smells: A case study of two open source systems. pages 390–400. IEEE, 10 2009.

[117] W. Oliveira, R. Oliveira, and F. Castor. A study on the energy consumption of android app development approaches. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 42–52, 2017.

[118] Wellington Oliveira, Renato Oliveira, and Fernando Castor. A study on the energy consumption of android app development approaches. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 42–52. IEEE, 2017.

[119] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, Takashi Ishio, Daniel M. German, and Katsuro Inoue. Search-based software library recommendation using multi-objective optimization. *Information and Software Technology*, 83:55–75, 2017.

[120] Shola Oyedeji, Ahmed Seffah, and Birgit Penzenstadler. A catalogue supporting software sustainability design. *Sustainability*, 10(7):2296, 2018.

[121] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *JSERD*, 5(1):7, 12 2017.

[122] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Lightweight detection of android-specific code smells: The adoctor project. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 487–491. IEEE, 2017.

[123] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. On the impact of code smells on the energy consumption of mobile applications. *Information and Software Technology*, 105:43–55, 2019.

[124] Abhinav Pathak, Y. Charlie Hu, and Ming Zhang. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, New York, NY, USA, 2011. Association for Computing Machinery.

[125] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42, 2012.

[126] Birgit Penzenstadler and Henning Femmer. A generic model for sustainability with process- and product-specific instances. GIBSE '13, page 3–8, New York, NY, USA, 2013. Association for Computing Machinery.

[127] Birgit Penzenstadler, Ankita Raturi, Debra Richardson, Coral Calero, Henning Femmer, and Xavier Franch. Systematic mapping study on software engineering for sustainability (se4s). In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, New York, NY, USA, 2014. Association for Computing Machinery.

[128] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery.

[129] K. Petersen, R. Feldt, Shahid Mujtaba, and Michael Mattsson. Systematic mapping studies in software engineering. In *EASE*, 2008.

[130] Gustavo Pinto and Fernando Castor. Energy efficiency: A new concern for application software developers. *Commun. ACM*, 60(12):68–75, November 2017.

[131] Gustavo H. Pinto and Fernando Kamei. What programmers say about refactoring tools? pages 33–36, New York, New York, USA, 2013. ACM Press.

[132] Ricardo Pérez-Castillo and Mario Piattini. Analyzing the harmful effect of god class refactoring on power consumption. *IEEE Software*, 31(3):48–54, 5 2014.

[133] Lina Qiu, Yingying Wang, and Julia Rubin. Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe. pages 176–186, New York, New York, USA, 2018. Association for Computing Machinery, Inc.

[134] Kent Rasmussen, Alex Wilson, and Abram Hindle. Green mining: Energy consumption of advertisement blocking methods. In *Proceedings of the 3rd International Workshop on Green and Sustainable Software*, GREENS 2014, page 38–45, New York, NY, USA, 2014. Association for Computing Machinery.

[135] Ghulam Rasool and Azhar Ali. Recovering android bad smells from android applications. *Arabian Journal for Science and Engineering*, 45(4):3289–3315, 2020.

[136] Reza Rawassizadeh. Mobile application benchmarking based on the resource usage monitoring. *International Journal of Mobile Computing and Multimedia Communications (IJMCMC)*, 1(4):64–75, 2009.

[137] Jan Reimann and Martin Brylski. A tool-supported quality smell catalogue for android developers. pages 14–15, 2014.

[138] dos Jose Pereira Reis, Fernando Brito e Abreu, and Glauco de F. Carneiro. Code smells incidence: Does it depend on the application domain? pages 172–177. IEEE, 9 2016.

[139] Haris Ribic and Yu David Liu. Energy-efficient work-stealing language runtimes. *ACM SIGARCH Computer Architecture News*, 42(1):513–528, 2014.

[140] Gilson Rocha, Fernando Castor, and Gustavo Pinto. Comprehending energy behaviors of java i/o apis. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2019.

[141] Ana Rodriguez, Mathias Longo, and Alejandro Zunino. Using bad smell-driven code refactorings in mobile applications to reduce battery usage. pages p.56–68, 2015.

[142] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 5 2009.

[143] Rui Rua, Marco Couto, and João Saraiva. Greensource: A large-scale collection of android code, tests and energy metrics. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, page 176–180. IEEE Press, 2019.

[144] C. Sahin, F. Cayci, I.L.M. Gutiérrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh. Initial explorations on design pattern energy usage. pages 55–61, 2012. cited By 73.

[145] Cagri Sahin, Lori Pollock, and James Clause. How do code refactorings affect energy usage? *8th ACM/IEEE International Symposium - ESEM '14*, pages 1–10, 2014.

[146] Mohamed Aymen Saied, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. Improving reusability of software libraries through usage pattern mining. *Journal of Systems and Software*, 145:164–179, 2018.

[147] Pasquale Salza, Fabio Palomba, Dario Di Nucci, Cosmo D'Uva, Andrea De Lucia, and Filomena Ferrucci. Do developers update third-party libraries in mobile apps? In *Proceedings of the 26th Conference on Program Comprehension*, pages 255–265, 2018.

[148] Yuru Shao, Ruowen Wang, Xun Chen, Ahemd M. Azab, and Z. Morley Mao. A lightweight framework for fine-grained lifecycle control of android applications. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[149] Satwinder Singh and Sharanpreet Kaur. A systematic literature review: Refactoring for disclosing code smells in object oriented software. *Ain Shams Engineering Journal*, 3 2017.

[150] Charlie Soh, Hee Beng Kuan Tan, Yauhen Leanidavich Arnatovich, and Lipo Wang. Detecting clones in android applications through analyzing user interfaces. In *2015 IEEE 23rd international conference on program comprehension*, pages 163–173. IEEE, 2015.

[151] n.d Square Inc. An http and http/2 client for android and java applications. [Online; accessed 2017-12-30].

[152] Nate Swanner. Users spend more money in apps as mobile web use declines, 2018.

[153] **Anwar, Hina**. Towards greener android application development. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 170–173. IEEE, 2020.

[154] **Anwar, Hina**, Berker Demirer, Dietmar Pfahl, and Satish Srirama. Should energy consumption influence the choice of android third-party http libraries? In *MOBILESoft '20: IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 87–97, 2020.

[155] **Anwar, Hina**, Iffat Fatima, Dietmar Pfahl, and Usman Qamar. *Tool Support for Green Android Development*, pages 153–182. Springer International Publishing, 2021.

[156] **Anwar, Hina** and Dietmar Pfahl. Towards greener software engineering using software analytics: A systematic mapping. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 157–166. IEEE, 2017.

[157] **Anwar, Hina**, Dietmar Pfahl, and Satish Narayana Srirama. An investigation into the energy consumption of http post request methods for android app development. In *13th International Conference on Software Technologies (ICSOFT 2018)*, pages 241–248, 2018.

[158] **Anwar, Hina**, Dietmar Pfahl, and Satish N Srirama. Evaluating the impact of code smell refactoring on the energy consumption of android applications. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 82–86. IEEE, 2019.

[159] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. pages 329–331. IEEE, 4 2008.

[160] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Ten years of jdeodorant: Lessons learned from the hunt for smells. IEEE, 2018.

[161] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. volume 1, pages 403–414. IEEE Press, 2015.

[162] Gias Uddin and Foutse Khomh. Automatic summarization of api reviews. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 159–170. IEEE, 2017.

[163] Roberto Verdecchia, René Aparicio Saez, Giuseppe Procaccianti, and Patricia Lago. Empirical evaluation of the energy impact of refactoring code smells. pages 365–383, 2018.

[164] D. Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, 2013.

[165] A. Vetrò, L. Ardito, G. Procaccianti, and M. Morisio. Definition, implementation and validation of energy code smells: an exploratory study on an embedded system. pages 34–39. ENERGY 2013-iaria, 2013.

[166] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. In *The 2014 ACM international conference on Measurement and modeling of computer systems*, pages 221–233, 2014.

[167] Nicolas Viennot, Edward Garcia, and Jason Nieh. A measurement study of google play. *SIGMETRICS Perform. Eval. Rev.*, 42(1):221–233, June 2014.

[168] Haoyu Wang and Yao Guo. Understanding third-party libraries in mobile app analysis. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 515–516. IEEE, 2017.

[169] Haoyu Wang, Yuanchun Li, Yao Guo, Yuvraj Agarwal, and Jason I Hong. Understanding the purpose of permission use in mobile apps. *ACM Transactions on Information Systems (TOIS)*, 35(4):1–40, 2017.

[170] Haoyu Wang, Zhe Liu, Yao Guo, Xiangqun Chen, Miao Zhang, Guoai Xu, and Jason Hong. An explorative study of the mobile app ecosystem from app developers' perspective. In *Proceedings of the 26th International Conference on World Wide Web*, pages 163–172, 2017.

[171] Haoyu Wang, Zhe Liu, Yao Guo, Xiangqun Chen, Miao Zhang, Guoai Xu, and Jason Hong. An explorative study of the mobile app ecosystem from app developers' perspective. WWW '17, page 163–172, Republic and Canton of Geneva, CHE, 2017. International World Wide Web Conferences Steering Committee.

[172] Haoyu Wang, Zhe Liu, Jingyue Liang, Narseo Vallina-Rodriguez, Yao Guo, Li Li, Juan Tapiador, Jingcun Cao, and Guoai Xu. Beyond google play: A large-scale comparative study of chinese android app markets. In *Proceedings of the Internet Measurement Conference 2018*, pages 293–307, 2018.

[173] Yimeng Wang, Yongbo Li, and Tian Lan. Capitalizing on the promise of ad prefetching in real-world mobile systems. In *2017 IEEE 14th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*, pages 162–170, 2017.

[174] Bowen Xu, Le An, Ferdian Thung, Foutse Khomh, and David Lo. Why

reinventing the wheels? an empirical study on library reuse and re-implementation. *Empirical Software Engineering*, 25(1):755–789, 2020.

[175] Aiko Yamashita and Steve Counsell. Code smells as system-level indicators of maintainability: An empirical study. *JSS*, 86(10):2639–2653, 10 2013.

[176] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. pages 242–251. IEEE, 10 2013.

[177] Tatsuhiko Yasumatsu, Takuya Watanabe, Fumihiro Kanei, Eitaro Shioji, Mitsuaki Akiyama, and Tatsuya Mori. Understanding the responsiveness of mobile app developers to software library updates. In *Proceedings of the Ninth ACM Conference on Data and Application Security and Privacy*, CODASPY '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery.

[178] Li Yuan. Detecting similar components between android applications with obfuscation. In *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*, pages 186–190. IEEE, 2016.

[179] Jiawei Zhan, Quan Zhou, Xiaozhuo Gu, Yuewu Wang, and Yingjiao Niu. Splitting third-party libraries' privileges from android apps. In *Australasian Conference on Information Security and Privacy*, pages 80–94. Springer, 2017.

[180] Lide Zhang, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P. Dick, Zhuoqing Morley Mao, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES/ISSS '10, page 105–114, New York, NY, USA, 2010. Association for Computing Machinery.

