



Universidade do Minho



universidade
de aveiro

U.PORTO

Tools and Techniques for Energy-Efficient Mobile Application Development

Luís Miranda da Cruz

Supervisor: Rui Maranhão Abreu

Doctoral Program in Computer Science MAP-i of the Universities of Minho,
Aveiro, and Porto

April, 2019

© Luís Cruz: April, 2019

Tools and Techniques for Energy-Efficient Mobile Application Development

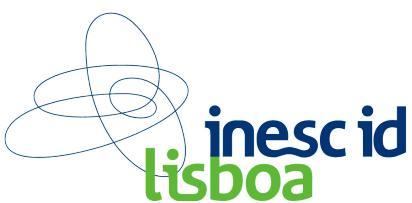
Luís Cruz

Thesis submitted to the Faculty of Engineering of University of Porto to
obtain the degree of

Doctor of Philosophy in Computer Science

April, 2019

The work developed in this thesis was supported by the scholarship number PD/BD/52237/2013 from Fundação para a Ciência e Tecnologia (FCT) and within the project GreenLab (POCI-01-0145-FEDER-016718).



Abstract

Driven by their convenience and ubiquitousness, smartphones have been widely adopted over the past ten years. As a consequence, mobile apps have rapidly become indispensable to perform everyday tasks. Users rely on mobile apps to accomplish tasks such as calling a taxi, renting a car, paying at grocery stores, ordering food, and so on. Nowadays, being unable to operate with a smartphone device, may imply being unable to carry out important tasks. This is the case when our smartphone runs out of battery.

Thus, it is essential to ensure that mobile apps are energy-efficient. However, as we demonstrate in this thesis, developing energy-efficient code is not as trivial as one could imagine. Previous research has tackled this issue by proposing techniques and tools to design energy-efficient mobile apps (e.g., studying anti-patterns) and reliably assess energy improvements (e.g., software-based energy estimators). In this thesis, we extend this research problem by studying how developers address energy efficiency, pinpointing potential issues entailed by building energy-efficient code, and proposing new techniques and practices to help developers address energy-related requirements.

We start by compiling the current state-of-the-art methodologies used to measure the energy consumption of mobile apps. Moreover, we study the limitations of these methodologies. In particular, we measure the energy overhead entailed by using User Interface testing frameworks during measurements. We show that popular frameworks such as *Calabash* and *AndroidViewClient* should be avoided when measuring energy consumption.

Following, we collect code smells from other non-functional requirements (in particular, performance) and explore their aptitude to improve energy efficiency. We find five performance-based code smells that effectively reduce energy consumption when fixed. Moreover, we study solutions typically applied to real-world energy efficiency issues. We collect and document a catalog of 22 energy patterns by inspecting energy-oriented changes in 1027 Android and 756 iOS apps. As a side contribution, we compare how Android and iOS developers address the energy efficiency of their apps.

We leverage energy patterns in an automatic refactoring tool. We use this tool to fix energy-based code smells in 45 open-source Android applications. Our results emphasize the importance of using tools to help developers improve the energy efficiency of their apps.

Last, we conduct an empirical study on the impact that managing the energy efficiency of mobile apps brings to the maintainability of the codebase. By analyzing a dataset of 539 energy efficiency-oriented changes, we show that improving energy efficiency hinders software maintainability. We discuss results and provide recommendations to help developers avoid common issues and adopt development processes that foster software maintainability.

Resumo

Impulsionados pela sua conveniência e ubiquidade, os telemóveis inteligentes, vulgarmente conhecidos como *smartphones*, tiveram uma grande adesão por todo o mundo nos últimos dez anos. Como consequência, os *smartphones* tornaram-se indispensáveis para realizar tarefas do dia-a-dia das pessoas. Os seus utilizadores recorrem a aplicações móveis para tarefas tão banais como chamar um táxi, alugar um carro, pagar no supermercado, encomendar comida, entre outras. Hoje em dia, ficar sem acesso a um *smartphone* pode implicar a impossibilidade de realizar tarefas importantes. Isto é o que acontece quando o telemóvel fica sem bateria.

Em consequência disso, é essencial garantir que as aplicações móveis fazem uma gestão eficiente da bateria – isto é, são energeticamente eficientes. No entanto, tal como demonstramos nesta tese, desenvolver código energeticamente eficiente não é tão simples como seria de imaginar. Trabalho relacionado estudou este problema propondo técnicas e ferramentas para desenhar aplicações mais eficientes (p. ex., estudando *code smells*) e para medir com fiabilidade melhoramentos no consumo energético (p. ex., usando estimadores de energia baseados em software). Nesta tese, alargamos o trabalho de investigação neste problema em diversas frentes: estudamos como os programadores tipicamente lidam com requisitos de eficiência energética; identificamos potenciais problemas relacionados com a criação de código energeticamente eficiente; e propomos novas técnicas e práticas para ajudar os programadores a criar software eficiente.

Começamos por fazer um levantamento do estado da arte das metodologias existentes para medição de energia em aplicações móveis. Estudamos também as limitações destas tecnologias. Nomeadamente, medimos o custo adicional no consumo de energia que advém da utilização de ferramentas de teste de interfaces de utilizador. Mostramos que ferramentas populares como *Calabash* e *AndroidViewClient* devem ser evitadas para medição de consumos de energia.

De seguida, colecionamos *code smells* relacionados com outros requisitos não funcionais (nomeadamente, performance) e exploramos o seu potencial para melhorar eficiência energética. Como resultado, descobrimos cinco *code smells* de performance que efetivamente reduzem o consumo de energia quando devidamente

corrigidos. Adicionalmente, estudamos soluções tipicamente aplicadas em problemas reais de eficiência energética. Como resultado de analisar este tipo de soluções em alterações de código de 1027 aplicações Android e 756 aplicações iOS, foi criado um catálogo com 22 padrões de energia devidamente documentados. Como contribuição suplementar, comparamos as diferenças entre programadores Android e programadores iOS a gerir a eficiência energética das suas aplicações.

Como contribuição, também desenvolvemos uma ferramenta de refatorização automática de *code smells* de energia – *Leafactor*. Recorrendo a esta ferramenta, foi possível corrigir *code smells* de energia em 45 aplicações open-source Android. Os resultados obtidos realçam a importância de ferramentas automáticas para ajudar programadores a melhorar a eficiência energética das suas aplicações móveis.

Por último, conduzimos um estudo empírico acerca do impacto que melhorar eficiência energética em aplicações móveis pode ter na sua facilidade de manutenção. Analisando um conjunto de dados com 539 alterações de código relacionadas com energia, mostramos que este tipo de alterações de código reduzem significativamente a facilidade de manutenção dos projetos. Discutimos os resultados e tecemos recomendações para ajudar os programadores de aplicações móveis a evitarem problemas recorrentes e a adotarem processos de desenvolvimento que promovem a facilidade de manutenção do software.

Acknowledgements

This thesis has only been made possible through the advice, help, and support of many people whom I would like to thank here.

First and foremost, I would like to express my most sincere and deepest gratitude to my supervisor and friend, Prof. Rui Maranhão Abreu. Throughout this journey, his endless support, patience, dedication, and professionalism were crucial to making this thesis possible. He is one of my best friends and a role model, both as a scholar and as a person.

I was fortunate enough to be able to collaborate with David Lo, from Singapore Management University, and with John Grundy, from Monash University. I want to thank them for all the stimulating conversations and thoughtful advice. I certainly hope to be able to work with both of them again.

My internship at Palo Alto Research Center was definitely an eye-opener and improved my perspective on research. Thank you, Jonathan Rubin, Alexander Feldman, Ion Matei, Nora Boettcher, Danny Bobrow, Johan de Kleer, Joy Smith, Wenjing Jin, Prakhar Jaiswal, Mayur Jain, and MinKyoung Kang, for all the support, kindness, friendship, and for recognizing my potential. You showed me that doing research is actually fun.

I want to thank all my Ph.D. student colleagues, particularly Alexandre Perez, Rui Pereira, Sofia Reis, and Nuno Cardoso, for all the shared frustrations, discussions, laughs, coffees, and beers.

I would also like to thank the IDSS group at INESC-ID. Prof. José Borbinha, for making me feel welcome since my first day there; Prof. Ricardo Chaves, for helping me with his hardware hacking skills; and all the folks at the “Junior IDSS”, for all our happy meetings.

My gratitude to the Green Software Lab, for being such an awesome group doing awesome research. To the SPeCS group, particularly Prof. João Cardoso, for letting me use his labs and equipment for my experiments.

To my friends in Barcelos, Alexandre Rodrigues, Mariana Carreira, Anthony Carvalho, Cláudio Novais, Ricardo Fonseca, Ricardo Vilas Boas, Nelson Jardim, Ana Silva, and

José Carlos Coelho. Thank you for all the breakfasts, surfed waves, trips, and dinners. I wouldn't be able to make it without all the fun moments we shared. (Alex, we will finish that surfboard).

To my friends in Porto, Pedro Cunha, David Miranda, Admilo Ribeiro, Ricardo Veiga, João Coelho, Daniel Moura, Lúcia Bandeirinha, Tiago Lourenço. Sorry for my absence in these last years, and thank you for being supportive and always prompt to listen to my complaints.

To my friends in Lisbon, Giulia Riggi, Beatrice Mainoli, Maria Meneses, and João Albuquerque. Thanks for all those dinners, nonsense arguments, dances, and all those things that made it so easy to escape from my thesis' frustrations. Cláudia Santos and João Sampaio, you have been the best housemates I could ask for.

Last, but certainly not least, I would like to thank my beloved family. My parents, thank you for all the understanding and help in all my decisions. *Pai, Mãe, muito obrigado por todo o carinho, paciência e apoio que me deram estes anos todos.* My sister, Cristina, thank you for being my best friend and for looking after me.

I am very fortunate to have all of you in my life.

Contents

List of Figures	xix
List of Tables	xxiii
List of Acronyms	xxv
1 Introduction	1
1.1 Concepts and Definitions	4
1.2 Problem Statement and Research Goals	4
1.3 Contributions	6
1.4 Thesis Outline	7
1.5 Origin of Chapters	7
2 Measuring Energy Consumption of Mobile Apps	9
2.1 Introduction	9
2.2 Set up power measurement tools	11
2.2.1 Energy profilers.	11
2.2.2 Power Monitors	12
2.3 Design a test case	14
2.4 Collect and Aggregate Power data	15
2.5 <i>Physalia</i> : a library to measure energy	16
2.6 Summary	18
3 Energy Footprint of UI Testing Frameworks	19
3.1 Introduction	19
3.2 Design of the Empirical Study	22
3.2.1 Energy Data Collection	23
3.2.2 UI testing frameworks	23
3.2.3 Test cases	24
3.2.4 Setup and Metrics	24
3.3 Results	27
3.3.1 Idle Cost	27
3.3.2 <i>Tap</i>	27
3.3.3 <i>Long Tap</i>	28

3.3.4	<i>Drag and Drop</i>	28
3.3.5	<i>Swipe</i>	30
3.3.6	<i>Pinch and Spread</i>	30
3.3.7	<i>Back Button</i>	32
3.3.8	<i>Input Text</i>	34
3.3.9	<i>Find by id</i>	34
3.3.10	<i>Find by description</i>	36
3.3.11	<i>Find by content</i>	36
3.3.12	Statistical significance	37
3.4	Discussion	38
3.5	Threats to validity	41
3.6	Related Work	42
3.7	Summary	44
4	Prevalence of Test Automation in Android Apps	45
4.1	Introduction	45
4.2	Related Work	49
4.3	Data collection	51
4.4	Prevalence of Automated Testing (RQ 4.1)	55
4.4.1	Results	56
4.4.2	Discussion	57
4.5	Evolution of the Testing Culture (RQ 4.2)	58
4.5.1	Results	59
4.5.2	Discussion	60
4.6	Automated Testing vs Popularity (RQ 4.3)	61
4.6.1	Results	64
4.6.2	Discussion	67
4.7	Code Issues and Test Automation (RQ 4.4)	68
4.7.1	Results	68
4.7.2	Discussion	70
4.8	Continuous Integration / Continuous Development (CI/CD) vs Test Automation (RQ 4.5)	70
4.8.1	Results	71
4.8.2	Discussion	72
4.9	Hall of Fame	73
4.10	Threats to validity	74
4.11	Summary	75
5	Energy Impact of Performance Anti-Patterns in Android Apps	77
5.1	Introduction	77
5.2	Empirical Study	79
5.2.1	Android Application Selection	79

5.2.2	Static Analysis and Refactoring	80
5.2.3	Generation of Automatic UI tests	82
5.2.4	Energy measurement tools setup	85
5.2.5	Experiments Execution	86
5.2.6	Data Analysis	86
5.3	Results	86
5.4	Discussion	90
5.5	Threats to the Validity	94
5.6	Related Work	95
5.7	Summary	96
6	Using Automatic Refactoring to Improve Energy Efficiency	99
6.1	Introduction	99
6.2	Energy Refactorings	101
6.2.1	ViewHolder: View Holder Candidates	101
6.2.2	DrawAllocation: Allocations within drawing code	103
6.2.3	WakeLock: Incorrect wakelock usage	103
6.2.4	Recycle: Missing recycle() calls	104
6.2.5	ObsoleteLayoutParams (OLP): Obsolete layout params	105
6.3	Automatic Refactoring Tool	106
6.3.1	AutoRefactor	106
6.3.2	XML refactorings	107
6.4	Empirical evaluation	107
6.5	Results	109
6.6	Discussion	112
6.7	Related Work	113
6.8	Summary	115
7	Catalog of Energy Patterns for Mobile Apps	117
7.1	Introduction	117
7.2	Related Work	118
7.3	Methodology	121
7.3.1	App Dataset Collection	121
7.3.2	Automatic gathering of commits, issues, and pull requests . .	122
7.3.3	Manual Refinement	124
7.3.4	Thematic Analysis	125
7.3.5	Reproducibility-Oriented Summary	126
7.4	Energy Patterns	126
7.4.1	Dark UI Colors	127
7.4.2	Dynamic Retry Delay	128
7.4.3	Avoid Extraneous Work	129
7.4.4	Race-to-idle	129

7.4.5	Open Only When Necessary	130
7.4.6	Push over Poll	130
7.4.7	Power Save Mode	130
7.4.8	Power Awareness	131
7.4.9	Reduce Size	131
7.4.10	WiFi over Cellular	132
7.4.11	Suppress Logs	132
7.4.12	Batch Operations	132
7.4.13	Cache	133
7.4.14	Decrease Rate	134
7.4.15	User Knows Best	134
7.4.16	Inform Users	135
7.4.17	Enough Resolution	135
7.4.18	Sensor Fusion	135
7.4.19	Kill Abnormal Tasks	136
7.4.20	No Screen Interaction	136
7.4.21	Avoid Extraneous Graphics and Animations	136
7.4.22	Manual Sync, On Demand	137
7.5	Data Summary and Discussion	137
7.5.1	Energy Patterns: Android vs. iOS	137
7.5.2	Co-occurrence of Patterns	139
7.5.3	Implications	141
7.6	Threats to validity	141
7.7	Summary	143
7.A	Grey Literature	143
8	Impact of Energy Patterns on Mobile App Maintainability	145
8.1	Introduction	145
8.2	Motivating Example & Research Questions	147
8.3	Methodology	150
8.3.1	Dataset	150
8.3.2	Baseline Commits	153
8.3.3	Maintainability Analysis	153
8.3.4	Typical Maintainability Issues	157
8.4	Results	158
8.5	Discussion	163
8.6	Threats to Validity	166
8.7	Related Work	167
8.7.1	Code maintainability	167
8.7.2	Energy patterns	168
8.7.3	Detecting Anti-patterns in Mobile Apps	168
8.8	Summary	169

9 Conclusions	171
9.1 Research Goals	171
9.2 Recommendations for Future Research	173
Bibliography	177

List of Figures

1.1 Steve Jobs announcing the Apple's 1st generation iPhone.	2
1.2 Typical UI of popular mobile apps.	3
1.3 Thesis Outline.	8
2.1 Architecture of the <i>Apple iPhone X</i>	10
2.2 Typical setup for energy measurements.	13
2.3 Energy consumption calculation.	17
3.1 Two versions of the example app.	21
3.2 Experimentation system to compare UI testing frameworks for Android. .	22
3.3 Violin plot of the results for the energy consumption of <i>Tap</i>	28
3.4 Violin plot of the results for energy consumption of <i>Long Tap</i>	29
3.5 Violin plot of the results for energy consumption of <i>Drag and Drop</i> . .	30
3.6 Violin plot of the results for energy consumption of <i>Swipe</i>	31
3.7 Violin plot of the results for energy consumption of <i>Pinch and Spread</i> . .	32
3.8 Violin plot of the results for energy consumption of <i>Back Button</i>	33
3.9 Violin plot of the results for energy consumption of <i>Input Text</i>	35
3.10 Violin plot of the results for energy consumption of <i>Find by id</i>	36
3.11 Violin plot of the results for energy consumption of <i>Find by description</i> . .	37
3.12 Violin plot of the results for energy consumption of <i>Find by content</i> . .	38
3.13 Selecting the most suitable framework for energy measurements.	39
4.1 Flow of data collection in the study.	52
4.2 Categories of apps included in our study with the corresponding app count for each category.	54
4.3 Distribution of the number of apps by age (in years).	54
4.4 Number of projects per framework.	56
4.5 Percentage of Android apps developed in projects with test cases over the age of the apps.	59
4.6 Cumulated frequency of projects with and without tests (from 9 to 0 years old), normalized by the total number of projects.	60

4.7	Cumulated percentage of projects with tests (from 9 to 0 years old), normalized by the total number of projects. All test categories are represented.	60
4.8	Boxplots with the distributions of the popularity metrics.	64
4.9	Do tests attract newcomers?	66
4.10	Tests: cause or consequence of a big community?	66
4.11	Comparison of the number of issues per file in projects with and without tests.	69
4.12	Android apps using CI/CD platforms.	71
4.13	Relationship between apps using CI/CD and apps using tests.	72
5.1	Example of a tree with the hierarchy of UI components in an Android app.	78
5.2	Experiments' workflow.	85
5.3	Energy consumption for <i>Loop - Habit Tracker</i>	88
5.4	Energy consumption for <i>Writeily Pro</i>	88
5.5	Energy consumption for <i>Talalarimo</i>	89
5.6	Energy consumption for <i>GnuCash</i>	89
5.7	Energy consumption for <i>Acrylic Paint</i>	90
5.8	Energy consumption for <i>Simple Gallery</i>	90
6.1	Architecture diagram of the automatic refactoring toolset.	106
6.2	LoF entry	107
6.3	Experiment's procedure for a single app.	108
6.4	Number of apps per category in the dataset.	109
6.5	An example of a PR containing code changes authored using <i>Leafactor</i> that was submitted to the Github project of the Android app <i>Slide</i>	110
6.6	Number of apps affected per refactoring.	111
7.1	Methodology used to extract energy patterns from mobile apps.	121
7.2	Distribution of categories in Android apps.	122
7.3	Distribution of categories in iOS apps.	123
7.4	UI themes with dark colors are more energy-efficient.	128
7.5	Power Save Mode allows to run the app in two different modes: a fully-featured mode and an energy-efficient mode.	131
7.6	Illustration of the energy pattern Batch Operation.	133
7.7	Comparison of the usage of energy patterns between Android and iOS mobile apps.	139
7.8	Co-occurrence of energy patterns in the same mobile application.	140
8.1	Methodology for data collection.	151
8.2	BCH's maintainability report of the app <i>NetGuard</i> for the guideline <i>Write short units of code</i>	154

8.3	Maintainability difference for the energy commit v_E .	157
8.4	Categories of apps included in our study with the corresponding app count for each category.	158
8.5	Maintainability differences for energy commits and baseline commits.	159
8.6	Maintainability differences among different types of energy commits.	160

List of Tables

3.1	Overview of the studied UI testing frameworks.	24
3.2	Android device's system Settings.	25
3.3	Descriptive statistics of <i>Tap</i> interaction.	27
3.4	Descriptive statistics of <i>Long Tap</i> interaction.	29
3.5	Descriptive statistics of <i>Drag and Drop</i> interaction.	30
3.6	Descriptive statistics of <i>Swipe</i> interaction.	31
3.7	Descriptive statistics of <i>Pinch and Spread</i> interaction.	32
3.8	Descriptive statistics of <i>Back Button</i> interaction.	33
3.9	Descriptive statistics of <i>Input Text</i> interaction.	34
3.10	Descriptive statistics of <i>Find by id</i> interaction.	35
3.11	Descriptive statistics of <i>Find by description</i> interaction.	36
3.12	Descriptive statistics of <i>Find by content</i> interaction.	37
4.1	Android tools analyzed.	53
4.2	Statistical analysis of the impact of tests on the popularity of apps.	65
4.3	Descriptive statistics of code issues on apps with (<i>W</i>) and without (<i>WO</i>) tests.	69
4.4	Statistical analysis of the impact of tests in mobile software code issues.	70
4.5	Hall of fame.	74
5.1	Metrics of applications used in experiments.	80
5.2	Anti-patterns found in open source applications.	82
5.3	Descriptive statistics of experiments	87
5.4	Significance Welch's t-test results.	91
5.5	Effect size of significant patterns.	91
6.1	Descriptive statistics of projects in the dataset.	109
6.2	Summary of refactoring results.	110
7.1	Descriptive statistics of the popularity metrics of the Android and iOS apps in the dataset.	123
7.2	Energy patterns' occurrences and related work.	127

8.1 Datasets of energy-oriented changes that were combined from previous work.	151
8.2 Example of a BCH report for a non-compliant case.	155

List of Acronyms

ADB	Android Debug Bridge
AIG	Automated Input Generation
API	Application Programming Interface
APK	Android Application Package
CEO	Chief Executive Officer
CI/CD	Continuous Integration / Continuous Development
CLI	Command Line Interface
CL	Common Language Effect Size
CPU	Central Processing Unit
CSV	Comma-Separated Values
DUT	Device Under Test
EFG	Event-flow Graph
FOSS	Free and Open Source Software
GPS	Global Positioning System
GPU	Graphics Processing Unit
IDE	Integrated Development Environment
IoT	Internet of Things
JVM	Java Virtual Machine
LOC	Lines of Code
OLED	Organic Light-Emitting Diode
AMOLED	Active Matrix Organic Light-Emitting Diode
OS	Operating System

PDA Personal Digital Assistant
PR Pull Request
SDK Software Development Kit
UI User Interface
URL Uniform Resource Locator
USB Universal Serial Bus
HTTP HyperText Transfer Protocol
REST Representational State Transfer
SI International System of Units

Introduction



Question everything generally thought to be obvious.

— Dieter Rams

Dating back to 2007, a particular event changed the history of software engineering. Steve Jobs, at the time **Chief Executive Officer (CEO)** of Apple, announced a new mobile phone device: the *iPhone* (cf. Figure 1.1). This new mobile device ran the *iOS* mobile operating system and, amongst other innovations¹, it introduced a multitouch interface, a capacitive touchscreen², threaded text messaging, and a mobile web browser with well-designed zooming and scrolling features [Grier, 2017].

However, the breakthrough innovation happened in the following year (2008), when a new platform to acquire and publish mobile applications (apps, for short) was announced: the *App Store*. For the first time, users could retrieve, buy, download, and install any app in their phone simply by using this distribution platform. In addition, the *App Store* took care of a number of tasks that, until then, had to be ensured by developers: listing, hosting and installing the apps, processing payments, certificating developers, and so on. By ensuring a big deal of the work entailed by publishing an application, more developers were attracted to join in and start building their apps. From the initial set of 500 iOS apps in 2008, Apple's *App Store* grew up to over 2.2 million apps, as of 2018³. Quickly, other markets have joined in: *Google Play*, *Amazon Appstore*, *MyApp*, *360 Mobile Assistant*, *Xiaomi App Store*, amongst others. As of 2018, *Google Play* was the biggest mobile app store, featuring over 3.3 million Android apps.

Over the years, smartphones have become increasingly more powerful. Mobile and wearable devices are nowadays the *de facto* personal computers, while desktop computers are becoming less popular. Software products and services have been increasingly focusing on mobile platforms as the main consumer target. For instance,

¹The article “*iOS: A visual history*” by *The Verge* gives a broader picture on the evolution of the iPhone: <https://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad> (Visited on June 5, 2020)

²Until then, the most similar alternatives were the **Personal Digital Assistant (PDA)** phones that featured a resistive touchscreen designed to be used with a stylus pen.

³An extensive list of mobile app markets is available here: <http://www.businessofapps.com/guide/app-stores-list/> (Visited on June 5, 2020).



Figure 1.1: Steve Jobs announcing the Apple's 1st generation iPhone (July 9, 2007). Source: Apple Inc.

as of 2019, the popular social media app *Instagram*⁴ provides messaging features in its mobile app that are not available in the analogous Web app. Likewise, the security mechanism *two-factor authentication* (also known as 2FA), which is commonly featured in desktop software, often requires users to have access to a smartphone.

The convenience of using smartphones allows users to accomplish important tasks ubiquitously. Examples of typical mobile apps are present in Figure 1.2. Users have access to a wide range of apps that allow them to perform everyday tasks, such as hailing a taxi (Figure 1.2a), making payments (Figure 1.2b), ordering food (Figure 1.2c), or even help them manage their medication intake (Figure 1.2d), and so on. In consequence, being unexpectedly deprived of using a smartphone means not being able to accomplish essential tasks. For instance, in an extreme scenario, if a user relies on a medication tracking app such as the one in Figure 1.2d, it might mean not taking medications as prescribed. In fact, an increasing number of users have been observed to suffer from anxiety, nervousness or discomfort when out of contact with a mobile phone – a condition technically referred as *nomophobia* [Bragazzi and Del Puente, 2014]. This raises the importance of one of the most significant limitations of smartphones: these devices run on batteries and have limited power resources. Hence, the more tasks users need to accomplish with their smartphones the less battery they will have available through the day.

A large scale study with over 4,000 smartphone users found that, in order to prevent losing connectivity, they charge their devices several times during the day [Ferreira et al., 2011]. In 2016, the manufacturer *LG* reported that nearly 90% of smartphone

⁴*Instagram* is a photo and video-sharing social networking service, available at: <http://www.instagram.com/> (Visited on June 5, 2020).

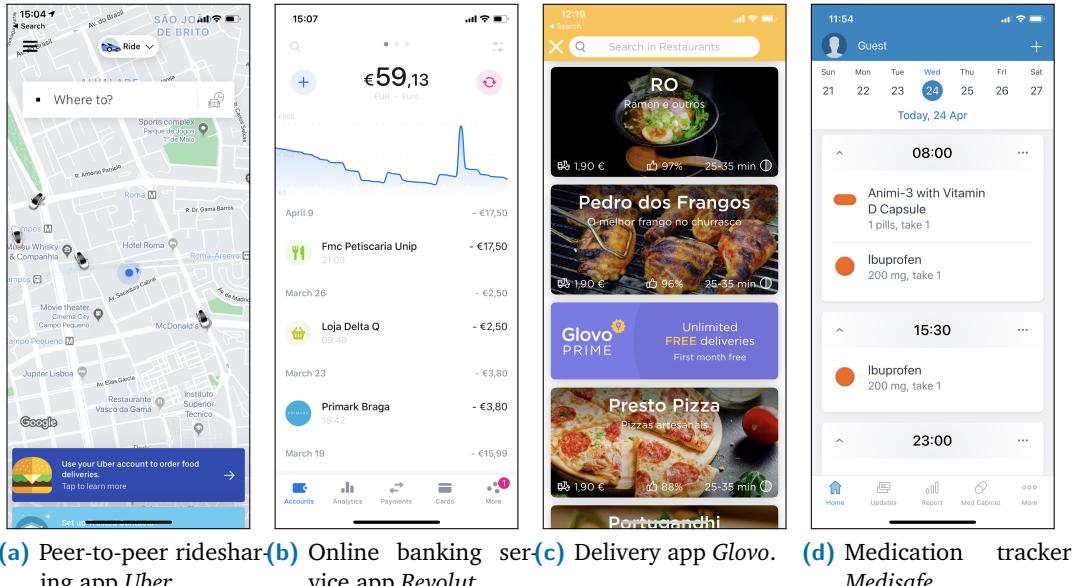


Figure 1.2: Typical UI of popular mobile apps.

users suffered from the fear of losing power on their phone⁵. They dubbed this condition as *Low Battery Anxiety*. Another marketing study conducted by the Australian comparison site *Finder*⁶ found evidence that longer battery life is the most desired feature for 89% of users in their next smartphone. Thus, energy efficiency in mobile applications is an essential concern for both users and developers. Apps that drain the battery life of mobile devices can ruin the user experience and, therefore, tend to be removed unless they offer an essential feature to users. Moreover, previous work found that users change their app usage patterns according to the battery level of their phone [Hosio et al., 2016]. In other words, users will stop using certain apps when their devices reach particular battery levels. Thus, any improvement in the energy efficiency of an application has a significant impact on its success.

However, related work shows that developers lack the knowledge of best practices to address the energy efficiency of mobile applications [D. Li and Halfond, 2014; Robillard and Medvidovic, 2016]. In particular, mobile apps often have energy requirements, but developers are unaware that energy-specific design patterns do exist [Manotas et al., 2016].

This thesis aims to help developers create energy-efficient mobile apps. We investigate existing limitations faced by mobile practitioners and leverage techniques that can be adopted to improve energy efficiency.

⁵LG media publication with reports on *Low Battery Anxiety*: <https://www.prnewswire.com/news-releases/low-battery-anxiety-grips-9-out-of-ten-people-300271604.html> (Visited on June 5, 2020)

⁶Finder's official website: <https://www.finder.com.au/> (Visited on June 5, 2020)

1.1 Concepts and Definitions

Throughout this document, we use the following definitions:

Definition 1 (Energy Efficiency) *Characteristic of a software system that indicates how well the hardware's energy resources are being used.*

Definition 2 (UI testing framework) *A framework that is able to mimic user interactions (e.g., tap, swipe, etc.) in the user interface of a given application.*

Definition 3 (Energy Test) *A sequence of interactions with a software application that explores a particular use case to assess energy efficiency.*

Definition 4 (Design Pattern) *A general, repeatable solution to common problems faced when solving software engineering design problems [Gamma, 1995]. Typically, a design pattern systematically names, motivates and explains a recommended solution.*

Definition 5 (Energy Pattern) *A design pattern that is known to improve the energy efficiency of a software system.*

Definition 6 (Code Refactoring) *The process of restructuring the code to improve non-functional requirements and reduce technical debt [Fowler, 2018].*

Definition 7 (Code smell) *A common sign in the source code that may indicate a deeper problem in the software and should be refactored [Fowler, 2018].*

Definition 8 (Maintainability) *As defined by the International Standards on software quality ISO/IEC 25010 is “The degree of effectiveness and efficiency with which a software product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements” [International Organization for Standardization, 2011].*

1.2 Problem Statement and Research Goals

As explained above, energy efficiency is a non-functional software requirement that arose with the advent of smartphones. A naive approach could address energy efficiency as a performance requirement. In this case, energy consumption would be modeled as a function of **Central Processing Unit (CPU)** cycles, and other simple metrics. However, related work has shown this is a very rough estimation given all the subcomponents that run under-the-hood of modern smartphones (e.g., location sensors, haptic feedback engine, **Graphics Processing Unit (GPU)**, multiple heterogeneous CPUs, network connections) [R. A. A. Pereira, 2018; S. Chowdhury et al., 2018a].

Given the recency of the new mobile computing paradigm, little is known on how to build energy-efficient mobile apps [D. Li and Halfond, 2014]. It is necessary to study approaches that address the idiosyncrasies of mobile apps and their running environments [Muccini et al., 2012].

Thus, in this thesis, we address the following main research question:

Main Research Question

What are the inherent limitations of state-of-the-art approaches to improving the energy efficiency of mobile applications, and what can be done to help developers address them?

Motivated by this main research question, this thesis incorporates the following research goals:

Research Goal I: *Describe the state-of-the-art approaches to measure the energy efficiency of mobile applications* (Chapter 2).

Practitioners can resort to different approaches to measure energy efficiency. Each of them has advantages and disadvantages that need to be considered when performing energy tests. We describe the state-of-the-art approaches and propose a methodology to perform reliable energy tests.

Research Goal II: *Study which User Interface (UI) testing frameworks can provide reliable energy efficiency assessments and check if existing mobile application projects are ready for automated energy testing approaches* (Chapters 3 and 4).

Energy tests require using UI testing frameworks to make sure measurements are repeatable. However, these frameworks are not designed for energy tests. Thus, it is important to understand which frameworks can be safely used for energy tests. In addition, we want to understand potential challenges in extending general-purpose testing practices to energy-efficiency requirements by analyzing the current testing culture in **Free and Open Source Software (FOSS)** Android projects.

Research Goal III: *Study and document best practices and recurrent solutions that can be reused to improve the energy efficiency of mobile apps.* (Chapters 5 and 7).

Energy-efficiency improvements typically require specialized developers. They need to come up with different solutions depending on the app, targeted users, context, and so on. We want to understand which practices can be reused by other developers to design energy-efficient mobile apps.

Research Goal IV: *Develop static analysis and automatic refactoring tools to help practitioners create energy-efficient mobile apps* (Chapter 6).

We study how automatic refactoring tools can help practitioners develop mobile apps that are free of energy *code smells*. Moreover, we study their benefits in real-world FOSS Android apps.

Research Goal V: *Assess the impact of improving energy efficiency in the maintainability of mobile software projects* (Chapter 8).

While improving the energy efficiency of mobile apps, developers ought to ascertain that their apps are maintainable. However, energy patterns require code changes that are not always simple and may affect different parts of app codebases. We aim to assess this impact and find energy patterns that require more attention.

1.3 Contributions

In this thesis, we deliver a number of contributions to help developers assess and ensure the energy efficiency of mobile applications:

- A methodology to reliably measure the energy consumption of mobile applications. The methodology is implemented and released in the Python library *Physalia*: <https://github.com/tqrg/physalia>.
- A comprehensive comparison of the overhead of UI testing frameworks on energy testing.
- Best practices regarding the **Application Programming Interface (API)** usage of UI testing frameworks for energy tests.
- A decision tree to help choose the UI testing framework which suits a given mobile software project.
- An empirical study on the adoption of testing techniques in the Android developer community.
- A static analysis tool to detect the usage of state-of-art testing frameworks. Available at: https://github.com/luiscruz/android_test_inspector.
- A comprehensive investigation of the relationship of automated test adoption with quality and popularity metrics for Android apps.
- An empirical study on the relationship between automated testing and CI/CD adoption.
- A selection of FOSS mobile apps that comply with testing best practices. Available at: https://luiscruz.github.io/android_test_inspector/.
- Empirical validation of performance-based best practices to improve the energy efficiency of Android apps.
- An automated refactoring tool, *Leafactor*, to apply energy efficiency best practices in Android application code bases.

- The submission of 59 **Pull Requests (PRs)** to the official code bases of 45 FOSS Android applications, comprehending 222 energy efficiency refactorings.
- A catalog of 22 energy patterns with a detailed description and instructions for mobile app developers and designers. It is available online: <https://tqrg.github.io/energy-patterns> and contributions from the community are enabled as PRs.
- A dataset with 1563 commits, issues, and PRs in which mobile app development practitioners address the energy efficiency of their apps. The dataset and collection tools are available online: <https://github.com/TQRG/energy-patterns>.
- An empirical comparison of how the energy efficiency of mobile app development is addressed in different platforms (viz. Android and iOS).
- An investigation of the impact of energy patterns in code maintainability. The reproducibility package is available here: <https://figshare.com/s/989e5102ae6a8423654d>.

A comprehensive explanation of these contributions is provided throughout this thesis.

1.4 Thesis Outline

The outline for the rest of this thesis is depicted as a tree diagram in Figure 1.3. As the root of the tree diagram is the main topic: mobile apps. Any chapters in different root-to-leaf paths can be read in parallel.

We start by studying energy efficiency in terms of methodology tools and techniques – Chapters 2 and 3. In parallel, we conduct an empirical study on the global picture of testing in FOSS Android apps (Chapter 4). We get back to energy efficiency by studying code patterns: in Chapter 5, we study which performance-based code patterns can be used to address energy efficiency, and, in Chapter 6, we leverage an automatic refactoring tool implementing those patterns. Finally, we build a catalog of energy patterns in Chapter 7, and study their impact on maintainability in Chapter 8. Conclusions and future work are discussed in Chapter 9.

All the URL references in this document have been saved in the *WayBackMachine* internet archive. In case an URL is not available, it can be accessed from <http://web.archive.org>.

1.5 Origin of Chapters

All chapters in this thesis have either been published in peer-reviewed journals and conferences or are currently under review. All publications have been co-authored

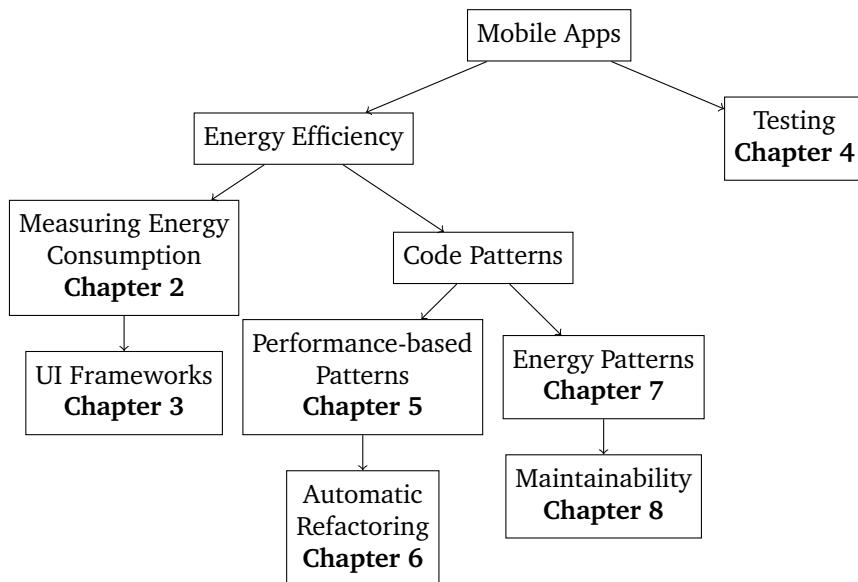


Figure 1.3: Thesis Outline.

with Rui Abreu. Publication of Chapter 4 has also been co-authored with David Lo. Publication of Chapter 8 has been co-authored with John Grundy, Li Li, and Xin Xia.

Chapter 3 is based on work submitted to the *IEEE Transactions on Software Engineering (TSE)*, which is currently under review. A preliminary version of this work was published as a poster in the *Proceedings of the International Conference on Software Engineering (ICSE'18)* [Cruz and Abreu, 2018a].

Chapter 4 is based on the work from [Cruz et al., 2019b], published in the *Empirical Software Engineering (EMSE)* journal.

Chapter 5 is based on work from [Cruz and Abreu, 2017], published in the *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2017)*.

Chapter 6 is based on work submitted to the *Journal of Software Engineering Research and Development (JSERD)*. An earlier version of this work was published in the *Ibero-American Conference on Software Engineering (CIBSE)* [Cruz and Abreu, 2018b], having been distinguished with the Best Paper award. A preliminary version of this work was published as a tool demo short paper in the *IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft 2017)* [Cruz et al., 2017].

Chapter 7 is based on the work from [Cruz and Abreu, 2019a], published in the *Empirical Software Engineering (EMSE)* journal.

Chapter 8 is based on work submitted to *The International Conference on Software Maintenance and Evolution (ICSME)*, which is currently under review.

Measuring Energy Consumption of Mobile Apps

Abstract

High energy consumption is a challenging issue that an ever increasing number of mobile applications face today. Despite being an important non-functional requirement of an application, energy consumption is being tested in an ad hoc way. In this chapter, we describe the state-of-the-art approaches to measure energy consumption. We present the two main categories of measurement tools: 1) energy profilers, and 2) power monitors. We pinpoint the benefits and limitations of the two categories. Moreover, we enumerate the required procedures to ensure the collection of reliable energy measurements. For instance, measuring energy consumption delves into mimicking user interactions, performing statistical validations, and so on. Finally, we present the library Physalia created to aid developers and researchers to gather reliable energy measurements. This library was used throughout the experiments in this thesis.

2.1 Introduction

When compared to traditional desktop applications, mobile apps run on devices with limited resources: lower computing capacity, less memory, lower power source, and so on. Nevertheless, mobile devices have very complex architectures, as illustrated in Figure 2.1. These architectures are designed to run operative systems that restrict apps to ensure optimal usage of resources. Thus, measuring energy consumption is a challenging task that developers and researchers face when testing the energy efficiency of their mobile apps.

Virtually, all the components presented in Figure 2.1 can be used during the execution of an application. Thus, the energy consumption of a mobile application cannot be constructed by merely measuring the execution time or the number of CPU cycles. A typical example that demonstrates this is the usage of different types of CPUs: smartphone architectures use fast but power-hungry CPUs for heavier tasks, and slow CPUs that consume less energy for simpler tasks (e.g., heterogeneous architecture *big.LITTLE*¹ used in mobile devices) [Yu et al., 2013; Diop et al., 2014]. Another example is the energy consumption entailed by starting and stopping components – a phenomenon coined as *tail energy consumption* [Pathak et al., 2011b]. Thus, it is not possible to measure energy consumption by measuring usage time. For example,

¹*big.LITTLE* architecture: <https://www.arm.com/why-arm/technologies/big-little> (Visited on June 5, 2020).

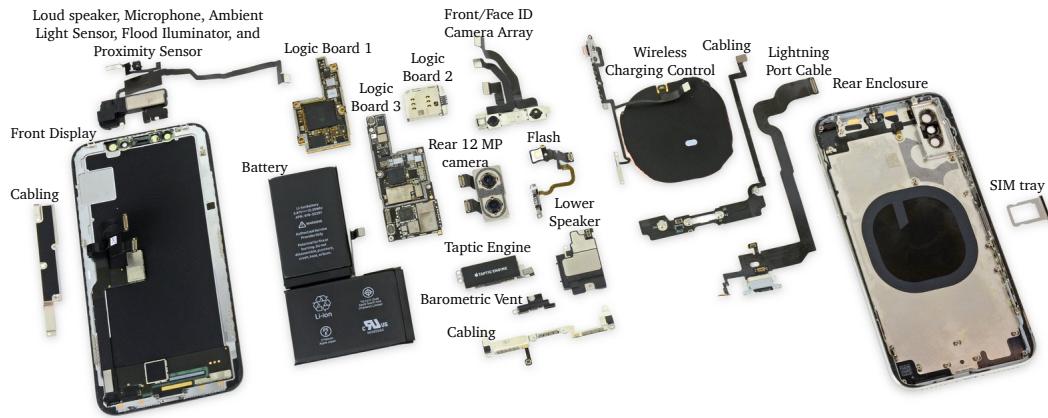


Figure 2.1: Architecture of the *Apple iPhone X* (2017). Labels are depicted on top of each component. Logic boards integrate several other components, such as, CPU (*Apple A11*), NFC controller, Bluetooth, Audio codec, Cellular receiver, etc.

a single task that requires using a component for T seconds will spend less energy than two tasks that need $T/2$ seconds and run separately, in the same conditions.

Moreover, energy consumption can be affected by many other factors: temperature, background tasks, display brightness level, and so on. Different executions of the same code may have different energy consumptions. Thus, measuring discernible improvements to the energy efficiency of an application is not only challenging but also time-consuming. For this reason, researchers have resorted to a number of different approaches to measure energy consumption.

In this chapter, we describe the different approaches used in previous work to measure the energy consumption of mobile applications, and pinpoint their main advantages and disadvantages. In sum, there are two main groups: 1) estimation-based measurements, using **energy profilers**, which are typically easy to set up but less reliable and can only be used in a small set of contexts, and 2) hardware-based estimations, using **power monitors**, which are complicated and hard to set up but provide more reliable measurements.

A single test for energy consumption comprehends the following steps:

1. Set up power measurement tools.
2. Design a test case, i.e., mimic the user interaction of a typical use case scenario of the mobile app under test.
3. Repeat the measurement at least 30 times.
4. Collect and aggregate power data.
5. Analyze results.

In this chapter, we further explain the state-of-the-art approaches to implement the steps mentioned above.

2.2 Set up power measurement tools

There are two main approaches to collect power data from the execution traces of mobile applications:

Energy profilers. Software tools that model energy consumption as a function of features that quantify impacting factors of energy consumption – e.g., number of CPU cycles, the amount of time **Global Positioning System (GPS)** was used, number of data packets transmitted using WiFi, and so on. These tools provide an estimation of the energy consumption.

Power monitors. Hardware-based tools that measure the power delivered to the smartphone in each timestamp. Power monitors provide **power measurements**, as opposed to the **energy estimates** collected with Energy profilers.

2.2.1 Energy profilers.

Previous work delivered software-based estimators aiming to collect reliable energy consumption data. These estimators typically model energy consumption as a function of features collected from the execution trace of the application and how long resources are being used (e.g., GPS, Cellular data, and so on).

State-of-the-art energy profilers (Android):

- *GreenScaler* [S. Chowdhury et al., 2018a].
- *AnaDroid* [Rua et al., 2019].
- *PETrA* [Di Nucci et al., 2017a].
- *Greenoracle* [S. A. Chowdhury and Hindle, 2016].
- *eProf* [Pathak et al., 2011b; Pathak et al., 2012a].
- *PowerTutor* [Zhang et al., 2010]. Used in [Zhang et al., 2012; Zhou et al., 2017; Zhou et al., 2015; Couto et al., 2015; Couto et al., 2014].
- *BatteryStats*². Delivered with the Android **Software Development Kit (SDK)**.
- *PowerProf* [Kjærgaard and Blunck, 2011].

²Documentation of *BatteryStats*: <https://developer.android.com/studio/profile/battery-historian> (Visited on June 5, 2020)

- *Trepn Power Profiler*³. Used in [Malavolta et al., 2017].
- *PowerBooter* [Zhang et al., 2010].
- *Sesame* [Dong and Zhong, 2011].
- *DevScope* [Jung et al., 2012].
- *AppScope* [Yoon et al., 2012].
- *V-edge* [Xu et al., 2013].
- *vLens* [D. Li et al., 2013]. Used in [D. Li et al., 2014a].
- *eLens* [Hao et al., 2013].

Solutions for iOS are more rare, including the *Energy Diagnostics Instrument*⁴ and the *Location Energy Instrument*, targeted for location-based use cases. Both are delivered with the iOS SDK. However, the documentation neither specifies which components are covered nor the overall expected accuracy provided by these tools.

A detailed comparison of energy profilers is addressed in previous work [Hoque et al., 2016]. Although these energy profilers can be very accurate, the validity of their measurements is highly dependent on the context in which they are used. For instance, as of 2016, *Trepn Profiler* was the only energy profiler considering the GPU as a source of energy consumption [Hoque et al., 2016]. Moreover, although most profilers report high accuracy, there is no standard benchmark to evaluate energy profilers. This means that the self-reported accuracy of 97.5% for *PowerTutor*[Zhang et al., 2010] is not necessarily higher than the self-reported accuracy of 94% for *eProf*[Pathak et al., 2012a]. In addition, there is no systematic approach to assess whether a particular profiler is adequate for a given context (e.g., a set of requirements).

2.2.2 Power Monitors

As an alternative to energy profilers, related studies have used hardware-based power monitors to measure energy consumption:

- *Monsoon Power Monitor*. Used in [Zhang et al., 2010; Banerjee and Roychoudhury, 2016; D. Li and Halfond, 2014; D. Li et al., 2014b; Cruz and Abreu, 2018a; Jung et al., 2012; Wan et al., 2017].

³*Trepn Power Profiler*'s website:<https://developer.qualcomm.com/trepn-profiler> (Visited on June 5, 2020).