[181] Yaocheng Zhang, Wei Ren, Tianqing Zhu, and Yi Ren. Saas: A situational awareness and analysis system for massive android malware detection. *Future Generation Computer Systems*, 95:548–559, 2019.

# Appendix A. STATISTICS AND PAIRWISE COMPARISON RESULTS FOR SELECTED THIRD-PARTY HTTP LIBRARIES

**Additional Material- Chapter 5**

## A.1. Statistics for Selected Third-party HTTP Libraries

In table A.1 quality ranking provided by 'Awesome Android' website is done by a third party that measure the structural quality with focus on maintainability, Each project is compared to other projects of equal size, language and ranking is given based on the ratio of highly unmaintainable code to the total size of the project.

In Table A.2 the percentage of apps means that a particular library is present in most of the apps while percentage of installs indicates the presence of library in most installs on actual Android devices. These statistics are crowd sourced i.e. the more popular the app the more coverage it has in terms of the libraries it is using. For less popular apps the libraries might not be detected. But these statistics can be considered relevant as large number of downloads are generated by relatively small number of apps.

Table A.1: Statistics for selected libraries gathered from Awesome Android website (Oct. 2019)

| ID. | Library | Stars | Watchers | Forks | Quality* | Launched |
|---|---|---|---|---|---|---|
| vo | Volley | 2.5k | 140 | 611 | L4 | 2017 |
| re | Retrofit | 33k | 1.6k | 6.3k | L1 | 2010 |
| ok | OkHttp | 34k | 1.7k | 7.6k | L2 | 2012 |
| async- h | Androidasynchttp | 10.5k | 810 | 4.3k | L4 | 2011 |
| async | Androidasync | 6.4k | 418 | 1.4k | L3 | 2012 |
| pic | Picasso | 17.1k | 937 | 4k | L2 | 2013 |
| uil | UIL | 16.6k | 1.4k | 6.4k | L3 | 2011 |
| gli | Glide | 27.4k | 1.1k | 5k | L5 | 2013 |

*Quality levels used on Awesome Android website (calculated by a third party, Luminous). L1= initial stage of development, highly unmaintainable. L2= not initial stage but still require refactoring to improve low quality. L3=medium structural quality. L4=good structural quality, maintenance cost under control. L5= fine-tuned and optimized structural quality.

Table A.2: Statistics for selected libraries gathered from the ‘AppBrain’ website. (Oct. 2019)

| ID. | Library | ‘AppBrain’ Statistics for overall apps in the Google Play Store | | ‘AppBrain’ Statistics for Top 500 apps in the Google Play Store | | ‘AppBrain’ Statistics overall | |
|---|---|---|---|---|---|---|---|
| | | % of apps using the lib | % of installs on devices | % of apps using the lib | % of installs on devices | Tags | # of apps using the lib |
| vo | Volley | 7.61% | 14.89% | 17% | 12.82% | Network, Image Loader | >62K |
| re | Retrofit | 11.87% | 20.37% | 55.60% | 28.78% | Network | >97K |
| ok | OkHttp | 4.63% | 6.86% | 7.40% | 8.16% | Network | >38K |
| async- h | Androidasynchttp | 2.70% | 2.87% | 4% | 0.66% | Network* | >22k |
| async | Androidasync | 0.88% | 0.48% | 0.80% | 0.18% | Network | >7K |
| pic | Picasso | 16.51% | 16.80% | 34% | 20.44% | Image Loader | >135K |
| uil | UIL | 4.30% | 3.28% | 2.40% | 0.78% | Image Loader | >35K |
| gli | Glide | 16.16% | 29.25% | 31% | 27.75% | Image Loader | >132K |

*only ‘network’ tag is assigned to the androidasynchttp library on ‘AppBrain’ website, however, we identified that it also performs image loading tasks.

## A.2. Features and Methods of Selected Third-party HTTP Libraries

Besides basic HTTP methods like GET and POST the third-party HTTP libraries also offer additional benefits like caching system, parallel request, support for HTTP/2, etc. Some HTTP libraries are for specific tasks like image loading.

Table A.3: HTTP request methods

| HTTP method | Description | Request has Body | Response has Body | Safe | Idempotent | Cacheable |
|---|---|---|---|---|---|---|
| **GET** | Only retrieve data and should have no other effect | Optional | Yes | Yes | Yes | Yes |
| **HEAD** | GET request, but without the response body. This is useful for retrieving meta , information written in response headers, without having to transport the entire content. | Optional | No | Yes | Yes | Yes |
| **POST** | Requests that the server accept the entity enclosed in the request as a new entry | Yes | Yes | No | No | Yes |
| **PUT** | Already existing resource is modified | Yes | Yes | No | Yes | No |
| **DELETE** | Deletes the specified resource | Optional | Yes | No | Yes | No |
| **CONNECT** | Converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL , encrypted communication (HTTPS) through an unencrypted HTTP proxy | Optional | Yes | No | No | No |
| **OPTIONS** | Returns the HTTP methods that the server supports for the specified URL | Optional | Yes | Yes | Yes | No |
| **TRACE** | Echoes the received request so that a client can see what (if any) changes or additions have been made by intermediate servers | No | Yes | Yes | Yes | No |
| **PATCH** | Applies partial modifications to a resource | Yes | Yes | No | No | No |

Table A.4: Features in selected network third-party libraries

| | OkHttp | Retrofit | Volley | Androidasynchttp | Androidasync |
|---|---|---|---|---|---|
| Cancel requests | – | ✓ | ✓ | – | – |
| Async Requests | ✓ | ✓ | ✓ | ✓ | ✓ |
| POST Request | ✓ | ✓ | ✓ | ✓ | ✓ |
| Delete Request | – | ✓ | – | – | – |
| GET Request | ✓ | ✓ | ✓ | ✓ | ✓ |
| Multiple Request | – | – | ✓ | – | – |
| Multipart uploads | – | ✓ | ✓ | ✓ | ✓ |
| Connection pooling | ✓ | – | ✓ | ✓ | – |
| Caching of responses | – | – | ✓ | ✓ | – |
| Retry policy | ✓ | – | ✓ | ✓ | – |
| Load and transform images | – | – | ✓ | ✓ | – |
| JSON | ✓ | ✓ | ✓ | ✓ | ✓ |
| SPDY, HTTP/2 support | ✓ | ✓ | – | – | – |
| Handle redirects | ✓ | – | – | ✓ | – |
| Gzip response decoding | ✓ | – | – | ✓ | – |

Table A.5: Features in selected image loading third-party libraries

| | Picasso | Universal image loader(UIL) | Glide |
|---|---|---|---|
| Async | ✓ | ✓ | ✓ |
| GET Request | ✓ | ✓ | ✓ |
| Disk Caching | ✓ | ✓ | ✓ |
| Load image | ✓ | ✓ | ✓ |
| Video stills | – | – | ✓ |
| Animated GIFs | – | – | ✓ |
| Displaying image | ✓ | ✓ | ✓ |
| Image cropping | ✓ | ✓ | ✓ |

## A.3. Results of Pairwise Comparison for the Mean Ranks of Energy by Library in Each Use-case

In the following tables observed differences > critical differences indicate significance at the $p < 0.05$ level. Significantly different pairs of libraries are highlighted in blue color.

Table A.6: Pairwise comparisons for the mean ranks of energy by library (UC-GF)

| Comparison | Observed Difference | Critical Difference |
|---|---|---|
| re , ok | 3.00 | 18.30 |
| vo , async-h | 4.00 | 18.30 |
| ok , async | 13.50 | 18.30 |
| re , async-h | 16.50 | 18.30 |
| re , async | 16.50 | 18.30 |
| ok , async-h | 19.50 | 18.30 |
| vo , re | 20.50 | 18.30 |

| vo , ok | 23.50 | 18.30 |
|---|---|---|
| async-h , async | 33.00 | 18.30 |
| vo , async | 37.00 | 18.30 |

Table A.7: Pairwise comparisons for the mean ranks of energy by library (UC-PF)

| Comparison | Observed Difference | Critical Difference |
|---|---|---|
| ok , async-h | 2.80 | 18.30 |
| re , async | 6.80 | 18.30 |
| ok , async | 8.20 | 18.30 |
| async-h , async | 11.00 | 18.30 |
| re , ok | 15.00 | 18.30 |
| vo , async-h | 17.10 | 18.30 |
| re , async-h | 17.80 | 18.30 |
| vo , ok | 19.90 | 18.30 |
| Vo , async | 28.10 | 18.30 |
| vo , re | 34.90 | 18.30 |

Table A.8: Pairwise comparisons for the mean ranks of energy by library (UC-PJO)

| Comparison | Observed Difference | Critical Difference |
|---|---|---|
| ok(M) , async(G) | 1.80 | 66.98 |
| async(G) , async(M) | 2.00 | 66.98 |
| re(M) , ok(G) | 2.20 | 66.98 |
| async-h(J) , async(J) | 3.40 | 66.98 |
| ok(J) , async-h(G) | 3.50 | 66.98 |
| async-h(G) , async-h(M) | 3.60 | 66.98 |
| ok(M) , async(M) | 3.80 | 66.98 |
| vo(M) , vo(J) | 6.58 | 68.82 |
| ok(J) , async-h(M) | 7.10 | 66.98 |
| vo(G) , vo(J) | 8.22 | 68.82 |
| ok(M) , async-h(M) | 8.80 | 66.98 |
| vo(G) , re(J) | 9.44 | 68.82 |
| re(J) , async-h(J) | 9.66 | 68.82 |
| async-h(M) , async(G) | 10.60 | 66.98 |
| ok(M) , async-h(G) | 12.40 | 66.98 |
| async-h(M) , async(M) | 12.60 | 66.98 |
| re(J) , async(J) | 13.06 | 68.82 |
| re(G) , re(M) | 13.10 | 66.98 |

| | | |
|---|---|---|
| async-h(G) , async(G) | 14.20 | 66.98 |
| vo(G) , vo(M) | 14.80 | 66.98 |
| re(G) , ok(G) | 15.30 | 66.98 |
| ok(M) , ok(J) | 15.90 | 66.98 |
| async-h(G) , async(M) | 16.20 | 66.98 |
| vo(J) , re(J) | 17.67 | 70.60 |
| ok(J) , async(G) | 17.70 | 66.98 |
| vo(G) , async-h(J) | 19.10 | 66.98 |
| ok(G) , async(M) | 19.10 | 66.98 |
| ok(J) , async(M) | 19.70 | 66.98 |
| ok(G) , async(G) | 21.10 | 66.98 |
| re(M) , async(M) | 21.30 | 66.98 |
| vo(G) , async(J) | 22.50 | 66.98 |
| ok(G) , ok(M) | 22.90 | 66.98 |
| re(M) , async(G) | 23.30 | 66.98 |
| vo(M) , re(J) | 24.24 | 68.82 |
| re(M) , ok(M) | 25.10 | 66.98 |
| ok(J) , async(J) | 25.60 | 66.98 |
| vo(J) , async-h(J) | 27.32 | 68.82 |
| ok(J) , async-h(J) | 29.00 | 66.98 |
| async-h(G) , async(J) | 29.10 | 66.98 |
| vo(J) , async(J) | 30.72 | 68.82 |
| ok(G) , async-h(M) | 31.70 | 66.98 |
| async-h(G) , async-h(J) | 32.50 | 66.98 |
| async-h(M) , async(J) | 32.70 | 66.98 |
| vo(M) , async-h(J) | 33.90 | 66.98 |
| re(M) , async-h(M) | 33.90 | 66.98 |
| re(G) , async(M) | 34.40 | 66.98 |
| ok(G) , async-h(G) | 35.30 | 66.98 |
| async-h(M) , async-h(J) | 36.10 | 66.98 |
| re(G) , async(G) | 36.40 | 66.98 |
| vo(M) , async(J) | 37.30 | 66.98 |
| re(M) , async-h(G) | 37.50 | 66.98 |
| re(G) , ok(M) | 38.20 | 66.98 |
| re(J) , ok(J) | 38.66 | 68.82 |
| ok(G) , ok(J) | 38.80 | 66.98 |
| re(M) , ok(J) | 41.00 | 66.98 |
| ok(M) , async(J) | 41.50 | 66.98 |
| re(J) , async-h(G) | 42.16 | 68.82 |
| async(G) , async(J) | 43.30 | 66.98 |