⁴Apple's *Energy Diagnostics Instrument* documentation: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/MonitorEnergyWithInstruments.html> (Visited on June 5, 2020)

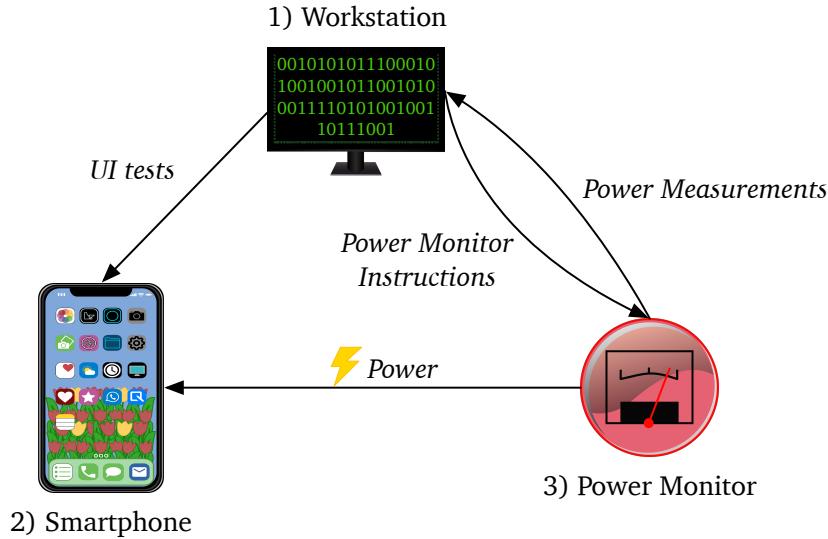


Figure 2.2: Typical setup for energy measurements.

- *ODROID*. Used in [Cruz and Abreu, 2017; Imes and Hoffmann, 2015; Shin et al., 2013]
- *NEAT* [Brouwers et al., 2014].
- *BattOr* [Schulman et al., 2011]. Used in [Aditya et al., 2014]
- *GreenMiner* [Hindle et al., 2014]. Used in [S. A. Chowdhury and Hindle, 2016]
- Custom solutions were used in [Abogharaf et al., 2012; Rasmussen et al., 2014; Sahin et al., 2016; Ferrari et al., 2015; Segata et al., 2014]

The main advantage of power monitors is the fact that they typically provide more accurate energy measurements. The drawback is that this approach requires a cumbersome preparation procedure⁵. A typical setup is depicted in Figure 2.2. It requires three main components: 1) a controlling workstation, 2) a smartphone, and 3) a power monitor. The workstation takes care of controlling the power monitor, with instructions to start/stop measurements and to set the power voltage supplied to the smartphone, and of triggering the execution of test cases in the smartphone. Finally, the power monitor sends the measured power data back to the workstation.

Under the hood, the smartphone needs to be disassembled and have its battery removed while the power monitor is connected directly to the power source of the smartphone. Such a procedure requires some hacking that may not be accessible for an ordinary developer. Besides, several measures need to be taken to ensure that the collected data is reliable. To name a few, differences in the temperature room may affect results, the USB port cannot be connected to the device, and so on.

⁵Step-by-step guide to setup a Monsoon Power Monitor with an Android smartphone: https://tqrg.github.io/physalia/monsoon_tutorial.html (Visited on June 5, 2020).

2.3 Design a test case

To measure the energy consumption of a mobile application, one needs to design the use case in which the energy consumption will be tested.

The test case is executed by interacting with the UI of the app. In other words, a test case is a sequence of UI interactions. These can be a simple *Tap*, *Double Tap*, a *Swipe*, a *Drag and Drop*, writing with the *Keyboard*, pressing the *Back Button* (in the case of Android), etc.

It is essential that the test case covers the scenario in which we want to test energy efficiency. For example, if we want to improve the energy efficiency of the use cases in which the app updates its data model from the cloud server, the test case should cover at least that part of the code. Although this may sound trivial, it can only be ensured if developers are aware of the code coverage achieved with their sequence of UI interactions.

Manual tests for energy efficiency are not reliable. This is different from testing other non-functional requirements. A test manually executed by a human entails delays between UI interactions that cannot be systematically reproduced. In other words, unnecessary delays during the execution of a test will delve into non-deterministic overheads in the measured energy consumption (e.g., unnecessary Display usage). Thus, automated test cases are more suitable to test energy efficiency.

Automated test cases are executed using UI frameworks. To name a few, the *Android SDK* is shipped with *MonkeyRunner*, *Robotium*, and *Espresso*. Besides, there are tools that record UI interactions and generate the respective test cases (e.g., *Espresso Test Recorder*, *Robotium Recorder*, etc.). We detail the state-of-the-art UI frameworks and analyze their reliability for energy efficiency tests later in Chapter 3.

An example of a UI test case extracted from the open source Android app *GnuCash*⁶ is presented below:

```
//class org.gnucash.android.test.ui.FirstRunWizardActivityTest
@Test
public void shouldDisplayFullCurrencyList(){
    assertThat(mAccountsDbAdapter.getRecordsCount()).isEqualTo(0);

    onView(withId(R.id.btn_save)).perform(click()); ❶

    onView(withText(R.string.wizard_option_currency_other)).perform(click());
    onView(withText(R.string.btn_wizard_next)).perform(click());
    onView(withText(R.string.wizard_title_select_currency)).check(matches(
        isDisplayed())); ❷

    onView(withText("AFA\u2022-\u2022Afghani")).perform(click());
    onView(withId(R.id.btn_save)).perform(click());
```

⁶*GnuCash* git repository: <https://github.com/codinguser/gnucash-android> (Visited on June 5, 2020).

```

onView(withText(R.string.wizard_option_let_me_handle_it)).perform(click());

onView(withText(R.string.btn_wizard_next)).perform(click());
onView(withText(R.string.wizard_option_disable_crash_reports)).perform(
    click());
onView(withText(R.string.btn_wizard_next)).perform(click());

onView(withText(R.string.review)).check(matches(isDisplayed()));
onView(withId(R.id.btn_save)).perform(click());

//default accounts should not be created
assertThat(mAccountsDbAdapter.getRecordsCount()).isZero(); ❸

boolean enableCrashlytics = GnuCashApplication.isCrashlyticsEnabled();
assertThat(enableCrashlytics).isFalse();

String defaultCurrencyCode = GnuCashApplication.getDefaultCurrencyCode();
assertThat(defaultCurrencyCode).isEqualTo("AFA");
}

```

In this test, a number of taps is executed to go over the initial setup wizard when the user installs the app. A single tap is executed with a statement akin to the one in ❶. First, the button is identified by its id (stored in R.id.btn_save) using the method `onView()`, and then a tap is instructed with the method call `perform(click())`. Given that we are not testing functional requirements, assertions akin to the ones in ❷, ❸ are not necessary.

2.4 Collect and Aggregate Power data

Energy measurements are affected by a myriad of external conditions. Despite the efforts to minimize most confounding factors, energy measurements can still undergo small deviations from the effective energy consumption entailed by the app. E.g., one can control the temperature of the device to a fixed temperature, but ensuring that the temperature is exactly the same in two different executions is not a trivial task. This leads to some level of non-determinism in energy measurements, making testing for energy efficiency difficult.

As a rule of thumb, measurements ought to be repeated at least 30 times to minimize bias [Sahin et al., 2014; Sahin et al., 2016; Cruz and Abreu, 2018a; Linares-Vásquez et al., 2014]. To assess whether energy consumption is different in two different versions of a given app, hypothesis testing is performed. It is formulated as follows:

$$H_0 : \mu_{W(A)} = \mu_{W(B)}$$

$$H_1 : \mu_{W(A)} \neq \mu_{W(B)}$$

where A and B are two different versions of an app, W denotes the energy consumption measured for a given version, and μ is the population mean.

Typically, the distribution of the mean of energy consumption for a given version follows a Normal distribution. This should be verified using a normality test (e.g., Shapiro-Wilk test) or using data visualizations (e.g., histograms, probability density function). Assuming data follows a Normal distribution, a parametric test is performed (e.g., Welch's t-test) using a significance level below 0.05. If the resulting p-value is below the significance level, then H_0 is rejected: there is statistical evidence that the two versions A and B have different energy consumptions.

In addition, the data gathered from power monitors and energy profilers is not necessarily a measurement of the total energy consumption. Data is returned as a set of pairs of timestamp (t) and power level (P): (t_i, P_i) .

The total energy consumption (i.e., work performed) between timestamps t_0 and t_n is calculated by integrating power over time:

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

where P is power and W is the energy consumption (i.e., work).

Since P is a continuous variable but what we have is a sample of measured values, mathematical integration needs to be approximated using a numerical approach based on the general Trapezoid Rule:

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

This calculation is illustrated in Figure 2.3. The energy spent during an experiment is given by the area of the function of Power between the timestamp when the interaction started (t_0) and the timestamp when interaction ended (t_n). As defined by the International System of Units, the unit to measure time is the second (s), power is watt (W), and energy consumption is joule (J). In the case illustrated in Figure 2.3, following the general Trapezoid Rule in Equation (2.2), the energy consumption is 28.71J.

2.5 *Physalia*: a library to measure energy

Given the particularities above, we leverage a *Python* library to help collect reliable measurements of app energy consumption: *Physalia*⁷. The library is available with

⁷*Physalia*'s official website: <https://tqrg.github.io/physalia/> (Visited on June 5, 2020)

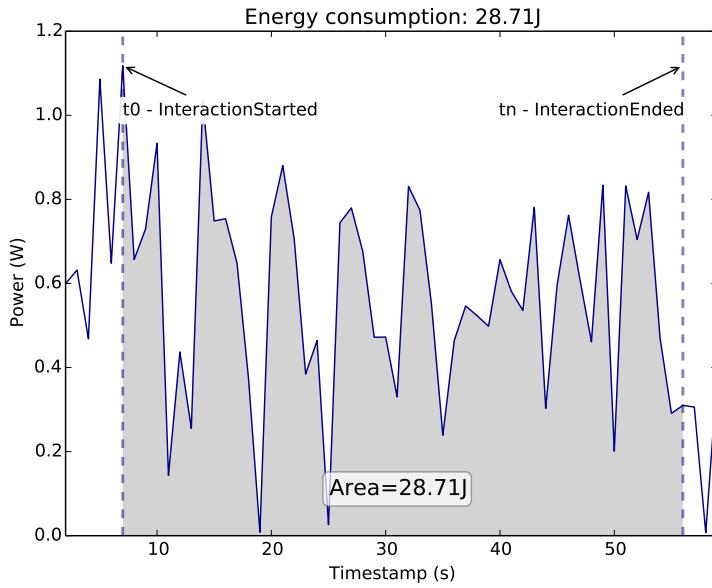


Figure 2.3: Energy consumption calculation.

an open source license and can be installed through the *Python* package manager *PyPI*⁸.

The library was designed to be compatible with any power monitor and energy profiler. Currently, it is fully tested to be used with the *Monsoon Power Monitor*. It takes care of the following details:

- Controlling an Android device under test. I.e., configure workstation connection to the device, install/uninstall **Android Application Packages (APKs)**, open/close apps, run test cases, and so on.
- Asynchronously instruct the power monitor to start and stop measuring energy consumption.
- Store and calculate the total energy consumption (cf. Equation (2.2)) based on the collected power data and perform hypothesis tests over samples.

The following snippet shows an example of a simple script using *Physalia* to measure the energy consumption entailed by launching an Android app.

```
from time import sleep
from physalia.power_meters import MonsoonPowerMeter
from physalia.energy_profiler import AndroidUseCase

MEASUREMENT_DURATION = 10 #seconds

def prepare(use_case): ❶
    use_case.prepare_apk()
def cleanup(use_case): ❷
```

⁸Package available here: <https://pypi.org/project/physalia/> (Visited on June 5, 2020)

```

use_case.kill_app()
def run(use_case): ❸
    use_case.open_app()
    sleep(MEASUREMENT_DURATION)

use_case = AndroidUseCase(
    "opening-native",
    app_apk=".apk/mobile_app.apk",
    app_pkg="pt.research.energyconsumption.mobileapp",
    app_version="0.0",
    prepare=prepare,
    run=run,
    cleanup=cleanup
)
power_meter = MonsoonPowerMeter(voltage=3.8, serial=12886) ❹
use_case.profile(power_meter=power_meter) ❺

```

- ❶ Set up and prepare the device and the app under test. In this case, it takes care of (re)installing the APK of the app.
- ❷ Clean the device after executing the energy measurement. In this case, the app is closed.
- ❸ The routine that is going to be measured. In this script, we only measure the energy consumption of opening the app and waiting 10 seconds.
- ❹ Set up the power meter to start measuring. The *Monsoon* power meter with serial number “12886” is set to output a voltage of 3.7V to the smartphone.
- ❺ Execute measurements 30 times and save data into a local CSV file.

2.6 Summary

In this chapter, we have studied the state-of-the-art approaches to test the energy consumption of mobile apps. Concretely, in this chapter:

- We provide researchers and developers with a detailed guide to measure the energy consumption of their mobile applications.
 - How to mimic real use case scenarios.
 - How to aggregate raw energy data.
 - How to determine whether two different app versions have different energy efficiency.
- We studied and listed existing tools to collect energy data.
- We propose the tool *Physalia* to help researchers and developers gather energy measurements complying with best practices.

Energy Footprint of UI Testing Frameworks



Measuring the Energy Footprint of Mobile Testing Frameworks

Luis Cruz and Rui Abreu

In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE, 2018.



On the Energy Footprint of Mobile Testing Frameworks

Luis Cruz and Rui Abreu

Submitted to IEEE Transactions on Software Engineering, 2019.

Abstract

As we have seen in Chapter 2, measuring energy consumption is not trivial. Such limitation becomes particularly disconcerting when mimicking typical usage scenarios of mobile applications: there is no knowledge as to what is the energy overhead imposed by the testing framework.

In this chapter, we study eight popular mobile UI testing frameworks to assess their overhead on energy measurements. We show that there are frameworks which increase energy consumption up to roughly 2200%. While limited in the interactions one can do, Espresso is the most energy-efficient framework. However, depending on the needs of the tester, Appium, Monkeyrunner, or UIAutomator are good alternatives. In practice, results show that deciding which is the most suitable framework is vital. We provide a decision tree to help developers make an educated decision on which framework suits best their testing needs.

3.1 Introduction

Automated testing tools help validate not only functional but also non-functional requirements such as scalability and usability [Morgado and Paiva, 2015; Moreira et al., 2013]. In Chapter 2, we have seen that the most reliable approach to measure the energy consumption of mobile software is by using user UI testing frameworks [D. Li et al., 2014a; S. Lee et al., 2015; Linares-Vásquez et al., 2014; Di Nucci et al., 2017b; Carette et al., 2017; Cao et al., 2017]. These frameworks are used to mimic user interaction in mobile apps while using an energy profiling tool. An alternative is to use manual testing, but it creates bias, is error-prone, and is both time and human resource consuming [Rasmussen et al., 2014].

While using a UI testing framework is the most suitable option to test apps, there are still energy-related concerns that need to be addressed. By replicating interactions, frameworks are bypassing or creating overhead on system behavior. For instance, before executing a *Tap*¹, it is necessary to programmatically look up the target UI component. This creates extra processing that would not happen during an ordinary execution of the app. These peculiarities are addressed in this chapter, as they may have a negative impact on energy consumption results.

As a motivational example, consider the following scenario: an app provides a *tweet* feed that consists of a list of *tweets* including their media content (such as, pictures, GIFs, videos, URLs). The product owner noticed that users rather value apps with low energy consumption. Hence, the development team has to address this non-functional requirement.

One idea is to show plain text and pictures with low resolution. Original media content would be rendered upon a user *Tap* on the *tweet*, as depicted in Figure 3.1. With this approach, energy is potentially saved by rendering only media that the user is interested in. To validate this solution, developers created automated scripts to mimic user interaction in both versions of the app while measuring energy consumption using a power meter. The script for the original version consisted in opening the app and scroll the next 20 items, whereas the new version's script consisted in opening the app and scrolling the next 20 items while tapping in 5 of them (a number they agreed to be the average hit rate of their users). A problem that arises is that the testing framework spends more energy to perform the five extra *Taps*. Imagining that for each *Tap* the testing framework consumes 1 joule² (J), the new version will have to spend at least 5J less than the original version to be perceived as more efficient. Otherwise, it gets rejected even though the new version could be more efficient.

More efficient frameworks could reduce this threshold to a more insignificant value. However, since testing frameworks have not considered energy consumption as an issue, developers do not have a sense of which framework is more suitable to perform reliable energy measurements.

In this chapter, we study popular UI testing frameworks in the context of testing the energy efficiency of mobile apps. This empirical study addresses the following research questions:

Research Question 3.1

Does the energy consumption overhead created by UI testing frameworks affect the results of the energy efficiency of mobile applications?

¹*Tap* is a gesture in which a user touches the screen with the finger.

²Joule (J) is the energy unit in the International System of Units.

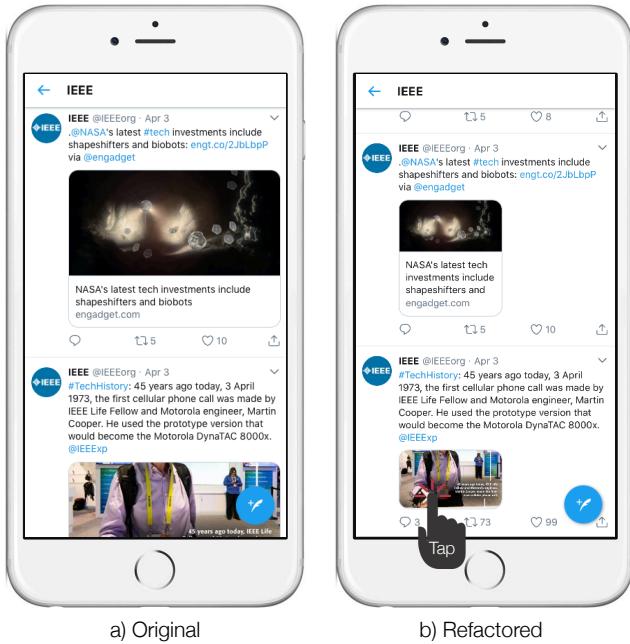


Figure 3.1: Two versions of the example app.

Research Question 3.2

What is the most suitable framework to profile energy consumption?

Research Question 3.3

Are there any best practices when it comes to creating automated scripts for energy efficiency tests?

We measure the energy consumption of common user interactions: *Tap*, *Long Tap*, *Drag And Drop*, *Swipe*, *Pinch & Spread*, *Back button*, *Input text*, *Find by id*, *Find by description*, and *Find by content*.

Results show that *Espresso* is the framework with the best energy footprint, although *Appium*, *Monkeyrunner*, and *UIAutomator* are also good candidates. On the other side of the spectrum are *AndroidViewClient* and *Calabash*, which makes them not suitable to test the energy efficiency of apps yet. We have further discovered that methods that use content to look up UI components need to be avoided since they are not energy savvy.

As main implication, overheads incurred by UI testing frameworks ought to be considered when measuring the energy consumption of mobile apps.

To sum up, this chapter's contributions are:

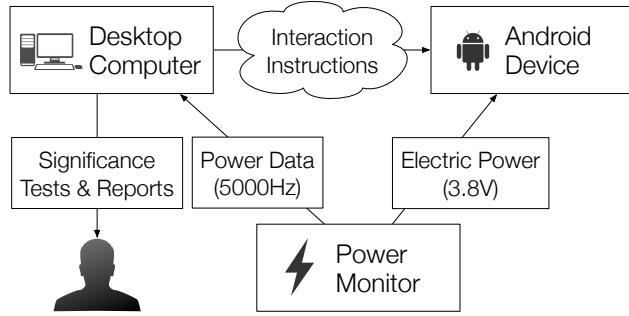


Figure 3.2: Experimentation system to compare UI testing frameworks for Android.

- A comprehensive study on the energy consumption of user interactions mimicked by UI testing frameworks.
- Comparison of the state-of-the-art UI testing frameworks and their features in the context of energy tests.
- Best practices regarding the API usage of the framework for energy tests.
- As a practical implication of our work, a decision tree to help choose the framework which suits one needs.

3.2 Design of the Empirical Study

To answer the research questions (cf. RQs 3.1 to 3.3), we designed an experimental setup to automatically measure energy consumption of Android apps. In particular, our methodology consists of the following steps:

1. Preparation of an Android device to use with a power monitor.
2. Creation of a stack of UI interaction scripts for all frameworks.
3. Automation of the execution of tests for each framework to run in batch mode.
4. Collection and analysis of data.

The methodology is illustrated in Figure 3.2. There are three main components: a desktop computer that serves as controller; a power monitor; and a mobile device running Android, i.e., the **Device Under Test (DUT)**. The desktop computer sends interaction instructions to be executed in the mobile device. The power monitor collects energy consumption data from the mobile device and sends it to the desktop computer. Finally, the desktop computer analyzes data and generates reports back to the user.

3.2.1 Energy Data Collection

We have adopted a hardware-based approach to obtain energy measurements. We use Monsoon’s *Original Power Monitor* with the sample rate set to 5000Hz, as used in previous research [Di Nucci et al., 2017b; Linares-Vásquez et al., 2014; D. Li and Halfond, 2014; D. Li et al., 2014b; Choudhary et al., 2015; Hindle, 2015; Banerjee and Roychoudhury, 2016; Banerjee et al., 2016]. Measurements are obtained using the *Physalia* toolset³, presented in Chapter 2. The steps described in *Physalia*’s tutorial⁴ were followed to remove the device’s battery and connect it directly to the Monsoon’s power source using a constant voltage of 3.8V. This is important to ascertain that we are collecting reliable energy consumption measurements.

3.2.2 UI testing frameworks

The state-of-the-art UI testing frameworks for Android used in our study are *AndroidViewClient*, *Appium*, *Calabash*, *Espresso*, *Monkeyrunner*, *PythonUIAutomator*, *Robotium*, and *UIAutomator*. The frameworks were chosen following a systematic criteria/review: freely available to the community, open source, featuring a realistic set of interactions, expressed through a human-readable and writable format (e.g., programming language), and used by the mobile development industry. To assess this last criterion *StackOverflow* and *Github* were used as a proxy. Some frameworks have been discarded for not complying with these criteria. As an example, *Ranorex*⁵ is not free to the community, and *RERAN* [Gomez et al., 2013] is designed to be used with a recording mechanism. *MonkeyTalk* has not been publicly released after being acquired by Oracle⁶, and *Selendroid* is not ready to be used with the latest Android SDK⁷. We decided not to include UI recording tools since they rely on the underlying frameworks (e.g., *Espresso Test Recorder*, *Robotium Recorder*).

Although most frameworks support using screen coordinates to specify interactions, we only study the usage by targeting UI components. Screen coordinates make the tests cumbersome to build and maintain, and are not common practice.

An overview of the features of the frameworks is in Table 3.1. It also details the frameworks as to whether the app’s source code is required, the ability to be used in remote script-based, i.e., whether simple interaction commands can be sent in real time to the DUT; WebView support, i.e., whether hybrid apps can also be automated;

³Physalia’s webpage: <https://tqrg.github.io/physalia/> (visited on June 5, 2020).

⁴Tutorial’s webpage: https://tqrg.github.io/physalia/monsoon_tutorial (visited on June 5, 2020).

⁵Ranorex’s website available at <https://www.ranorex.com> (visited on June 5, 2020).

⁶More information about *MonkeyTalk*’s acquisition: <https://www.oracle.com/corporate/acquisitions/cloudmonkey/> (visited on June 5, 2020)

⁷Running Selendroid would require changing its source code: <https://github.com/selendroid/selendroid/issues/1116> and <https://github.com/selendroid/selendroid/issues/1107> (visited on June 5, 2020)

Table 3.1: Overview of the studied UI testing frameworks.

Framework	Android View Client	Appium	Calabash	Espresso	Monkeyrunner	Python Ui Automator	Robotium	UIAutomator
Tap	✓	✓	✓	✓	✓	✓	✓	✓
Long Tap	✓	✓	✓	✓	✓	✓	✓	✓
Drag And Drop	✓	✓	✓	✗	✓	✓	✓	✓
Swipe	✓	✓	✓	✓	✓	✓	✓	✓
Pinch & Spread	✗	✗	✓	✗	✗	✓	✗	✓
Back button	✓	✓	✓	✓	✓	✓	✓	✓
Input text	✓	✓	✓	✓	✓	✓ ^(*)	✓ ^(*)	✓ ^(*)
Find by id	✓	✓	✓	✓	✓	✓	✓	✓
Find by description	✓	✓	✓	✓	✗	✓	✗	✓
Find by content	✓	✓	✓	✓	✗	✓	✓	✓
Tested Version	13.2.2	1.6.3	0.9.0	2.2.2	n.a.	0.3.2	5.6.3	2.1.2
Min Android SDK	All	Recmd. ≥ 17	All	≥ 8	n.a.	≥ 18	≥ 8	≥ 18
Black Box	Yes	Yes	Limited (**)	No	Yes	Yes	Yes	Yes
Remote script-based	Yes	Yes	Yes	No	Yes	Yes	No	No
WebView Support	Limited	Yes	Yes	Yes	Limited	Limited	Yes	Limited
iOS compatible	No	Yes	Yes	No	No	No	No	No
BDD support	No	Yes	Yes	Yes	No	No	Yes	Limited
Integration test	Yes	Yes	Yes	No	No	Yes	Yes	Yes
Language	Python	Any WebDriver compatible lang.	Gherkin/Ruby	Java	Jython	Python	Java	Java
License	Apache 2.0	Apache 2.0	EPL 1.0	Apache 2.0	Apache 2.0	MIT	Apache 2.0	Apache 2.0
SOverflow Qns	164	3,147	569	292	437	0	1,012	438
Github Stars	540	5,514	1,429	n.a.	n.a.	719	2,165	n.a.