| | | |
|---|---|---|
| ok(M) , async-h(J) | 44.90 | 66.98 |
| async(M) , async(J) | 45.30 | 66.98 |
| re(J) , async-h(M) | 45.76 | 68.82 |
| async-h(J) , async(G) | 46.70 | 66.98 |
| re(G) , async-h(M) | 47.00 | 66.98 |
| vo(G) , ok(J) | 48.10 | 66.98 |
| async-h(J) , async(M) | 48.70 | 66.98 |
| re(G) , async-h(G) | 50.60 | 66.98 |
| vo(G) , async-h(G) | 51.60 | 66.98 |
| re(G) , ok(J) | 54.10 | 66.98 |
| re(J) , ok(M) | 54.56 | 68.82 |
| vo(G) , async-h(M) | 55.20 | 66.98 |
| vo(J) , ok(J) | 56.32 | 68.82 |
| re(J) , async(G) | 56.36 | 68.82 |
| re(J) , async(M) | 58.36 | 68.82 |
| vo(J) , async-h(G) | 59.82 | 68.82 |
| vo(M) , ok(J) | 62.90 | 66.98 |
| vo(J) , async-h(M) | 63.42 | 68.82 |
| vo(G) , ok(M) | 64.00 | 66.98 |
| ok(G) , async(J) | 64.40 | 66.98 |
| vo(G) , async(G) | 65.80 | 66.98 |
| vo(M) , async-h(G) | 66.40 | 66.98 |
| re(M) , async(J) | 66.60 | 66.98 |
| vo(G) , async(M) | 67.80 | 66.98 |
| ok(G) , async-h(J) | 67.80 | 66.98 |
| vo(M) , async-h(M) | 70.00 | 66.98 |
| re(M) , async-h(J) | 70.00 | 66.98 |
| vo(J) , ok(M) | 72.22 | 68.82 |
| vo(J) , async(G) | 74.02 | 68.82 |
| vo(J) , async(M) | 76.02 | 68.82 |
| re(J) , ok(G) | 77.46 | 68.82 |
| vo(M) , ok(M) | 78.80 | 66.98 |
| re(M) , re(J) | 79.66 | 68.82 |
| re(G) , async(J) | 79.70 | 66.98 |
| vo(M) , async(G) | 80.60 | 66.98 |
| vo(M) , async(M) | 82.60 | 66.98 |
| re(G) , async-h(J) | 83.10 | 66.98 |
| vo(G) , ok(G) | 86.90 | 66.98 |
| vo(G) , re(M) | 89.10 | 66.98 |
| re(G) , re(J) | 92.76 | 68.82 |

| | | |
|---|---|---|
| vo(J) , ok(G) | 95.12 | 68.82 |
| vo(J) , re(M) | 97.32 | 68.82 |
| vo(M) , ok(G) | 101.70 | 66.98 |
| vo(G) , re(G) | 102.20 | 66.98 |
| vo(M) , re(M) | 103.90 | 66.98 |
| vo(J) , re(G) | 110.42 | 68.82 |
| vo(M) , re(G) | 117.00 | 66.98 |

Table A.9: Pairwise comparisons for the mean ranks of energy by library (UC-GJO)

| Comparison | Observed Difference | Critical Difference |
|---|---|---|
| async(M) , async-h(J) | 0.50 | 67.43 |
| re(G) , vo(G) | 5.10 | 67.43 |
| ok(M) , re(M) | 5.70 | 67.43 |
| vo(M) , re(M) | 6.70 | 67.43 |
| re(J) , ok(J) | 7.10 | 67.43 |
| async-h(G) , async-h(M) | 9.48 | 65.88 |
| async(G) , async(M) | 9.63 | 69.28 |
| re(J) , vo(J) | 9.90 | 67.43 |
| async(G) , async-h(J) | 10.13 | 69.28 |
| ok(G) , vo(J) | 10.34 | 69.28 |
| re(G) , ok(M) | 11.10 | 67.43 |
| ok(M) , vo(M) | 12.40 | 67.43 |
| vo(G) , ok(G) | 12.46 | 69.28 |
| async(G) , async(J) | 13.17 | 69.28 |
| async-h(G) , async-h(J) | 16.02 | 65.88 |
| vo(G) , ok(M) | 16.20 | 67.43 |
| async-h(G) , async(M) | 16.52 | 65.88 |
| re(G) , re(M) | 16.80 | 67.43 |
| vo(M) , async(J) | 17.00 | 67.43 |
| vo(J) , ok(J) | 17.00 | 67.43 |
| re(G) , ok(G) | 17.56 | 69.28 |
| ok(G) , re(J) | 20.24 | 69.28 |
| vo(G) , re(M) | 21.90 | 67.43 |
| vo(G) , vo(J) | 22.80 | 67.43 |
| async(M) , async(J) | 22.80 | 67.43 |
| async-h(J) , async(J) | 23.30 | 67.43 |
| re(G) , vo(M) | 23.50 | 67.43 |
| re(M) , async(J) | 23.70 | 67.43 |

| | | |
|---|---|---|
| async-h(M) , async-h(J) | 25.50 | 67.43 |
| async(M) , async-h(M) | 26.00 | 67.43 |
| async-h(G) , async(G) | 26.15 | 67.77 |
| ok(G) , ok(J) | 27.34 | 69.28 |
| re(G) , vo(J) | 27.90 | 67.43 |
| vo(G) , vo(M) | 28.60 | 67.43 |
| ok(G) , ok(M) | 28.66 | 69.28 |
| ok(M) , async(J) | 29.40 | 67.43 |
| async(G) , vo(M) | 30.17 | 69.28 |
| vo(G) , re(J) | 32.70 | 67.43 |
| ok(G) , re(M) | 34.36 | 69.28 |
| async(G) , async-h(M) | 35.63 | 69.28 |
| async(G) , re(M) | 36.87 | 69.28 |
| re(G) , re(J) | 37.80 | 67.43 |
| ok(M) , vo(J) | 39.00 | 67.43 |
| async-h(G) , async(J) | 39.32 | 65.88 |
| vo(G) , ok(J) | 39.80 | 67.43 |
| async(M) , vo(M) | 39.80 | 67.43 |
| vo(M) , async-h(J) | 40.30 | 67.43 |
| re(G) , async(J) | 40.50 | 67.43 |
| ok(G) , vo(M) | 41.06 | 69.28 |
| async(G) , ok(M) | 42.57 | 69.28 |
| re(M) , vo(J) | 44.70 | 67.43 |
| re(G) , ok(J) | 44.90 | 67.43 |
| vo(G) , async(J) | 45.60 | 67.43 |
| async(M) , re(M) | 46.50 | 67.43 |
| re(M) , async-h(J) | 47.00 | 67.43 |
| async-h(M) , async(J) | 48.80 | 67.43 |
| ok(M) , re(J) | 48.90 | 67.43 |
| vo(M) , vo(J) | 51.40 | 67.43 |
| async(M) , ok(M) | 52.20 | 67.43 |
| ok(M) , async-h(J) | 52.70 | 67.43 |
| re(G) , async(G) | 53.67 | 69.28 |
| re(M) , re(J) | 54.60 | 67.43 |
| ok(M) , ok(J) | 56.00 | 67.43 |
| async-h(G) , vo(M) | 56.32 | 65.88 |
| ok(G) , async(J) | 58.06 | 69.28 |
| vo(G) , async(G) | 58.77 | 69.28 |
| vo(M) , re(J) | 61.30 | 67.43 |
| re(M) , ok(J) | 61.70 | 67.43 |

| | | |
|---|---|---|
| async-h(G) , re(M) | 63.02 | 65.88 |
| re(G) , async(M) | 63.30 | 67.43 |
| re(G) , async-h(J) | 63.80 | 67.43 |
| async-h(M) , vo(M) | 65.80 | 67.43 |
| vo(G) , async(M) | 68.40 | 67.43 |
| vo(M) , ok(J) | 68.40 | 67.43 |
| vo(J) , async(J) | 68.40 | 67.43 |
| async-h(G) , ok(M) | 68.72 | 65.88 |
| vo(G) , async-h(J) | 68.90 | 67.43 |
| ok(G) , async(G) | 71.22 | 71.08 |
| async-h(M) , re(M) | 72.50 | 67.43 |
| async-h(M) , ok(M) | 78.20 | 67.43 |
| re(J) , async(J) | 78.30 | 67.43 |
| re(G) , async-h(G) | 79.82 | 65.88 |
| ok(G) , async(M) | 80.86 | 69.28 |
| ok(G) , async-h(J) | 81.36 | 69.28 |
| async(G) , vo(J) | 81.57 | 69.28 |
| vo(G) , async-h(G) | 84.92 | 65.88 |
| ok(J) , async(J) | 85.40 | 67.43 |
| re(G) , async-h(M) | 89.30 | 67.43 |
| async(M) , vo(J) | 91.20 | 67.43 |
| async(G) , re(J) | 91.47 | 69.28 |
| vo(J) , async-h(J) | 91.70 | 67.43 |
| vo(G) , async-h(M) | 94.40 | 67.43 |
| ok(G) , async-h(G) | 97.37 | 67.77 |
| async(G) , ok(J) | 98.57 | 69.28 |
| async(M) , re(J) | 101.10 | 67.43 |
| re(J) , async-h(J) | 101.60 | 67.43 |
| ok(G) , async-h(M) | 106.86 | 69.28 |
| async-h(G) , vo(J) | 107.72 | 65.88 |
| async(M) , ok(J) | 108.20 | 67.43 |
| ok(J) , async-h(J) | 108.70 | 67.43 |
| async-h(M) , vo(J) | 117.20 | 67.43 |
| async-h(G) , re(J) | 117.62 | 65.88 |
| async-h(G) , ok(J) | 124.72 | 65.88 |
| async-h(M) , re(J) | 127.10 | 67.43 |
| async-h(M) , ok(J) | 134.20 | 67.43 |

Table A.10: Pairwise comparisons for the mean ranks of energy by library (UC-GI)

| Comparison | Observed Difference | Critical Difference |
|---|---|---|
| pic , gli | 2.10 | 18.30 |
| vo , async-h | 4.40 | 18.30 |
| uil , pic | 10.50 | 18.30 |
| uil , gli | 12.60 | 18.30 |
| async-h , gli | 17.90 | 18.30 |
| async-h , pic | 20.00 | 18.30 |
| vo , gli | 22.30 | 18.30 |
| vo , pic | 24.40 | 18.30 |
| async-h , uil | 30.50 | 18.30 |
| vo , uil | 34.90 | 18.30 |

# Appendix B. LIST OF SELECTED PUBLICATIONS, ENERGY BUGS, AND CODE SMELLS

**Additional Material- Chapter 6**

## B.1. List of Selected Publications

| Study ID | Reference |
|---|---|
| P1 | Wu, H., Yang, S., & Rountev, A. (2016). Static detection of energy defect patterns in Android applications. In Proceedings of the 25th International Conference on Compiler Construction - CC 2016 (pp.185–195). New York, New York, USA: ACM Press. https://doi.org/10.1145/2892208.2892218 |
| P2 | Cai, H., Zhang, Y., Jin, Z., Liu, X., & Huang, G. (2015). Delay-Droid: Reducing Tail-Time Energy by Refactoring Android Apps. In Proceedings of the 7th Asia-Pacific Symposium on Internetware - Internetware '15 (pp. 1–10). New York, New York, USA: ACM Press. https://doi.org/10.1145/2875913.2875915 |
| P3 | Kim, C. H. P., Kroening, D., & Kwiatkowska, M. (2016). Static Program Analysis for Identifying Energy Bugs in Graphics-Intensive Mobile Apps. In 2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS) (pp. 115–124). IEEE. https://doi.org/10.1109/MASCOTS.2016.28 |
| P4 | Palomba, F., Nucci, D. Di, Panichella, A., Zaidman, A., & De Lucia, A. (2017). Lightweight Detection of Android-specific Code Smells: the aDoctor Project. in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb 2017, pp. 487–491. |
| P5 | Xu, Z., Wen, C., & Qin, S. (2018). State-taint analysis for detecting resource bugs. Science of Computer Programming, 162, 93–109. https://doi.org/10.1016/j.scico.2017.06.010 |
| P6 | Westfield, B., & Gopalan, A. (2016). Orka: A new technique to profile the energy usage of android applications. In SMARTGREENS 2016 - Proceedings of the 5th International Conference on Smart Cities and Green ICT Systems (pp. 213–224). SciTePress. https://doi.org/10.5220/0005812202130224 |
| P7 | Hecht, G., Rouvoy, R., Moha, N., & Duchien, L. (2015). Detecting Antipatterns in Android Apps. In Proceedings - 2nd ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft 2015 (pp. 148–149). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/MobileSoft.2015.38 |
| P8 | Liang, G., Wang, J., Li, S., & Chang, R. (2014). PatBugs: A Pattern-Based Bug Detector for Cross-platform Mobile Applications. In 2014 IEEE International Conference on Mobile Services (pp. 84–91). IEEE. https://doi.org/10.1109/MobServ.2014.21 |

| | |
|---|---|
| P9 | Jiang, H., Yang, H., Qin, S., Su, Z., Zhang, J., & Yan, J. (2017). Detecting Energy Bugs in Android Apps Using Static Analysis. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 10610 LNCS, pp. 192–208). Springer Verlag. https://doi.org/10.1007/978-3-319-68690-5_12 |
| P10 | Carette, A., Younes, M. A. A., Hecht, G., Moha, N., & Rouvoy, R. (2017). Investigating the energy impact of Android smells. In 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 115–126). IEEE. https://doi.org/10.1109/SANER.2017.7884614 |
| P11 | Lin, Y., Okur, S., & Dig, D. (2015). Study and Refactoring of Android Asynchronous Programming (T). In 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 224–235). IEEE. https://doi.org/10.1109/ASE.2015.50 |
| P12 | Barde, K., Kulkarni, J., Bloch, S., Luo, C., Hanumaiah, V., Kim, E., & Patel, H. (2015). SEPIA: A framework for optimizing energy consumption in Android devices. In 2015 12th Annual IEEE Consumer Communications and Networking Conference (CCNC) (pp. 562–569). IEEE. https://doi.org/10.1109/CCNC.2015.7158035 |
| P13 | Kwon, Y., Lee, S., Yi, H., Kwon, D., Yang, S., Chun, B., Paek, Y. (2015). Mantis: Efficient Predictions of Execution Time, Energy Usage, Memory Usage and Network Usage on Smart Mobile Devices. IEEE Transactions on Mobile Computing, 14(10), 2059–2072. https://doi.org/10.1109/TMC.2014.2374153 |
| P14 | Chen, X., & Zong, Z. (2016). Android App Energy Efficiency: The Impact of Language, Runtime, Compiler, and Implementation. 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) (pp. 485–492). IEEE. https://doi.org/10.1109/BDCloud-SocialCom-SustainCom.2016.77 |
| P15 | Morales, R., Saborido, R., Khomh, F., Chicano, F., & Antoniol, G. (2018). EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. IEEE Trans. Softw. Eng., 44(1), 1176–1206. https://doi.org/10.1109/TSE.2017.2757486 |
| P16 | Wang, C., Guo, Y., Shen, P., & Chen, X. (2017). E-Spector: Online energy inspection for Android applications. In 2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED) (pp. 1–6). IEEE. https://doi.org/10.1109/ISLPED.2017.8009207 |
| P17 | Yepang Liu, Chang Xu, Cheung, S. C., & Jian Lu. (2014). GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications. IEEE Transactions on Software Engineering, 40(9), 911–940. https://doi.org/10.1109/TSE.2014.2323982 |
| P18 | Banerjee, A., Chong, L. K., Ballabriga, C., & Roychoudhury, A. (2018). EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. IEEE Transactions on Software Engineering, 44(5), 470–490. https://doi.org/10.1109/TSE.2017.2689012 |

| | |
|---|---|
| P19 | Couto, M., Carção, T., Cunha, J., Fernandes, J. P., & Saraiva, J. (2014). Detecting anomalous energy consumption in android applications. In Brazilian Symposium on Programming Languages (Vol. 8771 LNCS, pp. 77–91). Springer Verlag. https://doi.org/10.1007/978-3-319-11863-5_6 |
| P20 | Fischer, L. M., Brisolara, L. B. de, & Mattos, J. C. B. de. (2015). SEMA: An Approach Based on Internal Measurement to Evaluate Energy Efficiency of Android Applications. In 2015 Brazilian Symposium on Computing Systems Engineering (SBESC) (pp. 48–53). IEEE. https://doi.org/10.1109/SBESC.2015.16 |
| P21 | Nguyen, M. D., Huynh, T. Q., & Nguyen, T. H. (2016). Improve the performance of mobile applications based on code optimization techniques using PMD and android lint. In International Symposium on Integrated Uncertainty in Knowledge Modelling and Decision Making,IUKM 2016. Lecture Notes in Computer Science (Vol. 9978 LNAI, pp. 343–356). Springer Verlag. https://doi.org/10.1007/978-3-319-49046-5_29 |
| P22 | M. Couto, J. Saraiva and J. P. Fernandes, Energy Refactorings for Android in the Large and in the Wild, 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 2020, pp. 217-228, doi: 10.1109/SANER48275.2020.9054858. |
| P23 | Wei Song, Jing Zhang, and Jeff Huang. 2019. ServDroid: detecting service usage inefficiencies in Android applications. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019). Association for Computing Machinery, New York, NY, USA, 362–373, doi:https://doi.org/10.1145/3338906.3338950 |
| P24 | Cruz, Luís & Abreu, Rui. (2018). Using Automatic Refactoring to Improve Energy Efficiency of Android Apps. |
| P25 | M. Sun and G. Tan, ‘NativeGuard: Protecting android applications from third-party native libraries’, in WiSec 2014 - Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks, New York, NY, USA, 2014, pp. 165–176, doi: 10.1145/2627393.2627396. |
| P26 | W. Hu, D. Octeau, P. D. McDaniel, and P. Liu, ‘Duet’, in Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks - WiSec ’14, New York, New York, USA, 2014, pp. 141–152, doi: 10.1145/2627393.2627404. |
| P27 | A. Narayanan, L. Chen, and C. K. Chan, ‘AdDetect: Automated detection of Android ad libraries using semantic analysis’, in 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Apr. 2014, pp. 1–6, doi: 10.1109/ISSNIP.2014.6827639. |
| P28 | B. Liu, B. Liu, H. Jin, and R. Govindan, ‘Efficient privilege de-escalation for ad libraries in mobile apps’, in MobiSys 2015 - Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, New York, New York, USA, May 2015, pp. 89–103, doi: 10.1145/2742647.2742668. |

| | |
|---|---|
| P29 | J. Crussell, C. Gibler, and H. Chen, 'AnDarwin: Scalable Detection of Android Application Clones Based on Semantics', IEEE Transactions on Mobile Computing, vol. 14, no. 10, pp. 2007–2019, Oct. 2015, doi: 10.1109/TMC.2014.2381212. |
| P30 | M. Backes, S. Bugiel, and E. Derr, 'Reliable third-party library detection in Android and its security applications', in Proceedings of the ACM Conference on Computer and Communications Security, New York, New York, USA, Oct. 2016, vol. 24-28-Octo, pp. 356–367, doi: 10.1145/2976749.2978333. |
| P31 | B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, 'Statistical deobfuscation of Android applications', in Proceedings of the ACM Conference on Computer and Communications Security, New York, NY, USA, 2016, vol. 24-28-Octo, pp. 343–355, doi: 10.1145/2976749.2978422. |
| P32 | C. Soh, H. B. Kuan Tan, Y. L. Arnatovich, A. Narayanan, and L. Wang, 'LibSift: Automated Detection of Third-Party Libraries in Android Applications', in 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), 2016, vol. 0, pp. 41–48, doi: 10.1109/APSEC.2016.017. |
| P33 | Z. Ma, H. Wang, Y. Guo, and X. Chen, 'LibRadar: Fast and accurate detection of third-party libraries in Android apps', in Proceedings - International Conference on Software Engineering, May 2016, pp. 653–656, doi: 10.1145/2889160.2889178. |
| P34 | F. Wang, Y. Zhang, K. Wang, P. Liu, and W. Wang, 'Stay in Your Cage! A Sound Sandbox for Third-Party Libraries on Android', in Computer Security – ESORICS 2016, I. Askoxylakis, S. Ioannidis, S. Katsikas, and C. Meadows, Eds. Cham: Springer International Publishing, 2016, pp. 458–476. |
| P35 | H. Yu, X. Xia, X. Zhao, and W. Qiu, 'Combining collaborative filtering and topic modeling for more accurate android mobile app library recommendation', in ACM International Conference Proceeding Series, New York, New York, USA, 2017, vol. Part F1309, pp. 1–6, doi: 10.1145/3131704.3131721. |
| P36 | J. Zhan, Q. Zhou, X. Gu, Y. Wang, and Y. Niu, 'Splitting Third-Party Libraries' Privileges from Android Apps', in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), vol. 10343 LNCS, Springer Verlag, 2017, pp. 80–94. |
| P37 | M. Li et al., 'LibD: Scalable and Precise Third-Party Library Detection in Android Markets', in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), May 2017, pp. 335–346, doi: 10.1109/ICSE.2017.38. |
| P38 | J. Fu, Y. Zhou, H. Liu, Y. Kang, and X. Wang, 'Perman: Fine-Grained Permission Management for Android Applications', in 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE), Oct. 2017, vol. 2017-Octob, pp. 250–259, doi: 10.1109/ISSRE.2017.38. |