(*) Although it supports *Input Text*, it does not apply a sequential input of key events. This is more energy-efficient but it is more artificial, bypassing real behavior (e.g., auto correct).

(**) Requires to manually enable Internet permission ("android.permission.INTERNET").

compatibility with iOS, and supported programming languages. The most common languages supported by these frameworks are *Python* and *Java*.

3.2.3 Test cases

For each framework, a script was created for every interaction that was supported by the framework, totaling 73 scripts. Scripts were manually and carefully crafted and peer reviewed to ascertain similar behavior across all frameworks. Essentially, each script calls a specific method of the framework that mimics the user interaction that we pretend to study. To minimize overheads from setup tasks (e.g., opening the app, getting app's UI hierarchy), the method is repeated multiple times (up to 200 times, depending on the complexity of the interaction).

3.2.4 Setup and Metrics

We compare the overhead in energy consumption using as baseline the energy usage of interactions when executed by a human. Baselines for each interaction were measured by asking two Android users (one female and one male) to execute the interactions as in the automated scripts. For instance, in one of the experiments the participants had to click 200 times in the *Back Button*. All interactions were measured except for *Find by id*, *Find by description*, and *Find by content*, as these are helper methods provided by the UI testing frameworks and do not apply to human interactions.

As mentioned above, energy measurements are prone to random variations due to the nature of the underlying **Operating System (OS)**. Furthermore, one can also expect errors from the data collected from a power monitor [Saborido et al., 2015]. To make sure energy consumption values are reliable and have enough data

Table 3.2: Android device’s system Settings.

Setting	Value
⚙️ Adaptive Brightness	ⓘ Manual - 78%
Bluetooth	ⓘ Off
WiFi	ⓘ On
Cellular	ⓘ No SIM card
📍 Location Services	ⓘ Off
▢ Auto-rotate screen	ⓘ Off - Portrait
🔔 Zen mode	ⓘ On - Total Silence
🔒 Pin/Pattern Lock Screen	ⓘ Off
● Don’t Keep Activities	ⓘ On
⌚ Account Sync	ⓘ Off
_Android Version	6.0.1

to perform significance tests, each experiment was identically and independently repeated 30 times.

Since user interactions often trigger other tasks in a mobile device, tests have to run in a controlled environment. Thus, an Android application was developed by the authors for this particular study. The main goal of the app is preventing any side-effect from UI interactions, which in real apps would result in different behaviors, hence compromising measurements. The app prevents the propagation of the system’s event created by the interaction, and no feedback is provided to the user. Consequently, experiments only measure the work entailed by frameworks. Measured energy consumptions generalize to real apps, given that frameworks operate similarly regardless of the app under test.

The main settings used in the device are listed in Table 3.2. Android provides system settings that can be useful to control system behavior during experiments. Notifications and alarms were turned off, lock screen security was disabled, and the “Don’t keep Activities” setting was enabled. This last setting destroys every activity as soon as the user switches to another, erasing the current state of an app⁸.

WiFi is kept on as a requirement of our experimental setup. The reason lies in the fact that Android testing frameworks resort to the **Android Debug Bridge (ADB)** to communicate with the mobile device. ADB allows to install/uninstall/open apps, send test data, configure settings, lock/unlock the device, among other things. By default, it works through USB, which interferes with energy consumption measurements. Although Android provides settings to disable USB charging, we have verified that measurements are not reliable in such setup. If the USB cable remains connected to the device, despite not being used to charge the battery, it is still used to power the device. Fortunately, ADB can be configured to be used through a WiFi connection, which was leveraged in this work.

⁸More about “Don’t Keep Activities” setting available at: <https://stackoverflow.com/questions/22400859/dont-keep-activities-what-is-it-for/32427857#32427857> (visited on June 5, 2020).

In addition to the energy consumption sources mentioned before, there is another common one – the cost of having the device in idle mode. In this context, we consider idle mode when the device is active with the settings in Table 3.2 but is not executing any task. In this mode, the screen is still consuming energy. We calculate the idle cost for each experiment to assess the effective energy consumption of executing a given interaction. We measure the idle cost by collecting the energy usage of running the app for 120 seconds without any interaction. In addition to the mean energy consumption, we compare different frameworks using the mean energy consumption without the corresponding idle cost, calculated as follows:

$$\bar{x}' = \frac{\sum_{i=1}^{N=30} (E_i - IdleCost * \Delta t_i)}{N} \quad (3.1)$$

where N is the number of times experiments are repeated (30), E_i is the measured energy consumption for execution i , $IdleCost$ is the energy usage per second (i.e., power) of having the device in idle mode, expressed in watts (W), and Δt_i the duration of execution i .

After removing idle cost, we compute overhead in a similar fashion as previous work [Abdulsalam et al., 2015]:

$$Overhead(\%) = \left(\frac{\bar{x}'}{\bar{x}_{human}} - 1 \right) \times 100 \quad (3.2)$$

In other words, overhead is the percentage change of the energy consumption of a framework when compared to the real energy consumption induced by human interaction.

We also use \bar{x}' to compute the estimated energy consumption for a single interaction (Sg) as follows:

$$Sg = \frac{\bar{x}'}{M} \quad (3.3)$$

where M is the number of times the interaction was repeated within the same execution (e.g., in *Back Button*, $M = 200$).

Experiments were executed using an *Apple iMac Mid 2011* with a 2.7GHz *Intel Core i5* processor, 8GB *DDR3* RAM, and running OS X version 10.11.6. Room temperature was controlled for 24°C (75°F). DUT was a *Nexus 5X* manufactured by *LG*, running Android version 6.0.1. All scripts, mobile app, and data are available in the *Github* repository of the project⁹, which is released under the MIT open source license.

⁹The replication package is available on *Github*: <https://github.com/luiscruz/physalia-automators> visited on June 5, 2020.

Table 3.3: Descriptive statistics of *Tap* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank	Overhead
Human	5.56	1.61	12.84	3.14	78.44	—	—
AndroidViewClient	19.71	0.21	42.10	11.75	293.86	6	274.6%
Appium	54.73	1.14	128.47	30.46	761.49	8	870.8%
Calabash	29.25	0.72	60.10	17.89	447.28	7	470.2%
Espresso	6.07	0.16	12.93	3.63	90.70	1	15.6%
Monkeyrunner	18.08	1.28	49.97	8.63	215.87	4	175.2%
PythonUiAutomator	9.15	0.54	18.93	5.57	139.32	3	77.6%
Robotium	14.59	4.00	25.63	9.74	243.57	5	210.5%
UiAutomator	7.64	0.55	17.77	4.28	107.03	2	36.5%

3.3 Results

Next, we report the results obtained in the empirical study.

3.3.1 Idle Cost

In a sample of 30 executions, the mean energy consumption of having the app open for 120 seconds without any interaction is 22.67J. This translates into a power consumption of 0.19W (in other words, the app consumes 0.19 joules per second in idle mode). This value is used in the remaining experiments to factor out idle cost from the results.

3.3.2 Tap

Table 3.3 presents results for the *Tap* interaction. Each row in the table describes a framework as a function of the mean energy consumption (\bar{x}); standard deviation of energy consumption (s); duration of each execution of the script (Δt) in seconds; the mean energy consumption without idle cost (\bar{x}' , see Eq. 3.1); the estimated energy consumption for a single interaction (S_g , see Eq. 3.3); the position in the ranking (Rank), i.e., the ordinal position when results are sorted by the average energy consumption; and the percentage overhead when compared to the same interaction when executed by a human (as in Eq. 3.2). With the exception of the results for *Human* which are placed in the first row, the table is sorted in alphabetical order for the sake of comparison with results of other interactions.

From our experiments, we conclude that *Espresso* is the most energy efficient framework for *Taps*, consuming 3.63J on average after removing idle cost, while a single *Tap* is estimated to consume 0.09J. When compared to the human interaction, *Espresso* imposes an overhead of 16%. The least efficient frameworks for a *Tap* are *Appium*, and *Calabash*, with overheads of 871% and 470%, respectively. Using these frameworks for taps can dramatically affect energy consumption results.

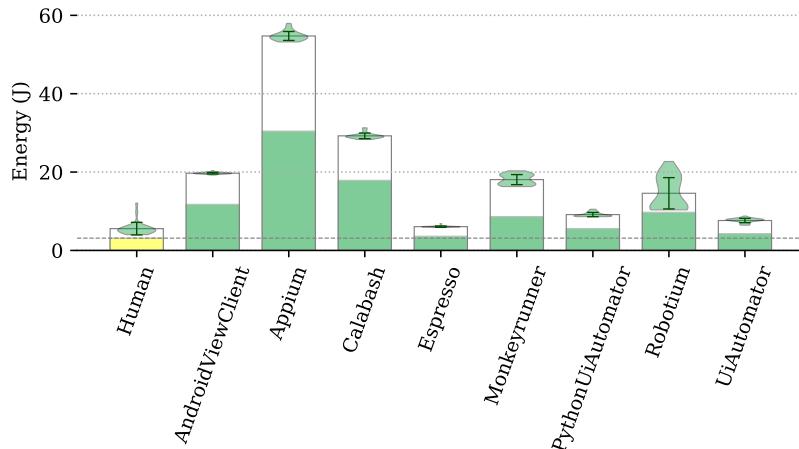


Figure 3.3: Violin plot of the results for the energy consumption of *Tap*.

A visualization of these results is in Figure 3.3. The height of each white bar shows the mean energy consumption for the framework. The height of each green or yellow bar represents the energy consumption without the idle cost. The yellow bar and the dashed horizontal line highlight the baseline energy consumption. In addition, it shows a violin plot with the probability density of data using rotated kernel density plots. The violin plots provide a visualization of the distribution, allowing to compare results regarding shape, location, and scale.

3.3.3 *Long Tap*

Results for the interaction *Long Tap* are in Table 3.4 and Figure 3.4. *Monkeyrunner* and *Espresso* are the most efficient frameworks, with overheads of 77% ($\bar{x}' = 12.60\text{J}$) and 81% ($\bar{x}' = 12.88\text{J}$), respectively. *PythonUIAutomator* and *Calabash* are the most inefficient (overhead over 300%).

A remarkable observation is the efficiency of *Appium's Long Tap* ($S_g = 0.40\text{J}$) when compared to its regular *Tap* ($S_g = 0.76\text{J}$). Common sense would let us expect *Tap* to spend less energy than *Long Tap*, but that is not the case. This happens because *Appium's* usage of *Long Tap* requires a manual instantiation of a `TouchAction` object¹⁰, while *Tap* creates it internally. Although creating such an object makes code less readable, the advantage is that it can be reused for the following interactions, making a more efficient use of resources.

3.3.4 *Drag and Drop*

Results for the interaction *Drag and Drop* are in Table 3.5 and Figure 3.5. *UIAutomator* is the best testing framework with an overhead of 185% ($\bar{x}' = 14.48\text{J}$). *Espresso*

¹⁰Appium's documentation for Touch Actions: <http://appium.io/docs/en/writing-running-appium/touch-actions/> (Visited on June 5, 2020).

Table 3.4: Descriptive statistics of *Long Tap* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank	Overhead
Human	13.33	2.21	32.86	7.12	177.92	—	—
AndroidViewClient	49.18	5.24	119.21	26.66	666.40	6	274.6%
Appium	25.34	0.86	49.60	15.96	399.08	4	124.3%
Calabash	46.96	1.81	94.27	29.14	728.57	8	309.5%
Espresso	19.87	0.54	37.00	12.88	321.94	2	80.9%
Monkeyrunner	21.68	0.74	48.07	12.60	315.04	1	77.1%
PythonUiAutomator	48.19	12.63	101.13	29.08	727.02	7	308.6%
Robotium	39.35	1.82	99.97	20.46	511.40	5	187.4%
UiAutomator	22.39	0.75	45.40	13.81	345.20	3	94.0%

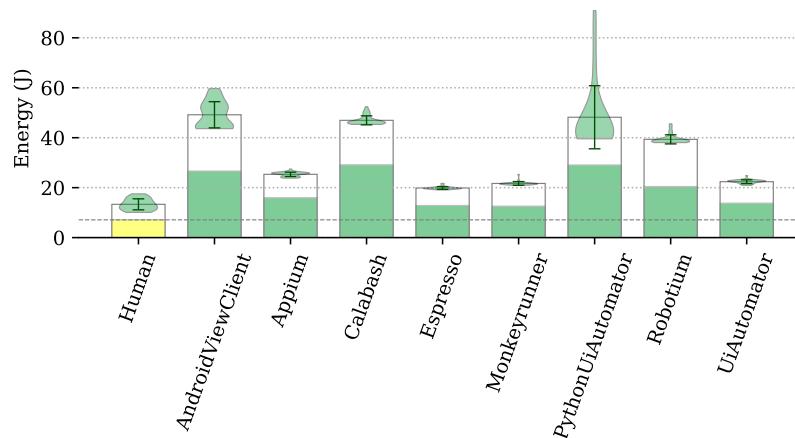


Figure 3.4: Violin plot of the results for energy consumption of *Long Tap*.

Table 3.5: Descriptive statistics of *Drag and Drop* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank	Overhead
Human	7.55	1.91	23.70	3.08	76.90	—	—
AndroidViewClient	21.31	0.76	62.15	9.57	239.24	2	211.1%
Appium	43.71	1.14	85.00	27.65	691.27	6	798.9%
Calabash	134.08	3.55	336.33	70.53	1763.27	7	2193.0%
Monkeyrunner	28.50	1.29	52.97	18.49	462.22	4	501.1%
PythonUiAutomator	36.30	3.77	93.53	18.62	465.56	5	505.4%
Robotium	20.63	1.02	52.17	10.77	269.29	3	250.2%
UiAutomator	14.48	0.63	30.27	8.76	219.02	1	184.8%

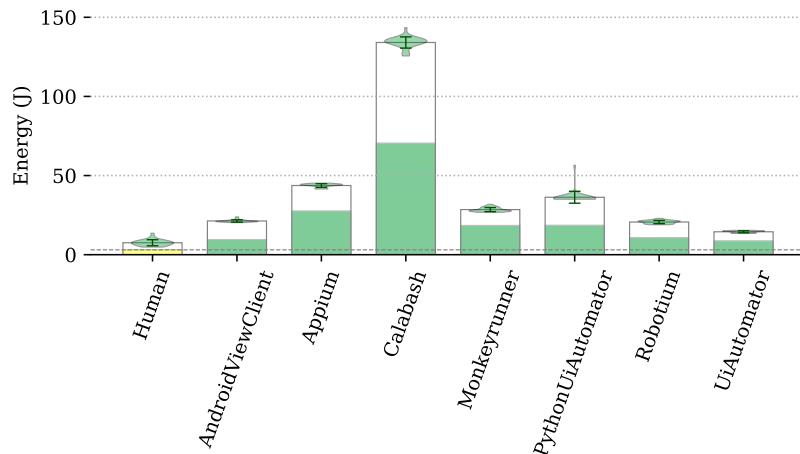


Figure 3.5: Violin plot of the results for energy consumption of *Drag and Drop*.

is not included in the experiments since *Drag and Drops* are not supported. The most energy greedy framework is *Calabash* with an overhead of 2193%. When compared to *UIAutomator*, one *Drag and Drop* with *Calabash* is equivalent to more than 11 *Drag and Drops*. Hence, *Calabash* should be avoided for energy measurements that include *Drag and Drops*.

3.3.5 *Swipe*

Results for the interaction *Swipe* are presented in Table 3.6 and Figure 3.6. *Espresso* is the best framework with an overhead of 29%, while *Robotium*, *AndroidViewClient*, *Monkeyrunner*, and *Calabash* are the most energy greedy with similar overheads, above 400%.

3.3.6 *Pinch and Spread*

Results for the interaction *Pinch and Spread* are presented in Table 3.7 and Figure 3.7. Although this interaction is widely used in mobile applications for features such as zoom in and out, only *Calabash*, *PythonUiAutomator*, and *UIAutomator* support it out of the box. *UIAutomator* is the most efficient framework, spending less energy than

Table 3.6: Descriptive statistics of *Swipe* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank	Overhead
Human	9.11	1.09	24.48	4.48	56.05	—	—
AndroidViewClient	45.29	0.62	115.87	23.39	292.41	6	421.7%
Appium	17.09	0.46	30.00	11.42	142.80	2	154.8%
Calabash	43.27	0.81	93.73	25.56	319.46	8	469.9%
Espresso	10.35	0.26	24.10	5.79	72.43	1	29.2%
Monkeyrunner	36.63	1.49	68.67	23.65	295.69	7	427.5%
PythonUiAutomator	26.42	0.91	54.60	16.11	201.32	3	259.1%
Robotium	41.30	0.67	96.00	23.16	289.46	5	416.4%
UiAutomator	27.56	0.65	60.13	16.20	202.49	4	261.2%

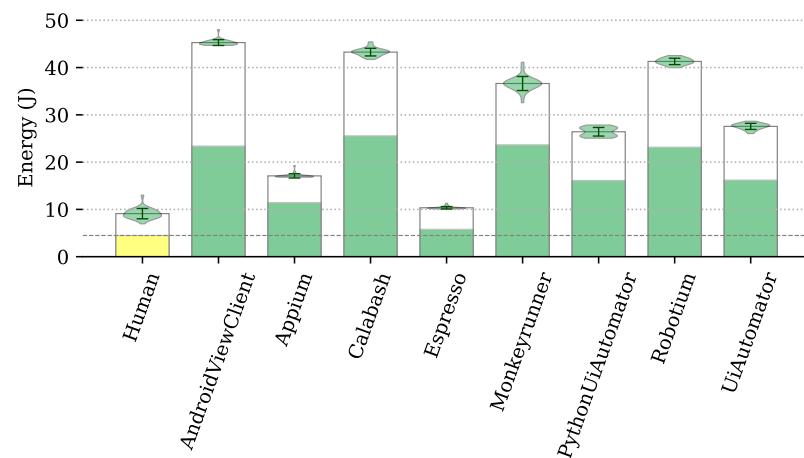


Figure 3.6: Violin plot of the results for energy consumption of *Swipe*.

Table 3.7: Descriptive statistics of *Pinch and Spread* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank	Overhead
Human	9.59	1.37	21.91	5.45	68.10	—	—
Calabash	41.31	8.66	81.93	25.83	322.83	3	374.0%
PythonUiAutomator	26.39	1.23	58.77	15.29	191.09	2	180.6%
UiAutomator	9.19	1.66	21.23	5.17	64.67	1	-5.0%

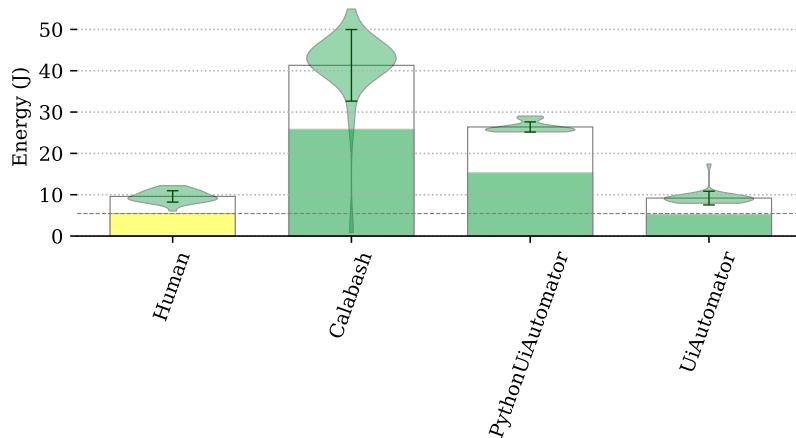


Figure 3.7: Violin plot of the results for energy consumption of *Pinch and Spread*.

the equivalent interaction performed by a human (-5%). The remaining frameworks, *PythonUiAutomator* and *Calabash* were not as efficient, providing overheads of 181% and 374%, respectively.

3.3.7 Back Button

Results for the interaction *Back Button* are presented in Table 3.8 and Figure 3.8. In this case, human interaction was considerably less efficient than most frameworks, being ranked fifth on the list. The main reason for this is that frameworks do not realistically mimic the *Back Button* interaction. When the user presses the back button, the system produces an input event and a vibration or haptic feedback simultaneously. However, frameworks simply produce the event. Thus, results are not comparable with the human interaction. Still, *AndroidViewClient* provided an overhead of 440%, being the least inefficient framework.

Another remarkable result was that *Robotium*, despite being energy efficient after removing idle cost, is the slowest framework. Thus, it is likely that *Robotium* is using a conservative approach to generate events in the device: it suspends the execution to wait for the back button event to take effect in the app.

Table 3.8: Descriptive statistics of *Back Button* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank	Overhead
Human	17.90	2.56	43.94	9.60	47.98	—	—
AndroidViewClient	85.75	2.11	179.73	51.79	258.94	8	439.7%
Appium	2.43	0.17	3.33	1.80	9.01	2	-81.2%
Calabash	30.95	0.77	80.57	15.73	78.63	7	63.9%
Espresso	8.89	0.29	35.17	2.25	11.25	4	-76.6%
Monkeyrunner	1.84	0.12	4.07	1.08	5.38	1	-88.8%
PythonUiAutomator	53.62	1.78	220.03	12.04	60.20	6	25.5%
Robotium	60.44	5.18	308.10	2.22	11.10	3	-76.9%
UiAutomator	49.87	1.03	208.20	10.53	52.64	5	9.7%

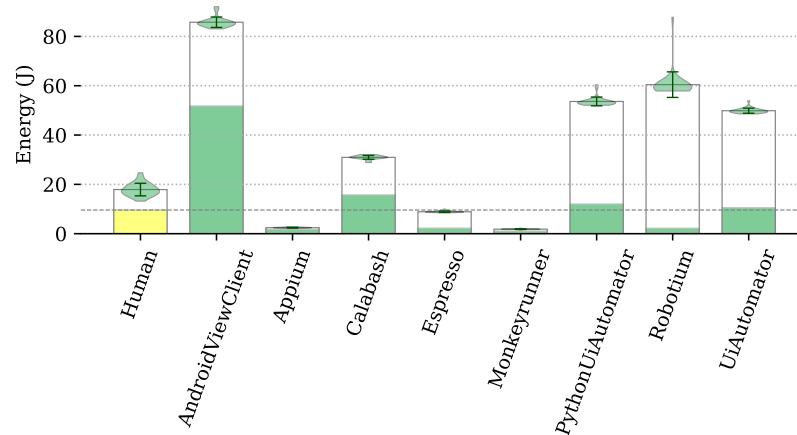


Figure 3.8: Violin plot of the results for energy consumption of *Back Button*.

Table 3.9: Descriptive statistics of *Input Text* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank	Overhead
Human	22.11	4.06	54.09	11.89	1189.37	—	—
AndroidViewClient	222.08	4.31	523.37	123.18	12318.21	8	935.7%
Appium	44.43	1.89	105.27	24.54	2453.84	7	106.3%
Calabash	27.14	1.03	62.40	15.35	1534.70	6	29.0%
Espresso	6.83	0.18	14.03	4.18	417.96	3	-64.9%
Monkeyrunner	6.18	0.29	8.03	4.67	466.58	5	-60.8%
PythonUiAutomator	9.16	4.35	25.37	4.37	436.83	4	-63.3%
Robotium	4.64	0.86	12.50	2.27	227.34	2	-80.9%
UiAutomator	2.93	1.39	8.00	1.42	142.02	1	-88.1%

3.3.8 *Input Text*

Results for the interaction *Input Text* are presented in Table 3.9 and Figure 3.9. Each iteration of *Input Text* consists in writing a pre-defined 17-character sentence in a text field and then clearing it all back to the initial state. Thus, the value for a single interaction (*Sg*) is the energy spent when this sequence of events is executed, but can hardly be extrapolated for other input interactions.