| P39 | J. Vronsky, R. Stevens, and H. Chen, ‘SurgeScan: Enforcing security policies on untrusted third-party Android libraries’, in 2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI), Aug. 2017, pp. 1–8, doi: 10.1109/UIC-ATC.2017.8397610. |
|---|---|
| P40 | D. Titze, M. Lux, and J. Schuette, ‘Ordol: Obfuscation-Resilient Detection of Libraries in Android Applications’, in 2017 IEEE Trustcom/BigDataSE/ICESS, Aug. 2017, pp. 618–625, doi: 10.1109/Trustcom/BigDataSE/ICESS.2017.292. |
| P41 | Y. Zhang et al., ‘Detecting third-party libraries in Android applications with high precision and recall’, in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2018, vol. 2018-March, pp. 141–152, doi: 10.1109/SANER.2018.8330204. |
| P42 | Y. Wang, H. Wu, H. Zhang, and A. Rountev, ‘ORLIS: Obfuscation-Resilient Library Detection for Android’, in Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, New York, NY, USA, 2018, pp. 13–23, doi: 10.1145/3197231.3197248. |
| P43 | X. Zhu, J. Li, Y. Zhou, and J. Ma, ‘AdCapsule: Practical Confinement of Advertisements in Android Applications’, IEEE Transactions on Dependable and Secure Computing, vol. 17, no. 3, pp. 1–1, 2018, doi: 10.1109/TDSC.2018.2814999. |
| P44 | Z. Tang et al., ‘Securing android applications via edge assistant third-party library detection’, Computers and Security, vol. 80, pp. 257–272, Jan. 2019, doi: 10.1016/j.cose.2018.07.024. |
| P45 | B. Li, Y. Zhang, J. Li, R. Feng, and D. Gu, ‘APPCOMMUNE: Automated Third-Party Libraries De-duplicating and Updating for Android Apps’, in 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb. 2019, pp. 344–354, doi: 10.1109/SANER.2019.8668009. |
| P46 | M. Diamantaris, E. P. Papadopoulos, E. P. Markatos, S. Ioannidis, and J. Polakis, ‘Reaper: Real-time app analysis for augmenting the android permission system’, in CODASPY 2019 - Proceedings of the 9th ACM Conference on Data and Application Security and Privacy, New York, NY, USA, 2019, pp. 37–48, doi: 10.1145/3292006.3300027. |
| P47 | Y. He, X. Yang, B. Hu, and W. Wang, ‘Dynamic privacy leakage analysis of Android third-party libraries’, Journal of Information Security and Applications, vol. 46, pp. 259–270, 2019, doi: 10.1016/j.jisa.2019.03.014. |
| P48 | J. Feichtner and C. Rabensteiner, ‘Obfuscation-resilient code recognition in android apps’, New York, NY, USA, 2019, doi: 10.1145/3339252.3339260. |
| P49 | A. K. Mondal, C. Roy, B. Roy, and K. A. Schneider, ‘Automatic Components Separation of Obfuscated Android Applications: An Empirical Study of Design Based Features’, in 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), Nov. 2019, pp. 23–28, doi: 10.1109/ASEW.2019.00022. |

| | |
|---|---|
| P50 | C. Chen, Z. Xing, and Y. Liu, 'What's Spain's Paris? Mining analogical libraries from Q&A discussions', Empirical Software Engineering, vol. 24, no. 3, pp. 1155–1194, Jun. 2019, doi: 10.1007/s10664-018-9657-y. |
| P51 | Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang, 'Diversified Third-party Library Prediction for Mobile App Development', IEEE Transactions on Software Engineering, pp. 1–1, 2020, doi: 10.1109/TSE.2020.2982154. |


## B.2. Android Energy Bugs Covered by Tools in the 'Detector' and 'Optimizer' Categories

Table B.1: Android energy bugs detected by each tool in 'Detector' and 'Optimizer' categories.

| Tool | RL | WB | VBS | IB | TMV | TDL | NCD | UL | UP | OLP | VHB | EMC | Ref |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wu et al. | ✓ | | | | | | | | | | | | P1 |
| GreenDroid | ✓ | ✓ | | | | | | | | | | | P17 |
| Kim et al. | | | | | | | ✓ | | | | | | P3 |
| Statedroid | ✓ | ✓ | ✓ | | | | | | | | | | P5 |
| PatBugs | | | | | | | | | | | | | P8 |
| SAAD | ✓ | | | | ✓ | ✓ | ✓ | ✓ | ✓ | | | | P9 |
| Paprika | ✓ | | | | | | ✓ | | | | | | P7 |
| aDoctor | ✓ | ✓ | | | | | | | | | | | P4 |
| DelayDroid | | | ✓ | | | | | | | | | | P2 |
| HOT-PEPPER | ✓ | | | | | | ✓ | | | | | | P10 |
| Asyncdroid | | | ✓ | | | | | | | | | | P11 |
| EARMO | | | | | | | | | | | | | P15 |
| EnergyPatch | ✓ | ✓ | ✓ | ✓ | | | | | | | | | P18 |
| Nguyen et al. | | | | | | | | | | | | | P21 |
| Chimera | ✓ | ✓ | | | | | | | | ✓ | ✓ | ✓ | P22 |
| ServDroid | ✓ | | ✓ | | | | | | | | | | P23 |
| Leafactor | | ✓ | | | | | | | | ✓ | | | P24 |

1. The '**RL**' (Resource Leak) energy bug is caused by energy leaks in resources such as camera, multimedia, and memory, that are not released when no longer needed.
2. The '**WB**' (Wake-lock Bug) energy bug is caused by excessive energy consumption because the device is kept in high power state for too long.
3. The '**VBS**' (Vacuous Background Services) energy bug is caused by services consuming resources in the background after the app exits.
4. The '**IB**' (Immortality Bug) energy bug is caused by apps that re-spawn after they have been closed.

5. The **TMV**’ (Too Many Views) energy bug is caused by exceeding the default number of widgets in view hierarchy.
6. The **TDL**’ (Too Deep Layout) energy bug is caused by exceeding the default nesting depth in view hierarchy.
7. The **NCD’** (Not Using Compound Drawables) energy bug is caused by not replacing multiple widgets.
8. The ‘**UL**’ (Useless Leaf) energy bug is caused by not removing a widget that does not impact the interface.
9. The ‘**UP**’ (Useless Parent) energy bug is caused by not removing widget which is not root view and has no background properties.
10. The ‘**OLP**’ (Obsolete Layout) Parameter energy bug is caused by XML layout parameters that do not contribute to the layout functionality.
11. The ‘**VHB**’ (View Holder Bug) energy bug is caused by using getView() in View Holder repeatedly instead of caching in a variable.
12. The ‘**EMC**’ (Excessive Method Calls) energy bug is caused by excessive calls of a method whose implementation can be extracted to the caller class.

## B.3. Android Code Smells Covered by Tools in ‘Detector’ and ‘Optimizer’ categories

1. The ‘**DTWC**’ (Data Transmission Without Compression) code smell is caused due to a file sent over the network without compression.
2. The ‘**DR**’ (Debuggable Release) code smell which when the Debuggable attribute is left set to true in an app.
3. The ‘**DW**’ (Durable Wake-lock) code smell arises when a wake-lock is acquired but not released.
4. The ‘**IDFP**’ (Inefficient Data Format and Parser) code smell arises when XML or JSON data is parsed with TreeParser instead of a more efficient parser like StreamParser.
5. The ‘**IDS**’ (Inefficient Data Structure) code smell arises when a data structure like Hash Map of objects can be replaced by a Sparse array.
6. The ‘**ISQLQ’** (Inefficient SQL Query) code smell arises when a SQL query issued to retrieve data from a remote server instead of using a JSON based query.
7. The ‘**IGS**’ (Internal Getter and Setter) code smell arises when getter and setter methods are used to access fields of a class externally.

8. The ‘**LIC**’ (Leaking Inner Class) code smell arises when an instance of an outer class is held by a non-static inner class.
9. The ‘**LT**’ (Leaking Thread) code smell arises when a thread is never stopped.
10. The ‘**MIM**’ (Member Ignoring Method) code smell arises when non static methods are being used to access class properties.
11. The ‘**NLMR**’ (No Low Memory Resolver) code smell which arises when the method onLowMemory() is not implemented by an activity to clear cache.
12. The ‘**PD**’ (Public Data) code smell arises when data private to an app is kept in a public data store that can be accessed by other apps.
13. The ‘**RAM**’ (Rigid Alarm Manager) code smell arises when Alarm Manager is triggered frequently without bundling the updates.
14. The ‘**SL**’ (Slow Loop) code smell arises when a slow version of loop is used which can be converted to a more efficient implementation. The ‘UC’ (Unclosed Closable) code smell arises when a resource is not closed after its usage,
15. The ‘**LC**’ (Lifetime Containment) code smell arises when a listener is registered but not unregistered as the activity is destroyed.
16. The ‘**LWS**’ (Long Wait State) code smell arises when a listener is added leading to a potentially long wait state for resources leading to app suspension.
17. The ‘**UHA**’ (Unsupported Hardware Acceleration) code smell arises when methods are called without hardware acceleration hence they use CPU instead of GPU.
18. The ‘**BFU**’ (Bitmap Format Usage) code smell arises when images are represented as Bitmaps which is memory intensive.
19. The ‘**UIO**’ (UI Overdraw) code smell arises due to overdrawing of UI components.
20. The ‘**IWR**’ (Invalidate Without Rect) code smell arises when exact rect to be redrawn is not mentioned in onDraw() method resulting in redrawing everything.
21. The ‘**HAT**’ (Heavy AsyncTask) code smell arises when heavy resource intensive operations are done inside the Async Task.
22. The ‘**HSS**’ (Heavy Service Start) code smell arises when heavy resource intensive operations are done inside the Service on main thread.

23. The ‘**HBR**’ (Heavy Broadcast Receiver) code smell arises when heavy resource intensive operations are done inside the Broadcast Receiver on main thread. The ‘IOD’ (Init ONDraw) code smell arises when new objects are created inside the onDraw() method which is called a large number of times.

24. The ‘**ERB**’ (Early Resource Binding) code smell arises when high energy resources are initialized before they need to be used.

25. The ‘**VHP**’ (View Holder Pattern) code smell arises when lists are loaded using the View Holder pattern that reused views.