UIAutomator is the framework with the lowest energy consumption ($\bar{x}' = 1.42\text{J}$). The human interaction spends more energy than most frameworks. The reason behind this is that frameworks have a different way to deal with text input. Most frameworks generate a sequence of events that will generate the given sequence of characters. On the contrary, the human interaction resorts to the system keyboard to generate this sequence. Thus the system has to process a sequence of taps and match it to the right character event. There are even other frameworks, namely *UIAutomator*, *PythonUIAutomator*, and *Robotium*, that, as showed in the overview of Table 3.1, implement *Input Text* more artificially. Instead of generating the sequence of events, they directly change the content of the text field. This is more efficient but bypasses system and application behavior—e.g., automatic text correction features.

Results showed that the *AndroidViewClient* is very inefficient and its overhead (936%) is not negligible when measuring the energy consumption of mobile apps.

3.3.9 *Find by id*

Results for the task *Find by id* are presented in Table 3.10 and Figure 3.10. *Find by id* is a method that looks up for a UI component that has the given *id*. It does not mimic any user interaction, but it is necessary to create interaction scripts. Methods *Find by description* and *Find by content* are used to achieve the same objective. For this reason, we do not report the consumption of a human interaction in these cases.

For the sake of consistency with previous cases, we report tables and figures in the same fashion. However, we consider that the overall cost of energy consumption (without removing idle cost) should not be discarded in this case. These methods are

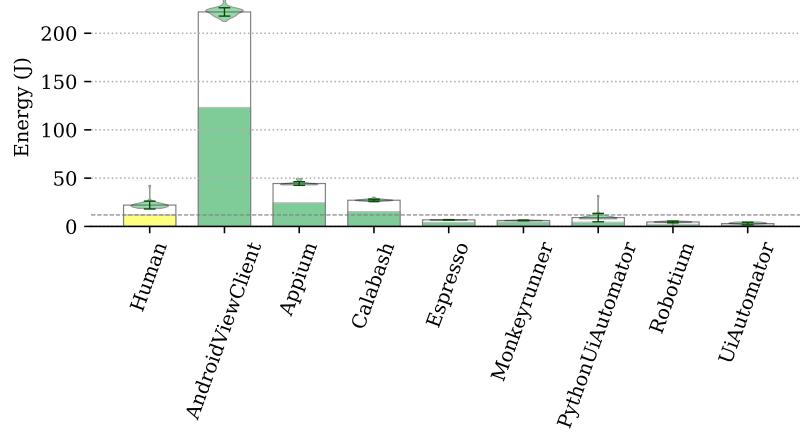


Figure 3.9: Violin plot of the results for energy consumption of *Input Text*.

Table 3.10: Descriptive statistics of *Find by id* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank
AndroidViewClient	37.52	1.64	129.91	12.97	46.34	7
Appium	5.94	0.51	12.73	3.53	12.62	5
Calabash	41.20	2.08	89.63	24.26	86.65	8
Espresso	1.37	0.11	2.03	0.99	3.54	2
Monkeyrunner	2.74	0.70	6.13	1.58	5.66	3
PythonUiAutomator	8.42	4.16	19.63	4.71	16.81	6
Robotium	27.97	0.46	143.03	0.94	3.37	1
UiAutomator	5.26	0.84	14.33	2.55	9.11	4

not required in manual interactions. Hence, we consider that a UI testing frameworks that takes longer to execute these methods should not benefit from having its idle cost removed. This approach is supported by our results, as we show below.

Robotium is the most energy-efficient, with an energy consumption without idle cost of $0.94J$. However, if we consider idle cost, *Robotium* is amongst the most energy greedy frameworks (after *Calabash* and *AndroidViewClient*). It has an overall energy consumption of $27.97J$. When considering idle cost, *Espresso* is the most energy-efficient framework.

This difference lies in the mechanism adopted by frameworks to deal with UI changes. After user interaction, the UI is expected to change and the status of the UI can become obsolete. Thus, frameworks need to wait until the changes the UI are complete. Results show that *Robotium* uses a mechanism based on suspending the execution to make sure the UI is up to date. On the other hand, *Espresso* uses a different heuristic, which despite spending more energy on computation tasks, it does not require the device to spend energy while waiting.

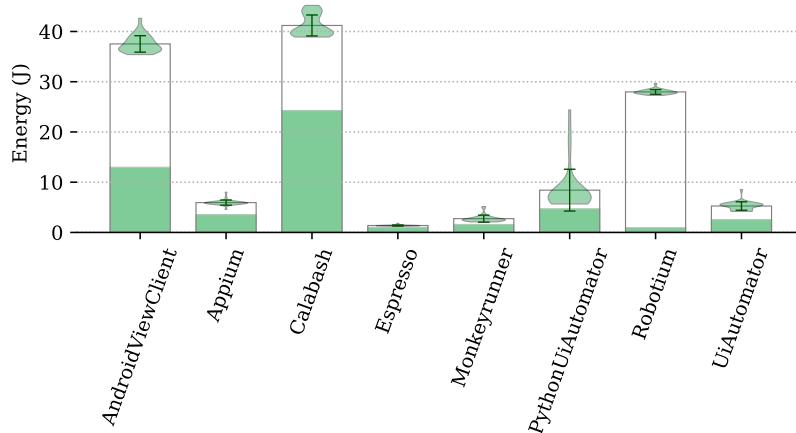


Figure 3.10: Violin plot of the results for energy consumption of *Find by id*.

Table 3.11: Descriptive statistics of *Find by description* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank
AndroidViewClient	36.85	0.78	127.45	12.77	45.59	5
Appium	6.41	0.58	13.93	3.77	13.48	4
Calabash	41.41	7.02	88.20	24.75	88.38	6
Espresso	1.37	0.10	2.10	0.97	3.46	1
PythonUiAutomator	6.62	0.49	15.10	3.76	13.44	3
UiAutomator	5.13	0.61	14.47	2.40	8.57	2

3.3.10 *Find by description*

Results for *Find by description* are presented in Table 3.11 and Figure 3.11. *Find by description* and *Find by id* are very similar regarding usage and implementation, which is confirmed by results. *Espresso* is the best framework regardless of idle cost ($\bar{x} = 1.37\text{J}$ and $\bar{x}' = 0.97\text{J}$). *Android View Client* and *Calabash* are distinctly inefficient. All other frameworks show reasonable energy footprints, except for *Robotium* and *Monkeyrunner*, which were not included since *Find by description* is not supported.

3.3.11 *Find by content*

Results for *Find by content* are presented in Table 3.12 and Figure 3.12. After removing idle cost, *Robotium* is the framework with best results ($\bar{x}' = 0.14\text{J}$). However, in resemblance to *Find by id*, *Robotium* is very inefficient when idle cost is not factored out ($\bar{x} = 23.74\text{J}$). In this case, *Appium* is the most efficient framework ($\bar{x} = 3.07\text{J}$).

Unlike with *Find by id* and *Find by description*, *Espresso* did not yield good results in this case ($\bar{x} = 9.43\text{J}$ and $\bar{x}' = 6.19\text{J}$). This is explained by the fact that *Espresso* runs natively on the DUT. Thus, finding a UI component by content entails additional

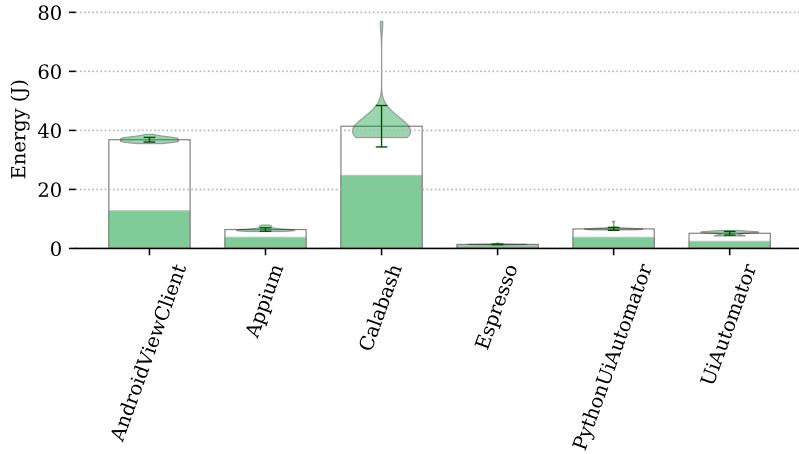


Figure 3.11: Violin plot of the results for energy consumption of *Find by description*.

Table 3.12: Descriptive statistics of *Find by content* interaction.

	\bar{x} (J)	s	Δt (s)	\bar{x}' (J)	Sg (mJ)	Rank
AndroidViewClient	36.89	1.65	127.62	12.77	106.43	6
Appium	3.07	0.31	6.07	1.92	16.02	4
Calabash	31.77	4.64	79.63	16.72	139.35	7
Espresso	9.43	0.99	17.13	6.19	51.58	5
PythonUiAutomator	3.10	0.19	6.90	1.79	14.93	3
Robotium	23.74	0.48	124.90	0.14	1.15	1
UiAutomator	3.50	0.62	9.40	1.72	14.37	2

processing: the DUT has to search for a pattern in all components' text content. Since remote script-based frameworks, such as *Appium*, can do such task using the controller workstation, they can be more energy-efficient from the DUT's perspective. For the same reason, *Find by content* has consistently higher energy usage than the other helper methods.

3.3.12 Statistical significance

Statistical significance of the mean difference of energy consumption between frameworks was assessed using the parametric Welch's t-test as used in previous work [Cruz and Abreu, 2017]. All but a few tests (2 out 105) resulted in a small p -value, below the significance level $\alpha = 0.05$. For those pairs where there was no statistical significance, we could not find any meaningful finding. Given the myriad number of tests performed, results are not presented. Violin plots corroborate statistical significance by presenting very distinct distributions among all different frameworks. For further details, all results and data are publicly available¹¹.

¹¹The replication package is available on *Github*: <https://github.com/luiscruz/physalia-automators> visited on June 5, 2020.

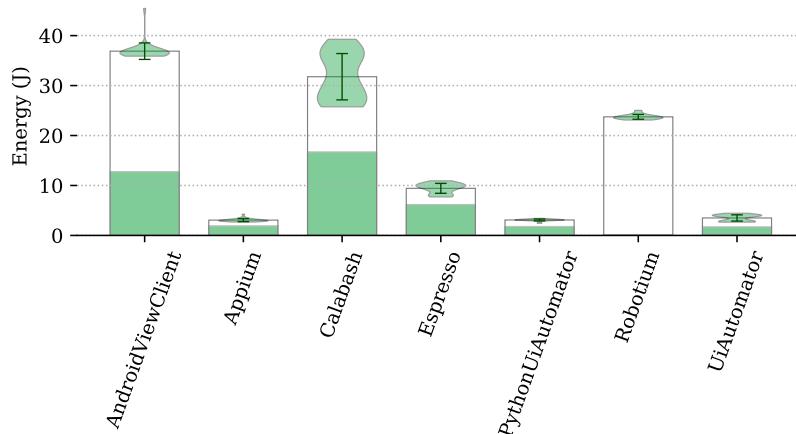


Figure 3.12: Violin plot of the results for energy consumption of *Find by content*.

3.4 Discussion

By answering the research questions, in this section we discuss our findings from the empirical evaluation, as well as outline their practical implications.

Research Question 3.1

Does the energy consumption overhead created by UI testing frameworks affect the results of the energy efficiency of mobile applications?

Yes, results show that interactions can have a tremendous overhead on energy consumption when an inefficient UI testing framework is used.

According to previous work, executing a real app during 100s yields an energy consumption of 58J, on average [D. Li et al., 2014a]. Considering our results, executing a single interaction such as *Drag and Drop* can increase energy consumption in 1.7J (overhead of 3% in this case). However, given that mobile apps are very reactive to user input [Joorabchi et al., 2013], in 100 seconds of execution, more interactions are expected to affect energy. Although a fair comparison must control for different devices and OS versions, this order of magnitude implies that overheads are not negligible. Thus, choosing an efficient UI testing framework is quintessential for energy tests.

Since all frameworks produce the same effect in the UI, the overhead of energy consumption is created by implementation decisions of the framework and not by the interaction itself. The main goal of a UI testing framework is to mimic realistic usage scenarios, but interactions with such overhead can be considered unrealistic.

One practical implication of the results in this work is to drive a change in the mindset of tool developers, bringing awareness of the energy consumption of their

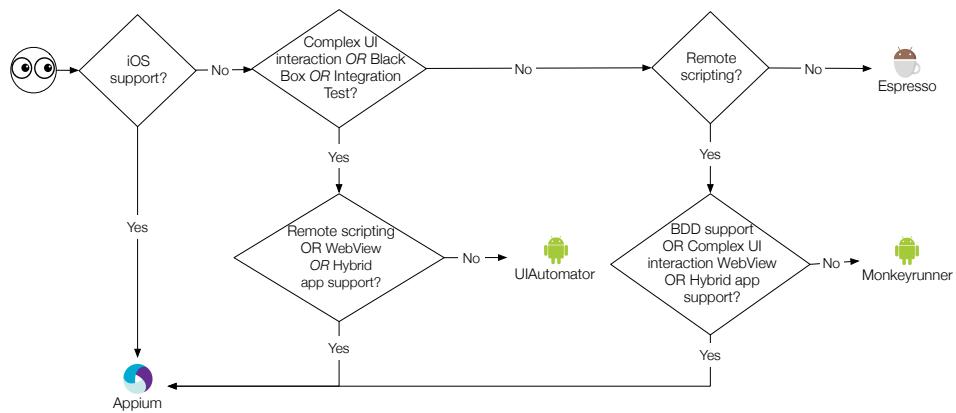


Figure 3.13: Selecting the most suitable framework for energy measurements.

frameworks. Thus, we expect future releases of UI testing frameworks to become more energy-efficient.

AndroidViewClient and *Calabash* consistently showed poor energy efficiency among all interactions. Despite providing a useful and complete toolset for mobile software developers, they should be used with prudence while testing the energy consumption of an app that heavily relies on user interactions. The work of Carette A. *et al.* (2017) [Carette et al., 2017] was affected by a poor choice of framework: the authors used *Calabash* to mimic between 136 and 325 user interactions per experiment. Our work shows that results would be different if the overhead of the framework had been factored out. *Calabash* was also used in other work [Cao et al., 2017] but, on the contrary, its impact can be considered insignificant since the experiments did not require much interaction, and the main source of energy consumption came from Web page loads. In any case, we consider that using a more energy efficient framework could corroborate the evidence or find new—even contradictory—conclusions.

Research Question 3.2

What is the most suitable framework to profile energy consumption?

Choosing the right framework for a project can be challenging: there is no *one solution fits all*. Based on our observations, Figure 3.13 depicts a decision tree to help software developers making an educated guess about the most suited and energy-efficient framework, given the idiosyncrasies of an app (that may restrict the usage of a framework). For example, if the project to be tested requires *WebView* support, one should use *Appium* rather than the other frameworks. *Robotium* is also an option if the app requires *Taps* or *Input Text* only, and neither iOS support nor remote script-based is required.

Remote script-based frameworks allow developers to easily create automation scripts. The script can be iteratively created using a console while interactions take effect

on the phone in real time. From our experience while doing this work, remote script-based frameworks are easier to use and set up (i.e., gradual learning curve). This is one of the reasons many frameworks decided to use scripting languages (e.g., Python and Ruby) instead of the official languages for Android, *Java* or *Kotlin*. Notwithstanding, remote script-based frameworks require an active connection with the phone during measurements, which leads to higher energy consumption (as is confirmed by results). Each step of the interaction requires communication with the DUT; hence, the communication logic unavoidably increases the energy consumption. On the contrary, other frameworks can transfer the interaction script in advance to the mobile phone and run it natively on the phone, which is more energy efficient.

There are, however, two scenarios where remote script-based frameworks exhibit the best results: *Back Button* with *Monkeyrunner* (see Table 3.8), and *Find By Content* with *Appium* (see Table 3.12). This is an interesting finding as it shows that remote script-based frameworks can also be developed in an energy efficient way. As such, this evidence shows that there is room for energy optimization in the other frameworks.

In addition, USB communication is out of question for remote script-based frameworks since it affects the reliability of measurements. Frameworks that do not support remote scripting can be used with USB connection if unplugged during measurements (using tools such as *Monsoon Power Monitor*).

Among remote script-based frameworks, *Monkeyrunner* is the most energy-efficient framework. The only problem is that it does not support many of the studied interactions. These results show that if energy consumption turns into a priority, it is possible to make complex frameworks such as *Appium* more energy efficient.

Research Question 3.3

Are there any best practices when it comes to creating automated scripts for energy efficiency tests?

One thing that stands out is the fact that looking up one UI component is expensive. This task is exclusively required for automation and does not reflect any real-world interaction. Taking the example of *Espresso*: a single *Tap* consumes 0.09J, while using content to look up a component consumes 0.05J. Since a common *Tap* interaction requires looking a component up, 36% of energy spent is on that task.

Looking up UI components is energy greedy because the framework needs to process the UI hierarchy find a component that matches a given id, description, or content. Since the app we use has a very simple UI hierarchy, the energy consumption is likely to be higher in real apps. Hence, using lookup methods should be avoided whenever possible. A naive solution could be using the pixel position of UI components instead

of identifiers. Pixel positions could be collected using a recorder. However, this is a bad practice since it brings major maintainability issues across different releases and device models. For that reason, state-of-the-art UI recorders used by Android developers, such as *Robotium Recorder*, yield scripts based on UI identifiers. As an alternative, we recommend caching the results of lookup calls whenever possible.

In addition, lookup methods *Find by Id* and *Find by Description* should be preferred to *Find by Content*. Results consistently show worse energy efficiency when using *Find by Content*. In *Espresso*, this difference gives an increase in energy consumption from 1.4J to 9.4J (overhead of 600%).

3.5 Threats to validity

Construct validity Frameworks rely on different approaches to collect information about the UI components that are visible on the screen. The app used in the experiments has a UI that remains unchanged upon user interactions. In a real scenario, however, the UI typically reacts to user interactions. Frameworks that have an inefficient way of updating their UI model of components visible in the screen, may entail a high overhead on energy consumption. However, as manually triggering this update is not supported in most frameworks, it was unfeasible to include it in our study.

In addition, the overheads are calculated based on the results collected from the human interaction from two participants. Although results showed a small variance between different participants, the energy consumption may vary with other humans. Nevertheless, differences are not expected to be significant, and results still apply.

Moreover, energy consumption for a single interaction is inferred by the total consumption of a sequence of interactions. Potential *tail energy consumptions*¹² of a single interaction are not being measured. This is mitigated by running multiple times the same interaction.

Internal validity The Android OS is continuously running parallel tasks that affect energy consumption. For that reason, system settings were customized as described in Section 3.2 (e.g., disabled automated brightness and notifications). Also, each experiment is executed 30 times to ensure statistical significance as recommended in related work [Linares-Vásquez et al., 2014].

UI interactions typically trigger internal tasks in the mobile application running in foreground. The mobile application used in experiments was developed to prevent any side-effects to UI events. To ensure that scripts are interacting with the device as expected, the application was set to a mode that is not affected by user interaction.

¹²Tail energy is the energy spent during initialization or closure of a resource.

Thus, the behavior is equal across different UI testing frameworks and experiments only measure their energy consumption.

External validity Energy consumption results vary upon different versions of Android OS, different device models, and different framework version. However, unless major changes are released, results are not expected to significantly deviate from the reported ones. Note that testing different devices requires disassembling them and making them useless for other purposes (that is to say that empirical studies as the one conducted by us are expensive), which can be economically unfeasible. Regardless, all the source code used in experiments will be released as Open Source to foster reproducibility.

3.6 Related Work

In this section we summarize related work on testing frameworks for mobile app testing and we differentiate our work regarding goals and methodology.

UI testing frameworks play an important role on the research of mobile software energy efficiency. They are used as part of the experimental setup for the validation of approaches for energy efficiency of mobile apps. *Monkeyrunner* has been used to assess the energy efficiency of Android's API usage patterns [Linares-Vásquez et al., 2014]. It was found that UI manipulation tasks (e.g., method `findViewById`) and database operations are expensive in terms of energy consumption. These findings suggest that UI testing frameworks might as well create a considerable overhead on energy consumption. *Monkeyrunner* has also been used to assess benefits in energy efficiency on the usage of Progressive Web Apps technology in mobile web apps [Malavolta et al., 2017], despite the fact that no statistically significant differences were found. *Android View Client* has been used to assess energy efficiency improvements of performance based optimizations for Android applications [Cruz and Abreu, 2017; Cruz and Abreu, 2018b], being able to improve energy consumption up to 5% in real, mature Android applications. Other works have also used *Robotium* [Hecht et al., 2016], *Calabash* [Cao et al., 2017; Carette et al., 2017], and *RERAN* [Gomez et al., 2013; Sahin et al., 2016]. Our work uses a similar approach for assessing and validating energy efficiency, but it has distinct goals as we focus on the impact of UI testing frameworks on energy efficiency results.

Previous work studied five Android testing frameworks in terms of fragilities induced by maintainability [Coppola et al., 2016; Coppola, 2017]. Five possible threats that could break tests were identified: 1) identifier change, 2) text change, 3) deletion or relocation of UI elements, 4) deprecated use of physical buttons, and 5) graphics change (mainly for image recognition testing techniques). These threats are aligned

with efforts from existing works [Z. Gao et al., 2016]. Our work differentiates itself by focusing on the energy efficiency of Android testing tools.

In a study comparing *Appium*, *MonkeyTalk*, *Ranorex*, *Robotium*, and *UIAutomator*, *Robotium* and *MonkeyTalk* stood out as being the best frameworks for being easy to learn and providing a more efficient comparison output between expected and actual result [Gunasekaran and Bargavi, 2015]. A similar approach was taken in other works [Kulkarni and Soumya, 2016; K.-C. Liu et al., 2017] but although they provide useful insights about architecture and feature set, no systematic comparison was conducted. We compare different frameworks with a quantitative approach to prevent bias of results.

Linares-Vásquez M. et al. (2017) have studied the current state-of-the-art in terms of the frameworks, tools, and services available to aid developers in mobile testing [Linares-Vásquez et al., 2017c]. It focused on 1) Automation APIs/Frameworks, 2) Record and Replay Tools, 3) Automated Test Input Generation Techniques, 4) Bug and Error Reporting/Monitoring Tools, 5) Mobile Testing Services, and 6) Device Streaming Tools. It envisions that automated testing tools mobile apps should address development restrictions: 1) restricted time/budget for testing, 2) needs for diverse types of testing (e.g., energy), and 3) pressure from users for continuous delivery. In a similar work, these issues were addressed by surveying 102 developers of Android open source projects [Linares-Vásquez et al., 2017b]. This work identified a need for automatically generated test cases that can be easily maintained over time, low-overhead tools that can be integrated with the development workflow, and expressive test cases. Our work differs from these studies by providing an empirical comparison solely on UI testing frameworks, and addressing energy tests.

Choudhary R., et al. (2015) compared **Automated Input Generation (AIG)** techniques using four metrics [Choudhary et al., 2015]: ease of use, ability to work on multiple platforms, code coverage, and ability to detect faults. It was found that random exploration strategies by *Monkey*¹³ or *Dynodroid* [Machiry et al., 2013] were more effective than more sophisticated approaches. Although our work does not scope AIG tools, very often they use UI testing frameworks (e.g., *UIAutomator* and *Robotium*) underneath their systems [Linares-Vásquez, 2015; Hao et al., 2014; Mahmood et al., 2014; C.-H. Liu et al., 2014]. Results and insights about energy consumption in our study may also apply to tools that build on top of UI testing frameworks.

¹³*UI/Application Exerciser Monkey* also known as *Monkey* tool: <https://developer.android.com/studio/test/monkey.html> (visited on June 5, 2020).

3.7 Summary

In this chapter, we have analyzed eight popular UI testing frameworks for mobile apps with respect to their energy footprint. This analysis is motivated by the fact that UI frameworks are quintessential to reliably test energy efficiency in mobile applications. Concretely, in this chapter:

- We design a methodology to measure the energy consumption of eight UI testing frameworks within six typical interactions and three helper methods (cf. Section 3.2).
- We show that the energy consumption of UI testing frameworks can affect results of energy tests. As an example, we have observed the overhead of the *Drag and Drop* interaction to go up to 2200% (cf. RQ 3.1).
- *Espresso* is observed to be the most energy-efficient framework (cf. RQ 3.2). However, depending on the needs of the tester, *Appium*, *Monkeyrunner*, or *UIAutomator* are good alternatives. We propose a decision tree (cf. Figure 3.13) to help in the decision-making process.
- We show that helper methods to find components in the interface should be minimized to prevent affecting energy results. In particular, lookup methods based on the content of the UI component need to be avoided. They consistently yield poor energy efficiency when compared to lookups based on id (e.g., in *Espresso* it creates an overhead of 600%) (cf. RQ 3.3).