Table B.2: Android code smells detected by each tool in the ‘Detector’ and ‘Optimizer’ categories.

| Tool | Ref | DTWC | DR | IDFP | IDS | ISQLQ | IGS | LIC | LT | MIM | NLMR | PD | RAM | SL | UC | LC | LWS | UHA | BFU | UIO | IWR | HAT | HSS | HBR | IOD | ERB | VHP |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Wu et al. | P1 | | | | | | | | | | | | | | ✓ | ✓ | ✓ | | | | | | | | | | |
| Kim et al. | P3 | | | | | | | | | | | | | | | | | | | | | | | | ✓ | | |
| Statedroid | P5 | | | | | | | | | | | | | | ✓ | ✓ | | | | | | | | | | | |
| PatBugs | P8 | | | | | | | | | | | | | | ✓ | | | | | | | | | | | | |
| SAAD | P9 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| ADoctor | P4 | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | | | | | | | |
| Paprika | P7 | | | | ✓ | | ✓ | ✓ | | ✓ | ✓ | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| GreenDroid | P17 | | | | | | | | | | | | | | ✓ | | | | | | | | | | | | |
| DelayDroid | P2 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| HOT PEPPER | P10 | | | | ✓ | | ✓ | ✓ | | ✓ | ✓ | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | |
| AsyncDroid | P11 | | | | | | | | | | | | | | | | | | | | | ✓ | | | | | |
| EARMO | P15 | | | | ✓ | | ✓ | | | | | | | | | | | | | | | | | | | ✓ | |
| EnergyPatch | P18 | | | | | | | | | | | | | | ✓ | | | | | | | | | | | | |
| Nguyen et al. | P21 | | | | | | | | ✓ | | | | | ✓ | | | | | | | | | | | | | |
| Chimera | P22 | | | | ✓ | | | | | ✓ | | | | | ✓ | | | | | | | | | | ✓ | | |
| ServDroid | P23 | | | | | | | | | | | | | | | | | | | | | | | | | ✓ | |
| Leafactor | P24 | | | | | | | | | | | | | | ✓ | | | | | | | | | | ✓ | | ✓ |

# Appendix C. INSTRUCTIONS FOR INSTALLING ARENA

## C.1. Installation Pre-requisites

1. The host system must have JAVA version 1.8 or above
2. Install Python 3.0 or above from here and add to the path of system environment variables.
3. Install Monsoon Power Monitor, and it's related Python libraries as per the user manual of Monsoon Power Monitor.
4. Install R from here with version 3.4.3. or above. Path to ./bin, ./bin/R.exe, and ./bin/Rscript.exe must be added to the path in System Environment Variables.
5. Install Rtools from here. Path to ./usr/bin must be added to the Path of System Environment Variables
6. Make sure no other instance of R is running.
7. The plugin will install the following R libraries automatically. However, in case of any un foreseen errors ARENA user can also install them manually using R : dplyr, ggpubr, RColorBrewer, ggplot, officer, flextable, propagate, pgirmess, RVAideMemoire, pastecs.
8. The mobile device must be a rooted device.
9. If test APK is selected, make sure it has a TestClass that runs all the instrumented tests.
10. The app under test must have API version >= 19

## C.2. Plugin Installation

1. Open IntelliJ Idea. Go to File > Settings > Plugin.
2. Click on the Settings icon and click Install Plugin From Disk
3. Choose the EnergyPlugin-1.0-SNAPSHOT.zip file (you can download this file from the bitbucket reposiotry https://bitbucket.org/hinaanwar2003/arena/src/master/)
4. Click on OK. Restart IDE to start using the plugin.

# Appendix D. INSTALLATION INSTRUCTION AND DETAILED DATA - REHAB

## D.1. Plugin Installation

1. Open IntelliJ Idea. Go to File > Settings > Plugin.
2. Click on the Settings icon and click Install Plugin From Disk
3. Choose the Rehab-1.0-SNAPSHOT.zip file (you can download this file from the bitbucket reposiotry https://bitbucket.org/hinaanwar2003/rehab/src/master/ )
4. Click on OK. Restart IDE to start using the plugin.

## D.2. Third-party Libraries and Version Used in Android Apps

The following tables show number of apps using versions of third-party serialization/deserialization libraries in a sample of Android apps taken from Google playstore and F-Droid repository in which selected third-party HTTP libraries are detected. For each serialization/deserialization library, versions are divided into groups based on type of change.

Table D.1: Group of versions created for Gson library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 1.1.0 | 0 | 0% | 0 | 0% | 1 | no change |
| 1.4.0 | 0 | 0% | 1 | 0% | 2 | MAJOR |
| 1.5.0 | 0 | 0% | 0 | 0% | 3 | MAJOR |
| 1.6.0 | 3 | 0% | 1 | 0% | 4 | MAJOR |
| 1.7.0 | | | | | | MAJOR |
| 1.7.1 | 8 | 1% | 8 | 3% | 5 | MINOR |
| 1.7.2 | | | | | | PATCH |
| 2.0.0 | 1 | 0% | 1 | 0% | 6 | MAJOR |
| 2.1.0 | | | | | | MAJOR |
| 2.2.0 | | | | | | MINOR |
| 2.2.1 | | | | | | PATCH |
| 2.2.2 | | | | | | PATCH |
| 2.2.3 | | | | | | MINOR |
| 2.2.4 | 288 | 42% | 92 | 37% | 7 | PATCH |
| 2.3.0 | | | | | | MINOR |
| 2.3.1 | | | | | | PATCH |
| 2.4.0 | | | | | | MINOR |
| 2.5.0 | | | | | | PATCH |
| 2.6.0 | | | | | | MINOR |
| 2.6.1 | | | | | | PATCH |
| 2.6.2 | | | | | | PATCH |
| 2.7.0 | | | | | | MAJOR |
| 2.8.0 | | | | | | MINOR |
| 2.8.1 | 379 | 56% | 145 | 58% | 8 | MINOR |

| 2.8.2 | | | | | | MINOR |
|---|---|---|---|---|---|---|
| 2.8.3 | | | | | | MINOR |
| 2.8.4 | | | | | | PATCH |
| 2.8.5 | | | | | | MINOR |
| 2.8.6 | | | | | | MINOR |
| **TOTAL** | **679** | | **248** | | | |

Table D.2: Group of versions created for Jackson library along with number of apps in which these groups were detected

| Version | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 2.2.2 | | | | | | no change |
| 2.2.3 | 15 | 2% | 2 | 1% | 1 | MINOR |
| 2.2.4 | | | | | | MINOR |
| 2.3.0 | | | | | | MAJOR |
| 2.3.1 | | | | | | PATCH |
| 2.3.2 | 16 | 2% | 0 | 0% | 2 | MINOR |
| 2.3.3 | | | | | | PATCH |
| 2.3.4 | | | | | | PATCH |
| 2.3.5 | | | | | | PATCH |
| 2.4.0 | | | | | | MAJOR |
| 2.4.1 | | | | | | PATCH |
| 2.4.2 | | | | | | PATCH |
| 2.4.3 | | | | | | PATCH |
| 2.4.4 | | | | | | PATCH |
| 2.4.5 | | | | | | PATCH |
| 2.4.6 | | | | | | PATCH |
| 2.5.0 | | | | | | MINOR |
| 2.5.1 | | | | | | MINOR |
| 2.5.2 | | | | | | MINOR |
| 2.5.3 | 14 | 2% | 41 | 10% | 3 | PATCH |
| 2.5.4 | | | | | | PATCH |
| 2.5.5 | | | | | | PATCH |
| 2.6.0 | | | | | | MINOR |
| 2.6.1 | | | | | | PATCH |
| 2.6.2 | | | | | | PATCH |
| 2.6.3 | | | | | | PATCH |
| 2.6.4 | | | | | | PATCH |
| 2.6.5 | | | | | | PATCH |
| 2.6.6 | | | | | | MINOR |
| 2.6.7 | | | | | | PATCH |
| 2.7.0 | | | | | | MAJOR |
| 2.7.1 | | | | | | MINOR |
| 2.7.2 | | | | | | MINOR |
| 2.7.3 | | | | | | PATCH |
| 2.7.4 | | | | | | PATCH |
| 2.7.5 | | | | | | MINOR |
| 2.7.6 | | | | | | PATCH |
| 2.7.7 | | | | | | PATCH |
| 2.7.8 | | | | | | PATCH |
| 2.7.9 | | | | | | PATCH |
| 2.8.0 | | | | | | MINOR |
| 2.8.1 | | | | | | PATCH |
| 2.8.2 | | | | | | PATCH |
| 2.8.3 | | | | | | PATCH |
| 2.8.4 | | | | | | PATCH |
| 2.8.5 | | | | | | PATCH |
| 2.8.6 | 32 | 4% | 105 | 27% | 4 | PATCH |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2.8.7 | | | | | | MINOR |
| 2.8.8 | | | | | | MINOR |
| 2.8.9 | | | | | | PATCH |
| 2.8.10 | | | | | | PATCH |
| 2.8.11 | | | | | | PATCH |
| 2.9.0 | | | | | | MINOR |
| 2.9.1 | | | | | | PATCH |
| 2.9.2 | | | | | | MINOR |
| 2.9.3 | | | | | | PATCH |
| 2.9.4 | | | | | | MINOR |
| 2.9.5 | | | | | | PATCH |
| 2.9.6 | | | | | | MINOR |
| 2.9.7 | | | | | | PATCH |
| 2.9.8 | | | | | | MINOR |
| 2.9.9 | | | | | | PATCH |
| 2.9.10 | | | | | | PATCH |
| 2.10.0 | | | | | | MINOR |
| 2.10.1 | | | | | | PATCH |
| 2.10.2 | | | | | | MINOR |
| 2.10.3 | | | | | | PATCH |
| 2.10.4 | | | | | | MINOR |
| 2.10.5 | | | | | | PATCH |
| 2.11.0 | | | | | | MINOR |
| 2.11.1 | | | | | | PATCH |
| 2.11.2 | | | | | | PATCH |
| 2.11.3 | | | | | | PATCH |
| 2.11.4 | | | | | | PATCH |
| 2.12.0 | 0 | 0% | 0 | 0% | 5 | MAJOR |
| **TOTAL** | **77** | | **148** | | | |

Table D.3: Group of versions created for Moshi library along with number of apps in which these groups were detected

| Versions | # of Google playstore apps | % | # of F-Droid apps | % | Group | Type of Change |
|---|---|---|---|---|---|---|
| 0.9.0 | 0 | 0% | 0 | 0% | 1 | no change |
| 1.0.0 | 1 | 20% | 1 | 3% | 2 | MAJOR |
| 1.1.0 | | | | | | MINOR |
| 1.2.0 | | | | | | MINOR |
| 1.3.0 | | | | | | MINOR |
| 1.3.1 | | | | | | PATCH |
| 1.4.0 | 0 | 0% | 2 | 5% | 3 | MAJOR |
| 1.5.0 | 0 | 0% | 1 | 3% | 4 | MAJOR |
| 1.6.0 | 4 | 80% | 34 | 89% | 5 | MAJOR |
| 1.7.0 | | | | | | MINOR |
| 1.8.0 | | | | | | MINOR |
| 1.9.0 | | | | | | MINOR |
| 1.9.1 | | | | | | PATCH |
| 1.9.2 | | | | | | PATCH |
| 1.9.3 | | | | | | PATCH |
| 1.10.0 | | | | | | MINOR |
| 1.11.0 | | | | | | MINOR |
| **TOTAL** | **5** | | **38** | | | |

# ACKNOWLEDGEMENT

First and foremost, I thank Almighty for giving me the opportunity and ability to pursue this research.

I am extremely grateful to my supervisor Dietmar Pfahl for the guidance, support and feedback throughout these years. I am thankful for the constant opportunities for professional development he gave me, but also for his time, help and discussions outside my PhD topic.

I am also grateful to my co-supervisor Satish Srirama, for his support and feedback.

I am grateful to the reviewers of my thesis for their comments and valuable feedback that have significantly improved my thesis.

I want to thank all my friends and colleagues at the University of Tartu for their support. I want to thank Kristiina and Karoliine for helping me with the Estonian translations in this thesis.

I want to thank my family for their unconditional love and support. I say thanks to my daughter, Rameen, for her hugs and laughs. She was always around and accompanied me in moments of hardship and frustration. She is my strength and kept me in high spirits. I also want to thank M.B. Aslam for helping me to move to Estonia.

Finally, I would like to acknowledge the Estonian Research Council, the Doctoral School of Information and Communication Technology (IKTDK), and the Estonian Centre of Excellence in ICT research (EXCITE) for funding my research. I am also thankful to Professor Marlon Dumas, for funding my research beyond the nominal time.

# SISUKOKKUVÕTE

Praegusel üldlevinud tehnoloogia kasutamise ajastul kasutatakse laialdaselt selliseid seadmeid nagu nutitelefone, tahvelarvuteid ja sülearvuteid. Kuna kõik need seadmed töötavad akudega, on energiatõhususe küsimus muutunud üheks oluliseks parameetriks, mille põhjal kasutajad seadmeid valivad. Energiatõhususe eesmärk on vähendada toodete ja teenuste pakkumisel vajaliku energia hulka. Digitaalsse seadme energiatõhususest on saanud osa selle üldisest tajutavast kvaliteedist.

Empiirilised uuringud on näidanud, et mobiilirakendused, mis akut säästavad, saavad kasutajatelt tavaliselt häid hinnanguid. Mobiilirakenduste energiasäästlikumaks muutmiseks on avaldatud palju uuringuid, mis tutvustavad koodi refaktoreerimise juhiseid ja tööriistu. Neid juhiseid ei saa aga energiatõhususe suhtes üldistada, sest iga konteksti jaoks pole piisavalt energiaga seotud andmeid. Olemasolevad energiatõhususe tööriistad ja profiilid on enamasti prototüübid, mida saab kasutada ainult väikese energiaga seotud probleemide alamhulga jaoks. Lisaks keskenduvad olemasolevad juhised ja tööriistad energiaprobleemide lahendamisele  textit tagantjärele, st kui need on juba koodi sisse viidud.

Androidi rakenduse koodi võib laias laastus jagada kaheks osaks: kirjutatud kood ja korduvkasutatav kood. Kirjutatud kood on iga rakenduse jaoks ainulaadne. Korduvkasutatav kood sisaldab kolmandate osapoolte teeke, mis on lisatud rakendustesse arendusprotsessi kiirendamiseks. Võrreldes arvuti- või veebirakendustega sisaldavad Android-rakendused mitut komponenti, millel on kasutaja juhitud töövood. Tüüpiline Androidi rakendus koosneb tegevustest, fragmentidest, teenustest, sisupakkujatest ja ülekannete vastuvõtjatest. Arhitektuuri erinevuse tõttu ei ole traditsiooniliste Java-põhiste rakenduste arendamisel kasutatavad tugivahendid Androidi rakenduste arendamisel ja hooldamisel nii kasulikud.

Alustuseks hindame erinevate Androidi rakenduste koodilõhnade refaktoreerimiste energiatarbimist. Uurisime viie Androidi koodilõhna tüübi refaktoreerimise energiamõju (kõigepealt individuaalselt tüübi ja seejärel permutatsioonide alusel) avatud lähtekoodiga Androidi omarakendustes. Uurisime ka koodilõhnade refaktoreerimise mõju Androidi avatud lähtekoodiga omarakenduste täitmisajale. Seejärel viime läbi empiirilise uuringu Androidi rakendustes kasutatavate kolmandate osapoolte teekide energiamõju kohta. Kolmandate osapoolte teegid valitakse kasutusstatistika ja populaarsete kolmandate osapoolte turgude ja kataloogide paremusjärjestuse alusel. Tuvastasime nende teekide tüüpilised kasutusjuhud Google Play poe kõige enam allalaaditud rakendustes. Valitud kolmandate osapoolte teeke on võimalik iga kasutusjuhu puhul kasutada üksteise alternatiividena. Meie tulemused näitavad, et kolmandate osapoolte võrguteekide energiatarbimine erineb oluliselt. Arutame tulemusi ja pakume üldisi kontekstipõhiseid juhiseid, mida

saaks kasutada rakenduste väljatöötamisel. Energiatarbimise mõõtmiseks kasutame riistvarapõhiseid energiamõõtmisi, kuna need on täpsemad. Energiaandmete statistilise analüüsiga hinnatakse kasutatud energia dispersiooni, mõju suurust ja protsentuaalset muutust.

Lisaks teeme süstemaatilise kirjanduse ülevaate, et teha kindlaks ja uurida praeguseid kaasaegseid tugivahendeid, mis on saadaval Androidi roheliseks arendamiseks. Teeme kokkuvõtte nende tööriistade ulatusest ja piirangutest ning tõstame esile lüngad uuringutes. Selle uuringu ja varem läbi viidud katsete põhjal toome esile riistvarapõhiste energiamõõtmiste jäädvustamise ja taasesitamise probleemid. Riistvarapõhise energia mõõtmise protsess on pikk ja järsu õppimiskõveraga. Seetõttu töötame Android Studio IDE jaoks välja abivahendi  textbf ARENA (**A**nalysing ene**R**gy **E**fficiency in a**N**droid **A**pps, mis võiks aidata koguda energiaandmeid ja analüüsida Androidi rakenduste energiatarbimist. See tööriist ühendab kõik tegevused, mis on vajalikud Androidi rakenduste energiatarbimise mõõtmiseks, statistiliseks analüüsimiseks ja aruandluseks (sh tulemuste tõlgendamine ja visualiseerimine graafikute kujul). ARENA on avatud lähtekoodiga ja selle rakendus on saadaval aadressil https://bitbucket.org/hinaanwar2003/arena

Lõpuks töötame Android Studio IDE jaoks välja abivahendi  textbf REHAB (**R**ecommending **E**nergy-efficient t**H**ird-p**A**rty li**B**raries- energiatõhusate kolmandate osapoolete teekide soovitamine), et soovitada arendajatele energiatõhusaid kolmandate osapoolete võrguteeke. Samuti arutleme selle üle, kuidas REHABi ulatust saab laiendada, viies läbi 8457 Androidi rakenduse kasutamise ja muudatuste analüüsi. Kasutusanalüüs kvantifitseerib valitud kolmandate osapoolte HTTP-teekide ja kõigi nende versioonide kasutamise reaalsetes Androidrakendustes. Muudatuste analüüs kvantifitseerib, milliseid muudatusi järjestikustes versioonides tehakse. REHAB on avatud lähtekoodiga ja selle rakendus on saadaval aadressil https://bitbucket.org/hinaanwar2003/rehab/src/master/.

# CURRICULUM VITAE

## Personal data


| | |
|---|---|
| Name: | Hina Anwar |
| Date of Birth: | 11.10.1986 |
| Nationality: | Pakistani |
| Language skills: | Urdu (native), English |
| Email: | hina.anwar@ut.ee |


## Education

| | |
|---|---|
| 2016-2021 | University of Tartu, Tartu, Estonia, Ph.D. in Computer Science |
| 2010-2013 | NUST College of E&ME, Rawalpindi, Pakistan, M.Sc in Software Engineering |
| 2004-2008 | Fatima Jinnah Women University, Rawalpindi, Pakistan, B.SC in Software Engineering |

## Employment

| | |
|---|---|
| 2017-2021 | Junior Research Fellow, University of Tartu, Estonia |
| 2013-2016 | Lecturer, University of Education, Pakistan (on study leave) |
| 2012-2012 | Lab Engineer, Air University, Pakistan |
| 2011-2011 | Research Assistant, NUST College of E&ME, Pakistan |

## Scientific work

Main fields of interest:

- Software Engineering
- Programming Pattern
- Data Mining and Machine Learning

# ELULOOKIRJELDUS

## Isiklikud andmed

| | |
|---|---|
| Nimi: | Hina Anwar |
| Sünniaeg: | 11.10.1986 |
| Kodakondsus: | Pakistan |
| Keeleoskus: | urdu (emakeel), inglise keel |
| E-post: | hina.anwar@ut.ee |

## Haridus

| | |
|---|---|
| 2016-2021 | Tartu Ülikool, Tartu, Eesti, Informaatike Ph.D. |
| 2010-2013 | NUST College of E&ME, Rawalpindi, Pakistan, Tarkvaratehnika MSc |
| 2004-2008 | Fatima Jinnah Women University, Rawalpindi, Pakistan, Tarkvaratehnika B.Sc |

## Tööhõive

| | |
|---|---|
| 2017-2021 | Nooremteadur, Tartu Ülikool, Eesti |
| 2013-2016 | Lektor, University of Education, Pakistan (õppepuhkusel) |
| 2012-2012 | Laboriinsener, Air University, Pakistan |
| 2011-2011 | Uurimisassistent, NUST College of E&ME, Pakistan |

## Teaduslik töö

Peamised huvialad:

- Tarkvaraarendus
- Programmeerimismuster
- Andmete kaevandamine ja masinõpe

# LIST OF ORIGINAL PUBLICATIONS

## Publications in the scope of the thesis


I **Anwar, Hina** and Dietmar Pfahl. Towards greener software engineering using software analytics: A systematic mapping. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 157–166. IEEE, 2017

II **Anwar, Hina**, Dietmar Pfahl, and Satish Narayana Srirama. An investigation into the energy consumption of http post request methods for android app development. In *13th International Conference on Software Technologies (ICSOFT 2018)*, pages 241–248, 2018

III **Anwar, Hina**, Dietmar Pfahl, and Satish N Srirama. Evaluating the impact of code smell refactoring on the energy consumption of android applications. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 82–86. IEEE, 2019

IV Iffat Fatima, **Anwar, Hina**, Dietmar Pfahl, and Usman Qamar. Tool support for green android development: A systematic mapping study. In *Proceedings of the 15th International Conference on Software Technologies (ICSOFT)*, number 1, pages 409–417, 2020

V **Anwar, Hina**. Towards greener android application development. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 170–173. IEEE, 2020