Prevalence of Test Automation in Android Apps



To the attention of mobile software developers: guess what, test your app!

Luis Cruz, Rui Abreu, and David Lo

In: *Empirical Software Engineering*, 2019.

Abstract

Software testing is an important phase in the software development lifecycle because it helps in identifying bugs in a software system before it is shipped into the hand of its end users. There are numerous studies on how developers test general-purpose software applications. The idiosyncrasies of mobile software applications, however, set mobile apps apart from general-purpose systems (e.g., desktop, stand-alone applications, web services). In this chapter, we investigate working habits and challenges of mobile software developers with respect to testing. A key finding of our exhaustive study, using 1000 Android apps, demonstrates that mobile apps are still tested in a very ad hoc way, if tested at all. However, we show that, as in other types of software, testing increases the quality of apps (demonstrated in user ratings and number of code issues). Furthermore, we find evidence that tests are essential when it comes to engaging the community to contribute to mobile open source software. We discuss reasons and potential directions to address our findings. Yet another relevant finding of our study is that Continuous Integration and Continuous Deployment (CI/CD) pipelines are rare in the mobile apps world (only 26% of the apps are developed in projects employing CI/CD) – we argue that one of the main reasons is due to the lack of exhaustive and automatic testing.

4.1 Introduction

Mobile app developers can resort to several tools, frameworks and services to develop and ensure the quality of their apps [Linares-Vásquez et al., 2017c]. However, it is still a fact that errors creep into deployed software, which may significantly decrease the reputation of developers and companies alike. Software testing is an important phase in the software development lifecycle because it helps in identifying bugs in the software system before it is shipped into the hand of end users. There are numerous studies on how developers test general-purpose software applications. The idiosyncrasies of mobile software apps, however, set mobile apps apart from

general-purpose systems (e.g., desktop, stand-alone applications, web services) [Hu and Neamtiu, 2011; Picco et al., 2014].

Therefore, the onset of mobile apps came with a new ecosystem where traditional testing tools do not always apply [Moran et al., 2017; Wang and Alshboul, 2015; Maji et al., 2010]: complex user interactions (e.g., swipe, pinch, etc.) need to be supported [Zaeem et al., 2014]; apps have to account for devices with limited resources (e.g., limited power source, lower processing capability); developers have to factor in an ever-increasing number of devices as well as OS versions [Khalid et al., 2014]; apps typically follow a weekly/bi-weekly time-based release strategy which creates critical time constraints in testing tasks [Nayebi et al., 2016]. Moreover, manual testing is not a cost-effective approach to ensure software quality and ought to be replaced by automated techniques [Muccini et al., 2012].

This work studies the adoption of automated testing by the Android open source community. We use the term “automated testing” as a synonym of “test automation”: the process in which testers write code/test scripts to automate test execution. AIG techniques were not considered in this study. We analyze the adoption of unit tests, UI tests, cloud based testing services, and CI/CD. Previous work, in a survey with 83 Android developers, suggests that mobile developers are failing to adopt automated testing techniques [Kochhar et al., 2015]. This is concerning since testing is an important factor in software maintainability [Visser et al., 2016]. We investigate this evidence by systematically checking the codebase of 1000 Android projects released as FOSS. Moreover, we delve into a broader set of testing technologies and analyze the potential impact they can have in different aspects of the mobile apps (e.g., popularity, issues, etc.).

As in related studies [Krutz et al., 2015], we opted to use open source mobile applications due to the availability of the data needed for our analysis. In particular, our work answers the following research questions:

Research Question 4.1

What is the prevalence of automated testing technologies in the FOSS mobile app development community?

Why and How: It is widely accepted that tests play an important role in ensuring the quality of software code. However, the extent to which tests are being adopted amongst the Android FOSS community is still not known. We want to assess whether developers have been able to integrate tests in their projects and which technologies have gained their acceptance. We do that by developing a static analysis tool that collects data from an Android project regarding its usage of test automation technologies. We apply the tool to our dataset of 1000 apps and analyze the pervasion of the different technologies.

Main findings: FOSS mobile apps are still tested in a very *ad hoc* way, if tested at all. Testing technologies were absent in almost 60% of projects in this study. *JUnit* and *Espresso* were the most popular technologies in their category with an adoption of 36% and 15%, respectively. Novel testing and development techniques for mobile apps should provide a simple integration with these two technologies to prevent incompatibility issues and promote test code reuse.

Research Question 4.2

Are today's mature FOSS Android apps using more automated testing than yesterday's?

Why and How: We want to understand how the community of Android developers and researchers is changing in terms of adoption of automated testing. In this study, we compare the pervasion of automated tests in FOSS Android apps across different years.

Main findings: Automated testing has become more popular in recent years. The trend shows that developers are becoming more aware of the importance of automated testing. This is particularly evident in unit testing, but UI testing also shows a promising gain in popularity.

Research Question 4.3

How does automated testing relates to popularity metrics in FOSS Android apps?

Why and How: One of the goals of mobile developers is to increase the popularity of their apps. Although many different things can affect the popularity of apps, we study how it can be related to automated tests. We run hypothesis tests over five popularity metrics to assess significant differences between projects with and without tests.

Main findings: Tests are essential when it comes to engaging the community to contribute to mobile open source software. We found that projects using automated testing also reveal a higher number of contributors and commits. The number of *Github Forks*, *Github Stars*, and ratings from *Google Play* users does not reveal any significant impact.

Research Question 4.4

How does automated testing affect code issues in FOSS Android apps?

Why and How: The collection of code issues helps developers assess whether their code follows good design architecture principles. It can help developers avoid potential bugs, performance issues, or security vulnerabilities in their software.

We use the static analysis tool *Sonar* to collect code issues in our dataset of FOSS Android apps and study whether automated testing brings significant differences.

Main findings: Automated testing is important to ensure the quality of software. This is also evident in terms of code issues. Projects without tests have a significantly higher number of minor code issues.

Research Question 4.5

What is the relationship between the adoption of CI/CD and automated testing?

Why and How: Previous work showed the adoption of CI/CD with automated testing has beneficial results in software projects [Hilton et al., 2016; Zhao et al., 2017]. For that reason, the adoption of CI/CD is getting momentum in software projects. We want to study whether CI/CD technologies have been able to successfully address the FOSS Android and whether developers are getting the most out of CI/CD in their projects. We use static analysis to collect data regarding the adoption of CI/CD technologies and compare it to the adoption of automated testing. In addition, we discuss how numbers differ from desktop software.

Main findings: CI/CD adoption in open source mobile app development is not as predominant as in other platforms — only 26% of apps are using it in their development process. We argue that one of the main reasons is the lack of exhaustive and automatic testing — results show evidence that open source projects with CI/CD are more likely to automate tests.

In sum, our work makes the following contributions:

- We created a publicly available dataset with open source apps. The dataset was built by combining data from multiple sources, including metrics of source code quality, popularity, testing tools usage, and CI/CD services adoption. Dataset is available here: https://github.com/luiscruz/android_test_inspector.
- We have studied the trends of the adoption of testing techniques in the Android developer community and identified a set of apps that use automated tests in their development cycle.
- We have developed a tool for static detection of usage of state-of-art testing frameworks. Available here: https://github.com/luiscruz/android_test_inspector.
- We have investigated the relationship of automated test adoption with quality and popularity metrics for Android apps.
- We have investigated the relationship between automated tests and CI/CD adoption.

- We deliver a list of 54 apps that comply with testing best practices.

The remainder of this chapter is organized as follows. Related work is discussed in Section 4.2. Section 4.3 outlines the methodology used to collect data in our study. Following, Sections 4.4 to 4.8 describe our methodology and present and discuss the results for each proposed research question. In Section 4.9, we present a Hall of Fame with apps that comply with the criteria of testing best practices. Threats to the validity are discussed in Section 4.10. Finally, a summary of the contributions in this chapter is presented in Section 4.11.

4.2 Related Work

Studies based on data collected from app stores have become a powerful source of information with a direct impact on mobile software teams [Martin et al., 2017]. More works have contributed with datasets of open source Android apps [Geiger et al., 2018; Pascarella et al., 2018; Das et al., 2016]. Our work releases a dataset that differentiates by containing information regarding testing practices in Android projects.

Previous work collected 627 apps from *F-Droid* to study the testing culture of app developers [Kochhar et al., 2015]. It was found that at the time of the analysis (2015) only 14% of apps contained test cases and that only 41 apps had runnable test cases from which only 4 had line coverage above 40%. In addition, the authors conducted a survey on 83 Android app developers and 127 Windows app developers to understand the common testing tools and the main challenges faced during testing. The most used framework was *JUnit*, being used by 18 Android developers, followed by *Monkeyrunner* and *Espresso*, with 8 and 7 developers, respectively. According to developers in the survey, the main challenges while testing are time constraints, compatibility issues, lack of exposure, cumbersome tools, emphasis on development, lack of organization support, unclear benefits, poor documentation, lack of experience, and steep learning curve. Our work extends and completes the study by Kochhar et al. via a more extensive data sample (1000 Android apps) and additional analyses. We adopt a comprehensive mining-software-repositories-cum-static-analysis approach to collect mobile software code repositories and empirically assess the benefits of having tests, rather than surveying developers. In addition, we compare the presence of tests in the project with potential issues of the app, satisfaction level of end users, among other popularity metrics. Moreover, we assess the use of different testing tools using static analysis and provide insights into observed trends on automated testing in the past years and compare the testing culture with the adoption of CI/CD.

More works have attempted to capture the current picture of app testing. Silva et al. have studied 25 open source Android apps in terms of test frameworks being used

and how developers are addressing mobile-specific challenges [Silva et al., 2016]. Results show that apps are not being properly tested, and tests for app executions under limited resource constraints are practically absent. It suggests that a lack of effective tools is one of the reasons for this phenomena. Our work differentiates itself by considering a more representative sample of apps and complements Silva et al. by providing insights on how developers and researchers can help bring new types of tests into the app development community.

Coppola et al. studied the fragility of UI testing in Android apps [Coppola et al., 2017]. The authors collected 18,930 open source apps available on Github and analyzed the prevalence of five scripted UI testing technologies. However, toy apps or forks of real apps were not factored out from the sample — we understand that real apps were underrepresented [Cosentino et al., 2016; Bird et al., 2009]. Thus, we restrict our study to apps that were published in *F-Droid*. In addition, we extend our study to a broader set of testing technologies, while studying relationships between automated testing and other metrics of a project.

Corral and Fronza have compared the success of apps with quality code metrics [Corral and Fronza, 2015]. They analyzed a sample of 100 apps and consider a number of code metrics: *Weighted Methods per Class*, *Depth of Inheritance Tree*, *Number of Children*, *Response for a Class*, *Coupling between Objects*, *Lack of Cohesion in Methods*, *Cyclomatic Complexity*, and *Logical Lines of Code*. Results demonstrated that these metrics only have a marginal impact on the success of the apps, showing that real drivers of user satisfaction are beyond source code attributes. Given that mobile apps are very different from traditional applications we find the above metrics too generic. We extend Corral and Fronza’s work by focusing on the impact of test automation. Furthermore, besides user satisfaction, we also analyze a number of code issues detected using static analysis and popularity metrics important for the survival of an open source project (e.g., number of contributors).

Previous work has studied the state-of-the-art tools, frameworks, and services for automated testing of mobile apps [Linares-Vásquez et al., 2017c]. It revealed that automated test tools should aid developers to overcome the following challenges: 1) restricted time/budget for testing, 2) needs for diverse types of testing (e.g., energy), and 3) pressure from users for continuous delivery. Related work surveyed developers of open source apps to understand their main barriers to mobile testing [Linares-Vásquez et al., 2017b]. Developers identified easy maintenance, low overhead in the development cycle, and expressiveness of test cases as important aspects that should be addressed by existing frameworks.

Previous work has compared different techniques and tools for AIG [Choudhary et al., 2015; Amalfitano et al., 2017; Zeng et al., 2016]. Choudhary et al. have compared AIG testing tools in terms of ease of use, ability to work on multiple platforms, code coverage, and ability to detect faults [Choudhary et al., 2015]. A

follow-up study showed that AIG techniques are not ready yet for an industrial setting since activity coverage is dramatically low [Zeng et al., 2016]. Our work does not scope AIG techniques — we focus on automated testing strategies that require the creation of test cases. In addition, we differ by studying the prevalence of testing tools and which test frameworks have actually gained the acceptance of mobile developers.

Other works have empirically studied tests on open source software. Kochhar et al. studied the correlation between the presence of test cases and project development characteristics [Kochhar et al., 2013a; Kochhar et al., 2013b]. It was found that tests increase the number of lines of code and the size of development teams. Our work adds value to these contributions by providing insights in the context of mobile app development, and by analyzing a broader set of metrics to study the potential benefits of automated tests in mobile app development.

Hilton et al. analyzed 34,000 open source projects on *GitHub* and surveyed 442 developers [Hilton et al., 2016] on the implications of adopting CI/CD in open source software. Results showed that most popular projects are using CI/CD and its adoption is continuously increasing. A similar approach showed that developers are improving automated tests after the adoption CI/CD [Zhao et al., 2017]. Our work only focuses on the relation between automated tests and CI/CD in the context of mobile development, bringing some enlightenment on how the adoption of CI/CD differs in mobile app development.

4.3 Data collection

Data was gathered from multiple sources, as presented in Figure 4.1. *F-Droid*, a catalog that lists 2,800 free and open source Android apps¹, is used to obtain metadata, package name, and source code repository. *GitHub* is used to collect activity and popularity metrics about the development of the app: number of stars, number of contributors, number of commits, and number of forks. Other popularity metrics are also gathered from *Google Play Store*: rating, and the number of users who rated the app. Test coverage information is obtained from the cloud services *Coveralls* and *Codecov*.

We extended the data by running the static analysis tool *Sonar*² to collect quality-related metrics and potential bugs. We select *Sonar* because it integrates the results of the state-of-the-art analysis tools *FindBugs*, *Checkstyle*, and *PMD*. Furthermore, it has been used with the same purpose in previous work [Krutz et al., 2015].

¹F-Droid's website: <https://f-droid.org> (Visited on June 5, 2020).

²Sonar's website: <https://www.sonarqube.org> (Visited on June 5, 2020).

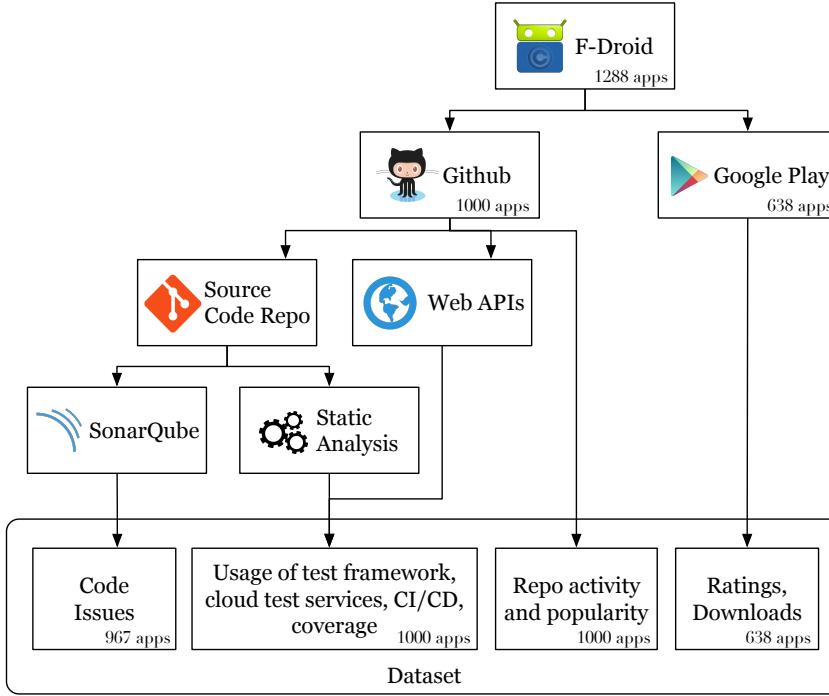


Figure 4.1: Flow of data collection in the study.

For each project, we gather the total number of code issues detected by *Sonar*. We also count the number of code issues according to severity, labeled as *blocker* (issue with severe impact on app behavior and that must be fixed immediately; e.g., memory leak), *critical* (issue that might lead to an unexpected behavior in production without impacting the integrity of the whole application; e.g., badly caught exceptions), *major* (issue that might have a substantial impact on productivity; e.g., too complex methods), and *minor* (issue that might have a potential and minor impact on productivity; e.g., naming conventions).

Directly comparing the number of issues in different projects can be misleading: small projects are more likely to have fewer issues than large projects, regardless of projects' code quality. To reduce this effect, we controlled for the size of the project by normalizing the number of issues by the number of files in a project, as follows:

$$I'(p) = \frac{I(p)}{F(p)}, \quad (4.1)$$

where p is a given project, $I(p)$ the number of issues of p , and $F(p)$ the number of files.

Since one of the main goals in this work is to assess how apps are being tested, we developed a tool to infer which testing frameworks a given project is using³. It works by fetching the source code of the app and looking for imported packages and

³Source code repository of the tool created to inspect automated testing technologies in Android projects: https://github.com/luiscruz/android_test_inspector

configuration files. The efficacy of this tool was validated with a random sample of apps which was manually labeled.

Table 4.1 lists all supported tools and frameworks aside with the number of search results in *StackOverflow*, as a proxy of popularity among the developers' community. Unit test tools, UI testing frameworks, and cloud based testing services were selected based on a previous survey on tools that support mobile testing [Linares-Vásquez et al., 2017c] and an online curated list of Android tools⁴.

Table 4.1: Android tools analyzed.

Name	StackOverflow Mentions*
<i>Unit testing</i>	
JUnit	67, 153
AndroidJUnitRunner	164
RoboElectric	245
RoboSpock	23
<i>UI testing</i>	
AndroidViewClient	474
Appium	9, 687
Calabash	1, 856
Espresso	4, 374
Monkeyrunner	1, 299
PythonUIAutomator	0
Robotium	3, 019
UIAutomator	1, 918
<i>Cloud testing services</i>	
Project Quantum	0
Qmetry	27
Saucelabs	1, 087
Firebase	100, 350
Perfecto	224
Bitbar [Kaasila et al., 2012]	16
<i>CI/CD services</i>	
Travis CI	3, 662
Circle CI	377
AppVeyor	655
CodeShip	564
CodeFresh	6
Wercker	200

* StackOverflow mentions as of January 26, 2018

We also collect information about the usage of CI/CD services in our study: *Travis CI*, *Circle CI*, *AppVeyor*, *Codeship*, *Codefresh*, and *Wercker*. The selection is based on CI/CD services that have a free plan for open source projects and which adoption can be automatically assessed — i.e., either they save their configuration in the code repository or have an open API that can be accessed with the *GitHub* organization and project name. Self-hosted CI/CD platforms (e.g., *GoCD*, *Jenkins*) are not included

⁴List of Android tools curated by Furiya: <https://github.com/wasabeef/awesome-android-tools> (Visited on June 5, 2020).

in this list. Although this is a subset of CI/CD services that can be used in a project, previous work found that *Travis CI* and *Circle CI* have more than 90% of share in *GitHub* projects using CI/CD services [Hilton et al., 2016].

We analyzed Android apps that are open source and published in *F-Droid*. The most popular version control repository is *GitHub*, being used by around 80% of projects. To make data collection clean, only projects using *GitHub* were considered. No other filtering was applied except in particular analyses that required data that was not available for all apps (e.g., *Google Play*'s ratings).

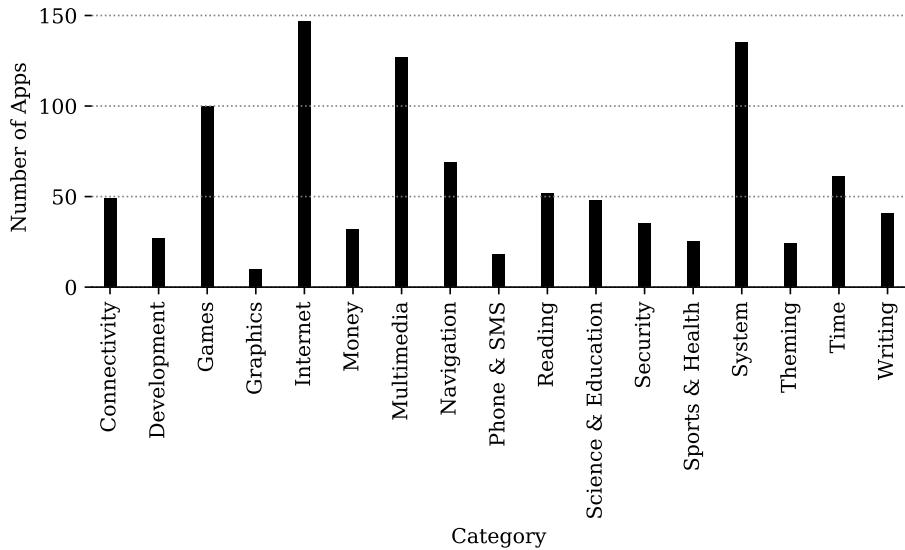


Figure 4.2: Categories of apps included in our study with the corresponding app count for each category.

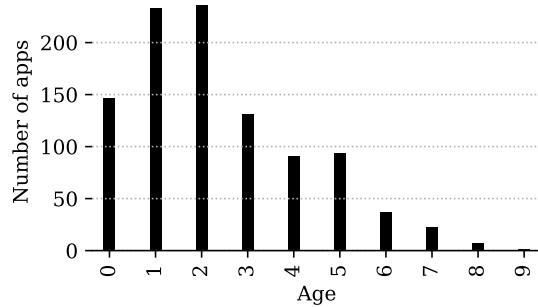


Figure 4.3: Distribution of the number of apps by age (in years).

Although *F-Droid*'s documentation reports that it hosts a total 2,800 apps⁵, only 1288 actually make it to the end user catalog. As we restrict our study to projects using *GitHub*, in total we analyze 1000 Android apps, roughly 35GB of source code collected between September 1–8, 2017. Apps in the dataset are spread amongst 17

⁵As reported in the *F-Droid*'s wiki page *Repository Maintenance*: https://f-droid.org/wiki/page/Repository_Maintenance (Visited on January 26, 2018).

categories, as presented in Figure 4.2, and are aged up to 9 years. The distribution of apps by age is presented in Figure 4.3.

Since in a few projects the static analysis tool *Sonar* does not successfully run, we collect code issues data for 967 apps, analyzing a total of 329,676 files. Additional data gathered from the *Google Play* store is available for 638 apps.

Reproducibility-oriented Summary

To foster reproducibility, based on previous guidelines for app store analyses [Martin et al., 2017], our work is best described as follows:

App Stores used to gather collections of apps. We use apps available on *F-Droid* and combine it with data available on *Google Play* store.

Total number of apps used. The study comprises 1000 apps.

Breakdown of free/paid apps used in the study. Only free apps are listed in our dataset.

Categories used. Apps in this study are spread across 17 categories. The distribution of apps is illustrated with the bar chart of Figure 4.2.

API usage. We collect usage of APIs related to test automation exclusively.

Whether code was needed from apps. Source code was required given the nature of analyses performed in the study.

Fraction of open source apps. Open source apps are used exclusively.

Static analysis techniques. We analyze source code with a self-developed tool for detection of tools, frameworks, and services' usage in the app's project and the static analysis techniques provided by *SonarQube* to gather code issues.

All scripts and tools developed in this work are publicly available with an open source license: https://luiscruz.github.io/android_test_inspector/. The same applies to the whole dataset, for the sake of reproducibility.

4.4 Prevalence of Automated Testing (RQ 4.1)

Testing is an essential task in software projects, and mobile apps are no different. Given the specific requirements of mobile apps, conventional approaches do not always apply. Thus, we want to assess how the FOSS mobile app development community is addressing automated testing. In this section, we study which testing approaches and technologies are most popular while discussing potential factors. In particular, we answer the following research question:

Research Question 4.1

What is the prevalence of automated testing technologies in the FOSS mobile app development community?

We compare the frequency of the automated testing technologies employed in the development of the apps in the dataset. The state-of-the-art technologies listed earlier in Table 4.1 were included, dividing them into three different categories: Unit testing, UI testing, and Cloud testing services. We resort to data visualizations and descriptive statistics to analyze the frequency of technologies in the dataset.

4.4.1 Results

Figure 4.4 shows, out of 1000 apps, the number of projects using a test framework. We include results for *Unit Testing*, *UI Testing*, and *Cloud Testing* frameworks. The first bar shows the number of apps that use any test tool. About 41% of apps (406) have tests. We can see that unit tests are present in 39% of projects (392) while *JUnit* is the most popular tool, with 36% of projects (363) adopting it. This means that 89% of projects with automated tests are using *JUnit*.

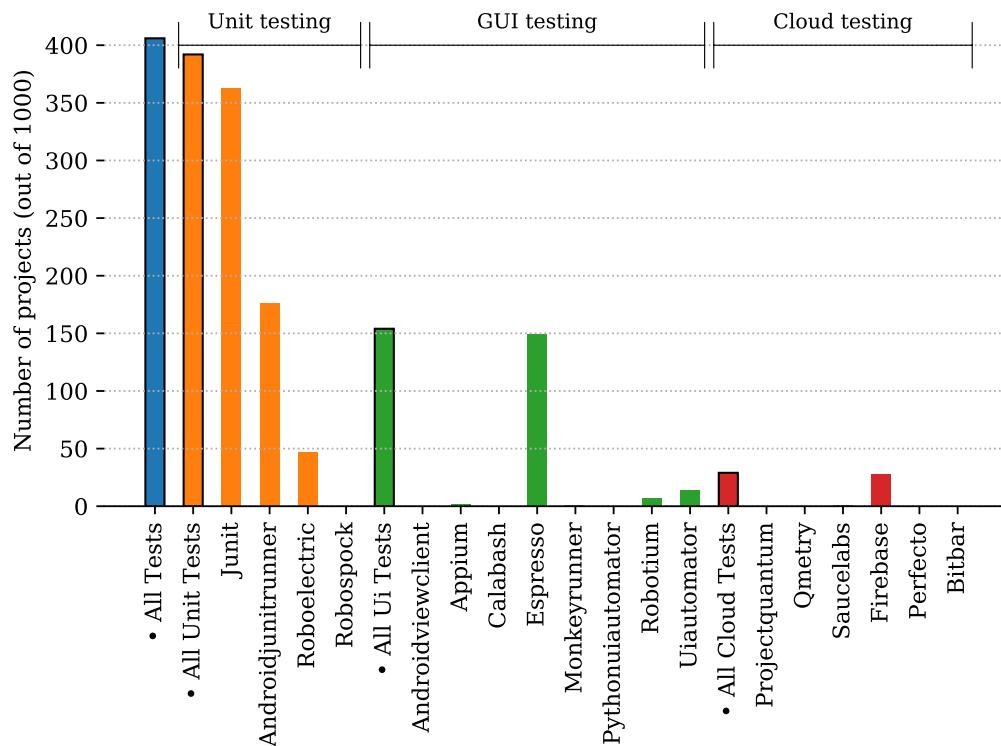


Figure 4.4: Number of projects per framework.

Only 15% of projects (154) have automated UI tests. *Espresso* is the most used framework — almost every project with UI tests is using *Espresso*. *UIAutomator*, *Robotium*, and *Appium* are used by a very small portion of projects in our dataset,

while *AndroidViewClient*, *Calabash*, *Monkeyrunner*, and *PythonUIAutomator* are not used in any project.

With less than 3% of projects (29) employing them, cloud testing services have not found their way into the open source mobile app development community. In total, 28 projects use *Google Firebase*, whereas only 1 project uses *Saucelabs*. All the other cloud test services in this study are yet to be adopted.

4.4.2 Discussion

Most mobile apps published in *F-Droid* do not have automated tests. Developers are relying on manual testing to ensure proper functioning of their apps, which is known to be less reliable and to increase technical debt [Stolberg, 2009; Bavani, 2012; Karvonen et al., 2017].

Given their simplicity, unit tests are the most common form of tests. *JUnit* is the main unit testing tool and the reason lies in the official Android Developer documentation for tests⁶, which introduces *JUnit* as the basis for tests in Android. Furthermore, other test tools often rely internally on *JUnit* (e.g., *AndroidJUnitRunner*).

Other unit testing tools such as *AndroidJUnitRunner* and *Robolectric* do not have a substantial adoption. These tools help running unit tests within an Android environment, instead of the desktop's **Java Virtual Machine (JVM)**. This is important given the complexity of an Android app's lifecycle, which might affect test results. However, many apps still do not cross that limit, providing only unit tests for parts of the software that can run absent from the mobile system. Since many apps follow a similar structure, based on Android's framework enforced design patterns, easily customizable boilerplate tests should be delivered along with those patterns.

UI tests are not so popular (15%), which can be explained by their cumbersome maintainability reported in previous work [Z. Gao et al., 2016; Coppola, 2017; X. Li et al., 2017]. Although there are many UI testing frameworks available, *Espresso* is the only one with substantial adoption. This is consistent with the phenomenon of *JUnit* for unit tests: *Espresso* is also promoted in the official Android Developer documentation. In fact, it is distributed with the Android SDK. Another strength is that *Espresso* provides mechanisms to prevent flakiness and to simplify the creation and maintenance of tests.

Previous work has considered *Espresso* as the most energy-efficient UI testing framework. The fact that these projects are already using it, leaves an open door for the creation of energy tests. On the other hand, *Espresso* still provides a limited set of user interactions, which can be a barrier to high test coverage [Cruz and Abreu, 2018a].

⁶Getting Started with Testing Android guide available at: <https://developer.android.com/training/testing/fundamentals> (Visited on June 5, 2020).

Unfortunately, studied cloud testing services have not reached the open source app community. This is probably due to the recency of the introduction of these technologies and the lack of a testing culture in mobile app development, as shown in our results.

The good news is that we observe an increasing adoption of unit and UI tests in the last two years. This trend can be observed by comparing our findings with previous work [Kochhar et al., 2015]; while the previous study highlights that the prevalence of automated tests in mobile apps was merely 14%, in this work, we observe that 41% of FOSS apps are developed with automated testing tools.

These findings provide useful implications for the development of new testing tools and techniques. Previous work has shown the importance of creating new types of tests for mobile apps (e.g., energy tests, security tests) [Linares-Vásquez et al., 2017c; Muccini et al., 2012; Wang and Alshboul, 2015]. Our results show the importance of simplifying the learning curve and the project's setup. Hence, new types of tests should be compatible at least with *JUnit* and *Espresso*, avoiding reinventing the wheel or complicating usage with new dependencies.

In addition, the adoption of these tools by the FOSS community is highly sensitive to the quality and accessibility of documentation. The fact that *Google* has control over the official documentation does not help third-parties to come aboard. Perhaps the official documentation should feature more tools that are not delivered with the Android SDK. The same concern applies to the academia that is developing many interesting tools for mobile development and testing. Often the lack of documentation is a big barrier to the adoption of innovative techniques by the software industry [Gousios et al., 2016; Kochhar et al., 2015].

Only 41% of FOSS apps have automated tests. Unit testing frameworks are the most popular, comprising 39% of projects. UI testing is being used by 15% of projects, while the adoption of Cloud testing platforms is negligible (3%).

4.5 Evolution of the Testing Culture (RQ 4.2)

Android testing tools are in constant evolution to fit the ever-changing constraints and requirements of mobile apps. Although we are currently far from having a satisfactory prevalence of automated testing, the evolution from past years can provide actionable information. We study which technologies and types of testing have gained momentum, and which ones are still failing to be perceived as beneficial in FOSS mobile app projects.

In particular, this section answers the following research question:

Research Question 4.2

Are today's mature FOSS Android apps using more automated testing than yesterday's?

We analyze how the adoption of automated testing relates to the age of an app and the time of an app's last update. We dig further and study the adoption of automated testing in mature FOSS apps by years since the last update. Trends on automated testing adoption over time are analyzed using scatter plots.

4.5.1 Results

The percentage of apps that are doing tests grouped by their age is presented in the plot of Figure 4.5. The data is presented from older to newer projects (i.e., 9–0 years old). The size of each circle is proportional to the number of apps with that age (e.g., older projects have smaller circles, meaning that there are fewer projects for those ages.). It is used to show the impact of results in each case. E.g., since projects that are six or more years old have small circles, they comprise a small number of projects. Hence, trends in those age groups are not significant.

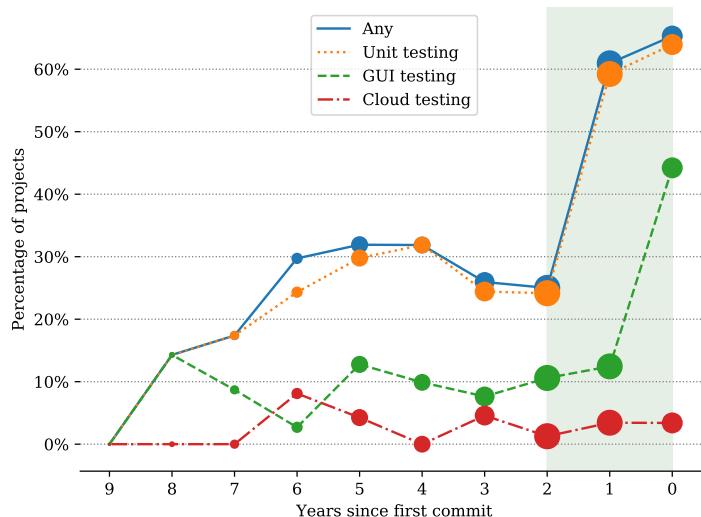


Figure 4.5: Percentage of Android apps developed in projects with test cases over the age of the apps.

The timeline in Figure 4.5 shows that apps that are less than two years old have significantly more tests than older apps. Moreover, the usage of UI testing frameworks has increased among apps that are under two years old.

In addition, we present in Figure 4.6 how new apps have been changing the overall test automation adoption. In the past two years (shaded region) the slope of projects with tests is higher than projects without tests. However, this recent change is not

able to change the overall pervasion of test automation: most projects are not doing it.

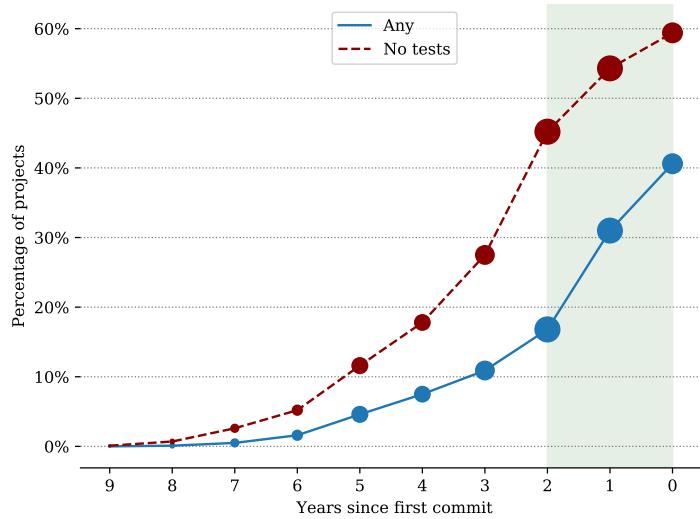


Figure 4.6: Cumulated frequency of projects with and without tests (from 9 to 0 years old), normalized by the total number of projects.

Finally, we present the timeline of the adoption for different kinds of testing techniques in Figure 4.7. The aforementioned trend is observable for unit testing and UI testing, which have a higher slope in the past two years (shaded region).

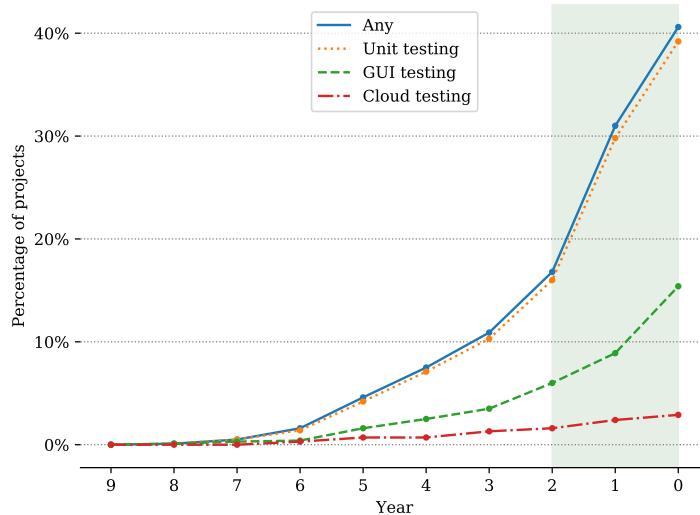


Figure 4.7: Cumulated percentage of projects with tests (from 9 to 0 years old), normalized by the total number of projects. All test categories are represented.

4.5.2 Discussion

Results show a significant increase in automated testing amongst new FOSS apps. However, the fact that older apps have a lower adoption rate of automated testing can

be explained by two phenomena: 1) automated testing is becoming more accessible to developers, who are becoming more aware of its benefits, 2) at some point during the lifespan of a project, developers realize that the overhead of maintaining automated testing is not worth the benefits and decide to remove it. While the first phenomenon reveals a positive trend, the latter is quite alarming — automated testing does not provide a long-term solution.

Giving a better sense of which phenomenon is more likely to happen, Figure 4.6 reveals that automated testing has been gaining popularity in the last two years.

It is worth noting that this increase is happening in both unit testing and UI testing. The fact that UI testing is gaining popularity is important — unit testing per se does not provide means to achieve high test coverage in mobile apps. This increase provides more case studies for researchers to study new types of mobile testing (e.g., energy, security, etc.).

Open source mobile developers are becoming more aware of the importance of using automated tests in their apps. This is observed more for apps that are updated recently than those updated several years ago.

4.6 Automated Testing vs Popularity (RQ 4.3)

In this section, we compare popularity metrics with the adoption of automated testing practices in FOSS Android apps. In other words, we answer the following research question:

Research Question 4.3

How does automated testing relates to popularity metrics in FOSS Android apps?

For this purpose, the following popularity metrics were selected:

Number of Stars. The number of Github users that have marked the project as favorite.

Number of Forks. The number of Github users that have created a fork of the repository.

Number of Contributors. The number of developers that have contributed to the project.

Number of Commits. The number of commits in the repository.

Average Rating. The average user rating from *Google Play* store.

Number of Ratings. The number of users rated the app on *Google Play*.

These metrics depend on a myriad of factors, which do not necessarily relate to mobile app development processes. Yet, they are notable metrics that developers do care about. Typically, developers need to drive their development process based on multiple sources of feedback [Nayebi et al., 2018]. We want to investigate whether there is any kind of relationship between these features and automated testing. Relationships can help motivate mobile app developers employing tests in their projects.

To remove atypical cases, we perform an outlier detection using the Z-score method with a threshold of three standard deviations. In addition, we perform the normality test *Shapiro-Wilk*, which tests the null hypothesis that data follows a normal distribution.

Then we apply hypothesis testing, using the non-parametric test Mann-Whitney U, with a significance level (α) of 0.05. We may also consider a parametric test (e.g., the standard t-test), in case we find variables that follow a Normal distribution. In addition, since we are conducting multiple comparisons, the Benjamini-Hochberg procedure is used to correct p -values and control false discovery rate.

The independent variable is whether an app has tests in its project source code while the dependent variables are the popularity metrics.

The hypothesis test is formulated as follows, with populations WO and W as the population of **apps without tests** and the population of **apps with tests**, respectively:

$$H_0 : P(W > WO) = P(WO > W)$$

$$H_1 : P(W > WO) \neq P(WO > W)$$

In other words, we test the null hypothesis (H_0) that a randomly selected value from population W is equally likely to be less than or greater than a randomly selected value from sample WO .

We perform hypothesis testing for each of the aforementioned metrics, formulated as follows:

Number of Stars

H_0 : a project with tests (W) has the same number of Github stars as a project without tests (WO).

H_1 : the number of Github Stars in projects with tests is different from the number of stars in a project without tests.

Number of Forks

H_0 : a project with tests (W) has the same number of forks as a project without tests (WO).

H_1 : the number of forks in projects with tests is different from the number of stars in a project without tests.

Number of Contributors

H_0 : projects with tests (W) have the same number of contributors as a project without tests (WO).

H_1 : the number of forks in projects with tests is different from the number of contributors in a project without tests.

Number of Commits

H_0 : a project with tests (W) has the same number of commits as a project without tests (WO).

H_1 : the number of commits in projects with tests is different from the number of commits in a project without tests.

Average Rating

H_0 : a project with tests (W) has the same rating as a project without tests (WO).

H_1 : the rating of a randomly selected project with tests is different from the rating in a project without tests.

Number of Ratings

H_0 : a project with tests (W) has the same number of rating as a project without tests (WO).

H_1 : the number of ratings of a randomly selected project with tests is different from the number of ratings in a project without tests.

In addition, we perform effect size analyses for variables showing statistical significance. We compute the mean difference ($\Delta\bar{x} = \bar{x}_W - \bar{x}_{WO}$), the difference of the medians ($\Delta Md = Md_W - Md_{WO}$), and the **Common Language Effect Size (CL)** [McGraw and Wong, 1992].

The mean difference ($\Delta\bar{x}$) measures the difference between the means of apps with tests (W) and apps without tests (WO) for a particular popularity metric. We compute it for being a conventional effect-size metric. In addition, since the distribution is not necessarily normal, we compute the difference of the medians (ΔMd) between apps with tests (W) and apps without tests (WO). Given that the median of a sample is the value that separates the higher half from the lower half of the sample, ΔMd measures how different this median value is in the two distributions.

There are nonetheless a few cases in which ΔMd does not capture differences in the two distributions [Kerby, 2014]. We complement the effect size analysis with the CL measure. CL is the recommended measure when there is no assumption on the shape of the distributions of the two samples being tested and it is commonly used in tandem with Mann-Whitney U test [Leech and Onwuegbuzie, 2002]. One advantage

of using CL to measure effect size is that it can be easily interpreted [Brooks et al., 2014]: it is the probability that the value from an individual randomly extracted from one sample will be higher than the value from an individual randomly extracted from another.

4.6.1 Results

The distributions of the popularity metrics are depicted in the boxplots of Figure 4.8. The medians are represented by the orange solid lines, while the means are by green dashed lines. The results of the normality tests *Shapiro-Wilk* yielded a low p -value ($p < 0.001$) for all metrics. Thus, none of the metrics follows a normal distribution, which highlights the suitability of using the Mann-Whitney U test over the standard t-test.

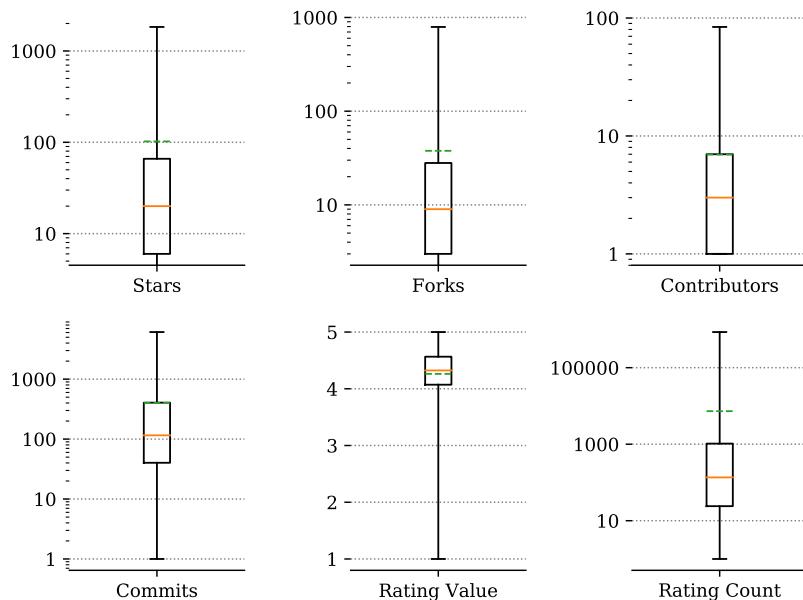


Figure 4.8: Boxplots with the distributions of the popularity metrics. Note that the y-axis is in log-scale for all metrics but ratings.

Hypothesis testing results are shown in Table 4.2 along with the effect size analysis: mean difference ($\Delta\bar{x}$), difference of median (ΔMd), and CL expressed in percentage. The bigger the effect size is, the bigger is the metric for apps with tests. The effect size analysis is only relevant in cases with statistical significance, which are highlighted in bold text.

There is statistical evidence that FOSS Android apps with tests are expected to have more **commits** and more **contributors**. Note, however, that this evidence does not imply that tests boost these variables. Conclusions must analyze the causality of this relationship (i.e., whether tests are cause or consequence) and the fact that there are many external variables that are expected to have a significant impact (e.g.,

Table 4.2: Statistical analysis of the impact of tests on the popularity of apps.

	<i>p</i> -value	$\Delta\bar{x}$	ΔM_d	CL (%)
Stars	0.2130	54.78	3.00	52.74%
Forks	0.4525	11.39	1.00	51.40%
Contributors	0.0052	2.17	0.00	55.80%
Commits	0.0008	247.58	49.00	57.13%
Rating Value	0.0835	0.05	0.05	54.77%
Rating Count	0.2465	-894.26	-56.00	47.03%

target users, originality of idea, design, marketing, etc.). Nonetheless, no statistical significance was found between having automated tests and the number of *GitHub* stars, forks and ratings on *Google Play*.

Projects with tests have on average more 248 commits in the whole project. The CL measure is small but substantial: the probability of a project with tests having more commits than a project without tests is 57%. Although the number of commits increases, one can argue that the number of commits can be related to overhead created by tests maintenance.

Projects with tests have a small but substantial CL for the number of contributors: the probability that a project with tests will have more contributors is 56%. Nevertheless, the direction of this relationship cannot be assessed with these results — i.e., there is no evidence of whether the presence of tests is a consequence of the high number of contributors in the project or, in contrary, it is a way of attracting more developers to contribute.

Tests and Contributors: developers' perception? We decided to conduct a follow-up study to assess the developers' perception of whether tests can lead to more contributors. We contacted 343 mobile open source developers to answer a survey with two close-ended questions:

1. *Do you think that more tests benefit/attract newcomers?*

Possible answers were: *Yes*; *No*; and *Maybe*.

2. *Is the presence of tests a reason or a consequence of a big community of contributors?*

Possible answers were: *Most likely a reason*; *Most likely a consequence*; *Both equally*; and *No impact*.

Respondents had an additional box where they could optionally leave their comments or feedback on the subject. Developers were selected by being active in an open

source mobile application available on *GitHub*. In the end, we had 44 responses. Data collected was anonymized and it is available online⁷.

As shown in the pie chart of Figure 4.9, 45.5% of our respondents believe that tests help new developers to contribute in a project, while 38.6% are not sure, and only 15.9% disagree with the statement. The pie chart of Figure 4.10 shows that, despite the recognized improvement from having tests, the majority of respondents believe that the presence of tests is more likely a consequence from having a big community of contributors (43.2%). A smaller part of respondents (25%) believe that the presence tests and the size of the community do not affect each other — i.e., they both depend on a different variable. Other respondents believe both variables affect each other (22.7%), while only 9.1% reckon tests as the cause.

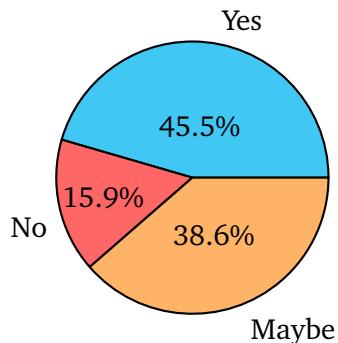


Figure 4.9: Do tests attract newcomers?