VI **Anwar, Hina**, Berker Demirer, Dietmar Pfahl, and Satish Srirama. Should energy consumption influence the choice of android third-party http libraries? In *MOBILESoft '20: IEEE/ACM 7th International Conference on Mobile Software Engineering and Systems*, pages 87–97, 2020

VII **Anwar, Hina**, Iffat Fatima, Dietmar Pfahl, and Usman Qamar. *Tool Support for Green Android Development*, pages 153–182. Springer International Publishing, 2021


## Publications out of the scope of the thesis


I Fauzia Khan, **Anwar, Hina**, Dietmar Pfahl, and Satish Srirama. Software techniques for making cloud data centers energy-efficient: A systematic mapping study. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2020)*, 2020

II Iffat Fatima, **Anwar, Hina**, Dietmar Pfahl, and Usman Qamar. Detection

and correction of android-specific code smells and energy bugs: An android lint extension. In *8th International Workshop on Quantitative Approaches to Software Quality (QuASoQ 2020)*, 2020

# DISSERTATIONES INFORMATICAE PREVIOUSLY PUBLISHED IN DISSERTATIONES MATHEMATICAE UNIVERSITATIS TARTUENSIS

19. **Helger Lipmaa.** Secure and efficient time-stamping systems. Tartu, 1999, 56 p.
22. **Kaili Müürisep.** Eesti keele arvutigrammatika: süntaks. Tartu, 2000, 107 lk.
23. **Varmo Vene.** Categorical programming with inductive and coinductive types. Tartu, 2000, 116 p.
24. **Olga Sokratova.** Ω-rings, their flat and projective acts with some applications. Tartu, 2000, 120 p.
27. **Tiina Puolakainen.** Eesti keele arvutigrammatika: morfoloogiline ühestamine. Tartu, 2001, 138 lk.
29. **Jan Villemson.** Size-efficient interval time stamps. Tartu, 2002, 82 p.
45. **Kristo Heero.** Path planning and learning strategies for mobile robots in dynamic partially unknown environments. Tartu 2006, 123 p.
49. **Härmel Nestra.** Iteratively defined transfinite trace semantics and program slicing with respect to them. Tartu 2006, 116 p.
53. **Marina Issakova.** Solving of linear equations, linear inequalities and systems of linear equations in interactive learning environment. Tartu 2007, 170 p.
55. **Kaarel Kaljurand.** Attempto controlled English as a Semantic Web language. Tartu 2007, 162 p.
56. **Mart Anton.** Mechanical modeling of IPMC actuators at large deformations. Tartu 2008, 123 p.
59. **Reimo Palm.** Numerical Comparison of Regularization Algorithms for Solving Ill-Posed Problems. Tartu 2010, 105 p.
61. **Jüri Reimand.** Functional analysis of gene lists, networks and regulatory systems. Tartu 2010, 153 p.
62. **Ahti Peder.** Superpositional Graphs and Finding the Description of Structure by Counting Method. Tartu 2010, 87 p.
64. **Vesal Vojdani.** Static Data Race Analysis of Heap-Manipulating C Programs. Tartu 2010, 137 p.
66. **Mark Fišel.** Optimizing Statistical Machine Translation via Input Modification. Tartu 2011, 104 p.
67. **Margus Niitsoo**. Black-box Oracle Separation Techniques with Applications in Time-stamping. Tartu 2011, 174 p.
71. **Siim Karus.** Maintainability of XML Transformations. Tartu 2011, 142 p.
72. **Margus Treumuth.** A Framework for Asynchronous Dialogue Systems: Concepts, Issues and Design Aspects. Tartu 2011, 95 p.
73. **Dmitri Lepp.** Solving simplification problems in the domain of exponents, monomials and polynomials in interactive learning environment T-algebra. Tartu 2011, 202 p.

74. **Meelis Kull.** Statistical enrichment analysis in algorithms for studying gene regulation. Tartu 2011, 151 p.
77. **Bingsheng Zhang.** Efficient cryptographic protocols for secure and private remote databases. Tartu 2011, 206 p.
78. **Reina Uba.** Merging business process models. Tartu 2011, 166 p.
79. **Uuno Puus.** Structural performance as a success factor in software development projects – Estonian experience. Tartu 2012, 106 p.
81. **Georg Singer.** Web search engines and complex information needs. Tartu 2012, 218 p.
83. **Dan Bogdanov.** Sharemind: programmable secure computations with practical applications. Tartu 2013, 191 p.
84. **Jevgeni Kabanov.** Towards a more productive Java EE ecosystem. Tartu 2013, 151 p.
87. **Margus Freudenthal.** Simpl: A toolkit for Domain-Specific Language development in enterprise information systems. Tartu, 2013, 151 p.
90. **Raivo Kolde.** Methods for re-using public gene expression data. Tartu, 2014, 121 p.
91. **Vladimir Šor.** Statistical Approach for Memory Leak Detection in Java Applications. Tartu, 2014, 155 p.
92. **Naved Ahmed.** Deriving Security Requirements from Business Process Models. Tartu, 2014, 171 p.
94. **Liina Kamm.** Privacy-preserving statistical analysis using secure multi-party computation. Tartu, 2015, 201 p.
100. **Abel Armas Cervantes.** Diagnosing Behavioral Differences between Business Process Models. Tartu, 2015, 193 p.
101. **Fredrik Milani.** On Sub-Processes, Process Variation and their Interplay: An Integrated Divide-and-Conquer Method for Modeling Business Processes with Variation. Tartu, 2015, 164 p.
102. **Huber Raul Flores Macario.** Service-Oriented and Evidence-aware Mobile Cloud Computing. Tartu, 2015, 163 p.
103. **Tauno Metsalu.** Statistical analysis of multivariate data in bioinformatics. Tartu, 2016, 197 p.
104. **Riivo Talviste.** Applying Secure Multi-party Computation in Practice. Tartu, 2016, 144 p.
108. **Siim Orasmaa.** Explorations of the Problem of Broad-coverage and General Domain Event Analysis: The Estonian Experience. Tartu, 2016, 186 p.
109. **Prastudy Mungkas Fauzi.** Efficient Non-interactive Zero-knowledge Protocols in the CRS Model. Tartu, 2017, 193 p.
110. **Pelle Jakovits.** Adapting Scientific Computing Algorithms to Distributed Computing Frameworks. Tartu, 2017, 168 p.
111. **Anna Leontjeva.** Using Generative Models to Combine Static and Sequential Features for Classification. Tartu, 2017, 167 p.
112. **Mozhgan Pourmoradnasseri.** Some Problems Related to Extensions of Polytopes. Tartu, 2017, 168 p.

113. **Jaak Randmets.** Programming Languages for Secure Multi-party Computation Application Development. Tartu, 2017, 172 p.
114. **Alisa Pankova.** Efficient Multiparty Computation Secure against Covert and Active Adversaries. Tartu, 2017, 316 p.
116. **Toomas Saarsen.** On the Structure and Use of Process Models and Their Interplay. Tartu, 2017, 123 p.
121. **Kristjan Korjus.** Analyzing EEG Data and Improving Data Partitioning for Machine Learning Algorithms. Tartu, 2017, 106 p.
122. **Eno Tõnisson.** Differences between Expected Answers and the Answers Offered by Computer Algebra Systems to School Mathematics Equations. Tartu, 2017, 195 p.

## DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

1. **Abdullah Makkeh**. Applications of Optimization in Some Complex Systems. Tartu 2018, 179 p.
2. **Riivo Kikas**. Analysis of Issue and Dependency Management in Open-Source Software Projects. Tartu 2018, 115 p.
3. **Ehsan Ebrahimi**. Post-Quantum Security in the Presence of Superposition Queries. Tartu 2018, 200 p.
4. **Ilya Verenich**. Explainable Predictive Monitoring of Temporal Measures of Business Processes. Tartu 2019, 151 p.
5. **Yauhen Yakimenka**. Failure Structures of Message-Passing Algorithms in Erasure Decoding and Compressed Sensing. Tartu 2019, 134 p.
6. **Irene Teinemaa**. Predictive and Prescriptive Monitoring of Business Process Outcomes. Tartu 2019, 196 p.
7. **Mohan Liyanage.** A Framework for Mobile Web of Things. Tartu 2019, 131 p.
8. **Toomas Krips.** Improving performance of secure real-number operations. Tartu 2019, 146 p.
9. **Vijayachitra Modhukur.** Profiling of DNA methylation patterns as biomarkers of human disease. Tartu 2019, 134 p.
10. **Elena Sügis.** Integration Methods for Heterogeneous Biological Data. Tartu 2019, 250 p.
11. **Tõnis Tasa.** Bioinformatics Approaches in Personalised Pharmacotherapy. Tartu 2019, 150 p.
12. **Sulev Reisberg.** Developing Computational Solutions for Personalized Medicine. Tartu 2019, 126 p.
13. **Huishi Yin.** Using a Kano-like Model to Facilitate Open Innovation in Requirements Engineering. Tartu 2019, 129 p.
14. **Faiz Ali Shah.** Extracting Information from App Reviews to Facilitate Software Development Activities. Tartu 2020, 149 p.
15. **Adriano Augusto**. Accurate and Efficient Discovery of Process Models from Event Logs. Tartu 2020, 194 p.
16. **Karim Baghery.** Reducing Trust and Improving Security in zk-SNARKs and Commitments. Tartu 2020, 245 p.
17. **Behzad Abdolmaleki.** On Succinct Non-Interactive Zero-Knowledge Protocols Under Weaker Trust Assumptions. Tartu 2020, 209 p.
18. **Janno Siim.** Non-Interactive Shuffle Arguments. Tartu 2020, 154 p.
19. **Ilya Kuzovkin.** Understanding Information Processing in Human Brain by Interpreting Machine Learning Models. Tartu 2020, 149 p.
20. **Orlenys López Pintado.** Collaborative Business Process Execution on the Blockchain: The Caterpillar System. Tartu 2020, 170 p.
21. **Ardi Tampuu.** Neural Networks for Analyzing Biological Data. Tartu 2020, 152 p.

22. **Madis Vasser.** Testing a Computational Theory of Brain Functioning with Virtual Reality. Tartu 2020, 106 p.
23. **Ljubov Jaanuska.** Haar Wavelet Method for Vibration Analysis of Beams and Parameter Quantification. Tartu 2021, 192 p.
24. **Arnis Parsovs.** Estonian Electronic Identity Card and its Security Challenges. Tartu 2021, 214 p.
25. **Kaido Lepik.** Inferring causality between transcriptome and complex traits. Tartu 2021, 224 p.
26. **Tauno Palts.** A Model for Assessing Computational Thinking Skills. Tartu 2021, 134 p.
27. **Liis Kolberg.** Developing and applying bioinformatics tools for gene expression data interpretation. Tartu 2021, 195 p.
28. **Dmytro Fishman.** Developing a data analysis pipeline for automated protein profiling in immunology. Tartu 2021, 155 p.
29. **Ivo Kubjas.** Algebraic Approaches to Problems Arising in Decentralized Systems. Tartu 2021, 120 p.