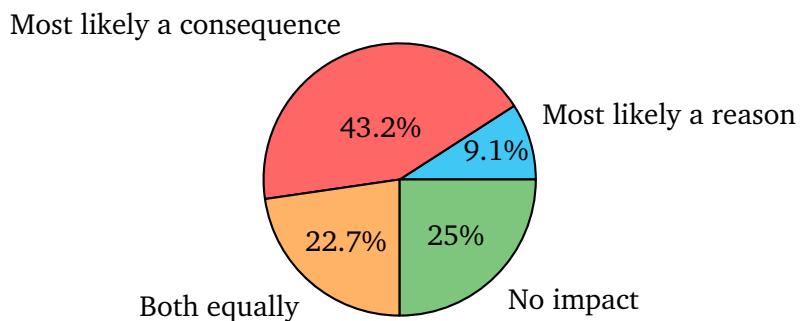


Figure 4.10: Tests: cause or consequence of a big community?

Feedback submitted by some developers provided some insights on their personal experience. Some developers pointed out that the adoption of CI/CD is probably “more influential than the actual tests”. Other developers emphasized the importance of having tests as “a good starting point for newcomers to get familiar with the project’s code and its features”. Finally, some developers state that the “maintenance burden of automated tests is really high” and that they can block major refactorings in software projects.

⁷Questionnaire responses are available online: https://docs.google.com/spreadsheets/d/e/2PACX-1vS39pRfJ7AZLXFVsqZvqZqEkJwg88AqxUTN_kKC6t2ay0JBTQByhnuVAFtoGP093yyjsCGX7FU3uON1/pubhtml?gid=1063568719&single=true

4.6.2 Discussion

Results show that FOSS Android projects with tests have more commits and more contributors. The increase in the number of commits can be explained by an overhead of commits induced by the maintenance and configuration of tests.

Responses to the questionnaire show that the presence of tests is more likely a consequence of having a big community. In addition, tests can help new developers contribute to the project. Since one of the main concerns in open source projects is to foster the community to contribute⁸, the importance of tests for this purpose cannot be discarded. Conventionally, maintainers of open source projects target this goal by inviting contributors, providing social and communication tools, and making sure that instructions on how to contribute are well documented. These results show that tests should also be part of their agenda.

This relationship is consistent with previous work. Automated tests help new developers be more confident about the quality of their contributions [Gousios et al., 2016]. Contributors are able to create PRs to a project with a reasonable level of confidence that other parts of the software will not break. The same applies to the process of validating a PR. Integrators usually have some barriers when accepting contributions from newcomer developers [Gousios et al., 2016]. The presence of automated tests helps reduce that barrier, and contributions with tests are more likely to be accepted [Gousios et al., 2016]. Another aspect of automated tests that contributes to this trend is the reported ability to provide up-to-date documentation of the software [Van Deursen, 2001; Beck, 2000].

Previous work that shows that app store's ratings are not able to capture the quality of apps [de Langhe et al., 2016; Ruiz et al., 2017]. Our results show that this is also the case for tests: there is no relationship between using tests and rating on *Google Play*.

Our findings have direct implications for different stakeholders of mobile software projects. Developers have to start using automated tests in their code in order to ensure quality in their contributions. Open source project maintainers must promote a testing culture to engage the community in their projects.

Automated testing is important to foster the community to contribute. There is statistical evidence that FOSS Android projects with tests have more contributors and more commits. Number of GitHub Stars, Github Forks and ratings on Google Play did not reveal any significant impact.

⁸*Five best practices in open source: external engagement* by Ben Balter: <https://ben.balter.com/2015/03/17/open-source-best-practices-external-engagement/> (Visited on June 5, 2020).

4.7 Code Issues and Test Automation (RQ 4.4)

Code issues are related to potential vulnerabilities of software. It is a major concern of developers to ship software with a minimal number of code issues. We study whether automated testing can help developers deploy mobile app software with fewer code issues. In other words, we aim at answering the following research question:

Research Question 4.4

How does automated testing affect code issues in FOSS Android apps?

We use the issues detected by the static analysis Sonar tool as a proxy of software code issues. We apply Sonar to our dataset of 1000 Android apps. As mentioned in Section 4.3, Sonar issues are divided into four categories, based on the severity of their impact. We evaluate the number of issues normalized for the number of files in the project ($I'(p)$).

We apply the same approach used in Section 4.6: we use hypothesis testing with the Mann-Whitney U test using a significance level (α) of 0.05. Benjamini-Hochberg procedure is used to correct p -values since four tests are performed in the same sample. Mean difference ($\Delta\bar{x}$), difference of median (ΔMd), relative difference ($\frac{\Delta Md}{Md_W}$), and CL are used to analyze effect size.

4.7.1 Results

We successfully collected code issue reports from 967 apps. It was not possible to collect data from 33 apps: Sonar failed due to characters invalid with UTF-8 encoding. This was the case of the reading app *FBReaderJ* and its file *ZLConfigReader.java* that contained characters that not even Github is able to render⁹. Since these apps consisted of a small portion of our dataset (3%), we decided to leave them out of this part of the study.

Table 4.3 presents descriptive statistics of the number of code issues per file $I'(p)$ for each level of severity — size of the sample (N), median (Md), mean (\bar{x}), and standard deviation (s). The table also presents the results of normality tests with the p -values for Shapiro-Wilk tests ($X \sim N$), showing that none of the metrics follows a normal distribution. Statistics are presented for both apps with tests (W) and apps without tests (WO).

Figure 4.11 illustrates the distribution of the $I'(p)$ in projects with tests (blue line, hatch fill) and without tests (red line, empty fill) for different types of issues. The

⁹Example of a source code file incompatible with Sonar tool: <https://git.io/fxNg9> (Visited on June 5, 2020).

Table 4.3: Descriptive statistics of code issues on apps with (W) and without (WO) tests.

	Tests	N	Md	\bar{x}	s	$X \sim N$
Blocker	W	398	0.00	0.02	0.04	$p < 0.0001$
	WO	569	0.00	0.05	0.59	$p < 0.0001$
Critical	W	398	0.24	0.34	0.39	$p < 0.0001$
	WO	569	0.26	0.48	0.90	$p < 0.0001$
Major	W	398	0.50	0.73	0.80	$p < 0.0001$
	WO	569	0.52	0.84	1.09	$p < 0.0001$
Minor	W	398	0.61	0.87	0.93	$p < 0.0001$
	WO	569	0.73	1.27	2.12	$p < 0.0001$

mean for each group is depicted with a dashed green line, while the median with a solid orange line. Types of issues with a statistically significant difference between W and WO are highlighted with thicker lines. Results show that projects with tests have significantly less minor code issues than projects without tests.

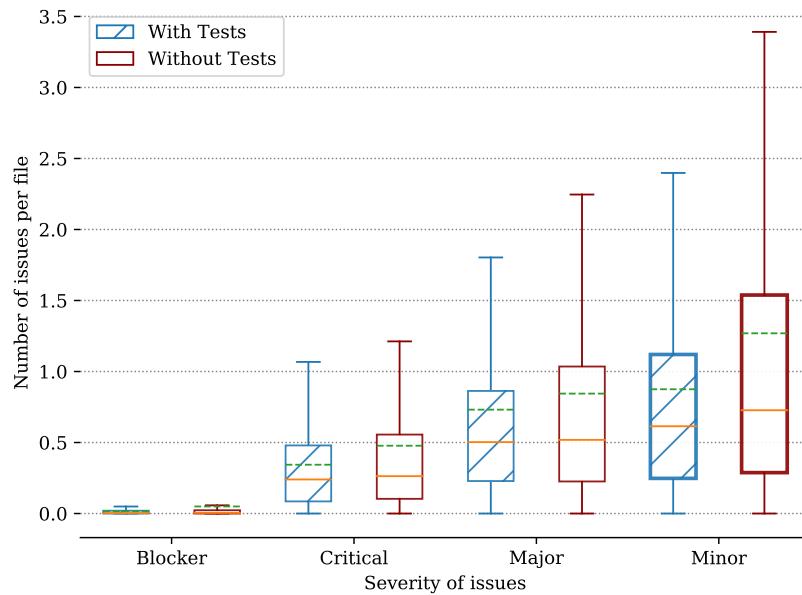


Figure 4.11: Comparison of the number of issues per file in projects with and without tests. Green dashed lines in each box represent the mean value, while orange solid lines represent the median.

Table 4.4 reports the resulting p -values and computes the effect-size metrics: mean difference ($\Delta\bar{x}$), difference of median (ΔMd), relative difference ($\frac{\Delta Md}{Md_W}$), and CL.

The number of minor issues per file increases significantly in projects without tests. The difference of median shows that projects without tests are expected to have 0.11 more minor issues per file (increase of 18%). Furthermore, as reported with the CL, projects without tests have more minor issues than projects with tests with a probability of 54%. The number of issues for higher severity levels is not significantly affected.

Table 4.4: Statistical analysis of the impact of tests in mobile software code issues.

Severity	p-value	$\Delta\bar{x}$	ΔMd	$\frac{\Delta Md}{Md_W} (\%)$	CL (%)
Blocker	0.1643	0.0337	0.0014	48	52.09%
Critical	0.1150	0.1337	0.0234	9	52.97%
Major	0.2939	0.1130	0.0157	3	51.02%
Minor	0.0440	0.3940	0.1127	18	54.32%

4.7.2 Discussion

Results show that there is a statistically significant and substantial relationship between using automated testing and the number of minor code issues that appear in the project. FOSS Android projects without automated testing have significantly more minor code issues. Given that only 41% of apps in this study have automated tests, mobile developers need to be aware of the importance of testing their apps.

On the other hand, although the normalized number of blocker, critical, and major bugs is higher for apps without tests than those with tests, the difference is not statistically significant. Other alternatives, such as manual testing, code inspection, or static analysis, are probably preventing such issues. Our sample size may also not be large enough to make the result to be statistically significant.

There is statistical evidence that FOSS Android projects without tests have 18% more minor code issues per file. In our sample, projects without tests also had more code issues for other severity levels: major (3%), critical (9%), and blocker (48%).

4.8 CI/CD vs Test Automation (RQ 4.5)

CI/CD has been proved to be beneficial in software projects and to have even better results when employed along with automated testing [Hilton et al., 2016; Zhao et al., 2017]. Thus, we study whether mobile app developers are using CI/CD in its full potential. Moreover, we delve into how mobile app projects set themselves apart from conventional software projects in terms of CI/CD adoption. In sum, this section answers the following research question:

Research Question 4.5

What is the relationship between the adoption of CI/CD and automated testing?

To answer this research question, we start by comparing the adoption of the different studied CI/CD technologies in Android FOSS projects. In addition, we compare the

frequency of projects that have adopted one of the studied CI/CD tools with the frequency of projects using automated testing.

For this analysis, we resort to data visualizations. To validate the relationship between automated testing and CI/CD we use Pearson's chi-squared test with a significance level of 0.05. This test was selected for being commonly used to compare binary variables.

4.8.1 Results

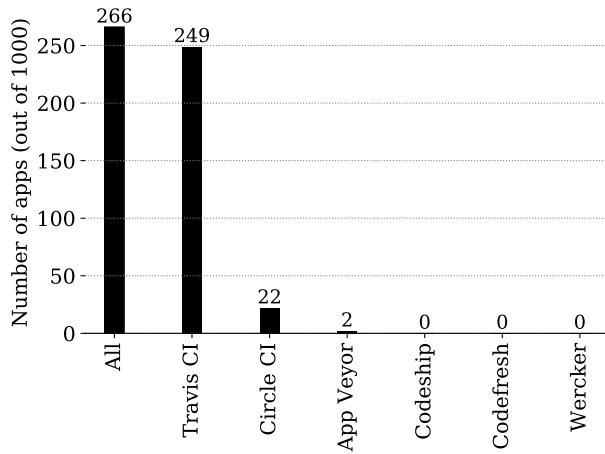


Figure 4.12: Android apps using CI/CD platforms.

We first analyze which apps are using CI/CD pipelines in their development practices. The distribution of CI/CD pipelines among these platforms is given in Figure 4.12. *Travis CI* is the most popular platform with 249 apps using it (25%), followed by *Circle CI*, being used by 2% of apps. However, in total, only 27% have adopted CI/CD.

The relationship between the prevalence of CI/CD and prevalence of tests is depicted by the mosaic plot in Figure 4.13. The size of each area is proportional to the number of apps in each group. Nearly 50% of apps are not having tests nor adopting CI/CD (region A). 26% of apps, despite having tests, are not using CI/CD (region B). 12% of apps are using CI/CD but are not doing any automated tests (region C). Only 15% of apps are using CI/CD effectively, with automated tests (region D). In addition, the mosaic plot suggests that automated testing is more prevalent in projects with CI/CD than projects without. This is confirmed by the Pearson's chi-squared test: $\chi^2 = 31.48, p = 2.009e-8$.

Online coverage trackers are useful tools that play well with CI/CD platforms. They help ensure that the code is fully covered. Nevertheless, only 19 projects are using it — 9 use *Coveralls* and 12 use *Codecov*, having 2 projects using both platforms.

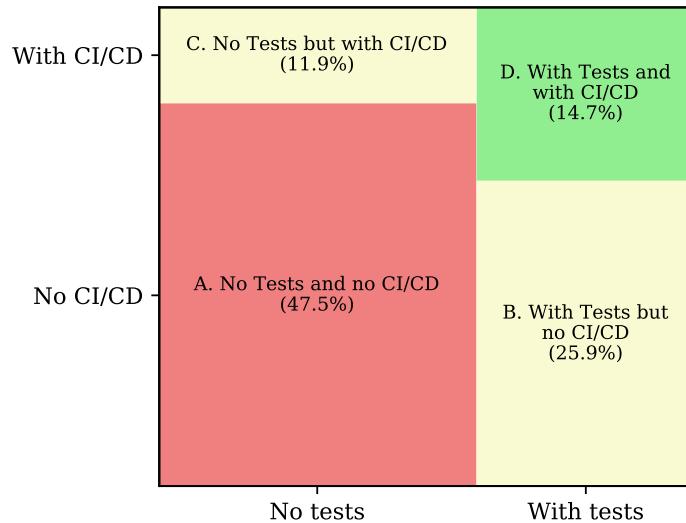


Figure 4.13: Relationship between apps using CI/CD and apps using tests.

However, only 4 have line coverage above 80%, and no meaningful results can be extrapolated.

4.8.2 Discussion

CI/CD is not as widely adopted by mobile app developers as compared to developers of general FOSS projects — only 26% of apps have adopted CI/CD services while the adoption in general open source software hosted by *GitHub* is 40% [Hilton et al., 2016].

There are 12% of apps that, despite using CI/CD, do not have automated tests. In practice, these projects are only using CI/CD tools to run static analyses. Yet, they rely on a pipeline that requires an approver to manually build and test the app.

The fact that there are projects that have tests but did not adopt CI/CD (26%) is also concerning. One of the main strengths of adopting CI/CD is improving software quality through test automation [Zhao et al., 2017]. Although CI/CD services have made a good work in simplifying the configuration of Android specific requirements (e.g., SDK version, emulator, dependencies, etc.), developers have reported that the main obstacle in adopting CI/CD in a project is having developers who are not familiar with it [Hilton et al., 2016]. Nevertheless, since these projects are already using automated tests, they could potentially benefit from a CI/CD pipeline with little effort. More research needs to be conducted to assess why mobile developers are not adopting CI/CD in their projects.

Travis CI and *Circle CI* are the most used CI/CD services, as expected from previous results for other types of software [Hilton et al., 2016]. Although the other platforms

have a well documented support for Android, they are not being used by the community.

Even more surprising is the fact that, from the 147 apps with both CI/CD and tests, only 19 are actually promoting full test coverage with coverage tracking services. This suggests that coverage is not a top priority metric for mobile developers, which is in sync with concerns by Gao *et al.* who have reported the need for coverage criteria to meet the idiosyncrasies of mobile app testing [J. Gao et al., 2014]. In particular, *Coverall* and *Codecov* platforms only report line coverage. Different coverage criteria, such as event/frame coverage, would be more suitable in the context of mobile apps.

More education and training is needed to get full benefits of CI/CD for mobile apps. Developers that are already performing automated tests in their apps should explore the integration of a CI/CD pipeline in their projects. This is also a good opportunity for newcomer developers willing to start contributing to open source projects.

CI/CD in mobile app development is not as prevalent as in other platforms; Automated testing is more prevalent in projects with CI/CD.

4.9 Hall of Fame

We have selected a set of apps from our dataset that we consider good candidates for studying best practices from the mobile app development community. We perform a systematic selection by choosing projects that perform unit tests, UI tests and are using CI/CD. In total, 54 apps satisfy these requirements¹⁰. We present in Table 4.5 one app for each category based on the popularity of that app among developers, using the number *Github Stars* as a proxy. Some categories, namely *Games*, *Money*, and *Phone & SMS*, did not have any app that meets the requirements.

Note, however, that although these projects follow best practices, they are not necessarily the ones with the highest ratings (e.g., rating in *Google Play*, number of Forks in *Github*). The success of apps also depends on a myriad of other factors. Nevertheless, the impact of best practices is not negligible and for that reason, these projects can be used as role models for new projects or subjects for case studies for further research.

¹⁰The whole set of apps in the Hall of Fame can be accessed online: https://luiscruz.github.io/android_test_inspector/.

Table 4.5: Hall of fame.

Category	Organization	Project Name
Internet	k9mail	k-9
Multimedia	TeamNewPipe	NewPipe
Writing	federicoiosue	Omni-Notes
Theming	Neamar	KISS
Time	fossasia	open-event-android
Sports & Health	Glucosio	android
Navigation	grote	Transportr
System	d4rken	reddit-android-appstore
Reading	raulhaag	MiMangaNu
Security	Oxbb	otp-authenticator
Science & Education	EvanRespaut	Equate
Connectivity	genonbeta	TrebleShot
Development	Adonai	Man-Man
Graphics	jiikuy	velocitycalculator

4.10 Threats to validity

Construct validity Code issues collected with *SonarQube* are used to measure the quality of code. Some projects might not follow common development guidelines due to specific requirements. Thus, generic static rules might not be able to capture the quality of such projects. Nevertheless, we expect that this is the case of a minimal number of apps and results are not affected. Metrics from *Google Play* and *GitHub* are used as proxies to measure user satisfaction, and popularity of apps. These metrics are affected by a number of factors and not always are sufficiently dynamic to cope with changes in the app [Ruiz et al., 2017].

Furthermore, the online coverage trackers investigated in this study only support line coverage. Coverage metrics for events or UI frames are more suitable for mobile applications. These metrics were not evaluated as they are not available in the state-of-the-art online coverage trackers. Finally, we did not consider AIG techniques since they are more advanced and thus are not popularly used in mobile app development yet.

Internal validity The usage of a test framework or service was assessed through a self-developed automatic tool based on static analysis and Web requests to service's APIs. To validate the accuracy of our tool we have manually labeled a random sample of 50 apps and compared the results. Our tool has successfully passed our validation with no false positives and no false negatives, but we understand that some corner cases may not have been checked yet. The same applies to the static analysis tool *SonarQube* used to collect code issues — it provides an approximation of the actual set of code issues in a project. Some issues detected by *SonarQube* may be false positives or may not generalize to other, distinct projects. Nevertheless, we

argue such cases are rare and they are not expected to have a significant effect in results.

External validity Our work has focused on free and open source apps. Our 1000-app dataset comprises a good proportion of these apps that are currently available for Android users. Findings in this work are likely to generalize to types of apps with a caveat: private companies usually have a different approach from open source organizations on software testing [Joorabchi et al., 2013]. We did not include testing services without a free plan for open source projects; Paid apps have different budgets and might be more willing to use paid services in their projects. Legal and copyright restrictions do not allow us to scope apps with commercial licenses. This is a known barrier for research based on app store analysis [Krutz et al., 2015; Nagappan and Shihab, 2016; Martin et al., 2017].

The adoption of CI/CD is based on a subset of CI/CD services available, as described in Section 4.3. This subset is equivalent to the one used by Hilton et al. to study CI/CD adoption in general software projects [Hilton et al., 2016].

4.11 Summary

Testing is a crucial activity during the software development lifecycle to ascertain the delivery of high quality (mobile) software. In this chapter, we study testing practices in the mobile development world. In particular, we investigated working habits and challenges of mobile app developers with respect to testing.

Concretely, in this chapter:

- We present a software tool that uses static analysis to identify the testing technologies being used in Android projects (cf. Section 4.3). Available here: https://github.com/luiscruz/android_test_inspector.
- We conduct a large-scale study with 1000 FOSS Android apps aiming at understanding the testing culture in mobile applications. Our results show that:
 - FOSS mobile apps are still tested in a very ad hoc way, if tested at all. Testing technologies were absent in almost 60% of projects in this study. *JUnit* and *Espresso* were the most popular technologies in their category with an adoption of 36% and 15%, respectively (answering RQ 4.1).
 - Although automated testing is far from being used, it has become more popular in recent years. This is particularly evident for unit testing and UI testing (answering RQ 4.2).
 - Testing activities play an important role in engaging contributors to a FOSS Android project. Moreover, projects with tests also reveal a higher number of commits. The number of *Github Stars*, *Github Forks*, and

ratings on *Google Play* did not reveal any significant relationship with the adoption of automated testing (answering RQ 4.3).

- Projects without tests have a higher number of minor code issues. Thus, automated testing plays an important role in ensuring the quality of the code of FOSS Android apps (answering RQ 4.4)
- CI/CD is not being effectively used by FOSS Android apps. Besides, results show that projects using CI/CD are more likely to adopt test automation (answering RQ 4.5).
- We select 54 FOSS Android apps that have adopted best testing practices (cf. Section 4.9). Since measuring energy consumption requires the usage of UI automation (cf. Chapter 2), these apps are a useful resource for future work on energy efficiency.

Energy Impact of Performance Anti-Patterns in Android Apps



Performance-based Guidelines for Energy Efficient Mobile Applications

Luis Cruz and Rui Abreu

In: IEEE/ACM International Conference on Mobile Software Engineering and Systems, MobileSoft, 2017.

Abstract

As a result of the advent of mobile computing, a number of best practices have been delivered to optimize the performance of Android applications. However, these guidelines fall short to address energy consumption. As mobile software applications operate in resource-constrained environments, guidelines to build energy-efficient applications are of utmost importance. In this chapter, we study whether eight best performance-based practices recommended by Google have an impact on the energy consumption of Android applications. In an experimental study with six popular mobile applications, we observed that the battery of the mobile device can last up to approximately an extra hour if the applications are developed with energy-aware practices. This work paves the way for a set of guidelines for energy-aware automatic refactoring techniques.

5.1 Introduction

A number of tools have been designed to help developers build high-performance Android apps [Hecht et al., 2015; Palomba et al., 2017]. For instance, *Google* ships the static analysis tool *lint* as part of the Android SDK. The tool detects typical structural problems that may lie in the codebases of Android applications — also known as **anti-patterns** or **code smells**.

A common anti-pattern is *Overdraw*, as illustrated in Figure 5.1. UI views in Android are described using several nested layouts that will be drawn on top of each other. If each of these layouts has a background color, the same pixel has to be drawn several times on each refresh. This leads to extraneous graphics processing, which is a potential source of unnecessary energy consumption [Pinto et al., 2014].

In this chapter, we analyze the impact of eight Android performance-based anti-patterns in terms of energy consumption in real, mature Android applications.

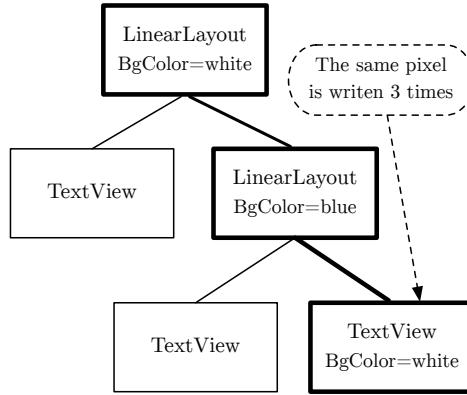


Figure 5.1: Example of a tree with the hierarchy of UI components in an Android app. Pixels in the white *TextView* had to be painted with white and blue and finally white.

Aiming at providing a methodology to help developing energy efficient Android applications, this work answers the following research questions:

Research Question 5.1

Do best practices for performance improvement also improve energy efficiency?

Research Question 5.2

Do these best practices actually have an impact on real, mature Android applications?

The main contributions of this work are:

- We provide a set of best practices to develop energy efficient mobile applications.
- We study and discuss the impact of each practice on energy consumption.
- We pave the way for a toolset to automatically detect and refactor energy inefficiencies in mobile applications.

In particular, the takeaway message of this chapter is

Fixing anti-patterns, viz. ViewHolder, DrawAllocation, WakeLock, ObsoleteLayoutParam and Recycle, lead to more energy-efficient mobile application, saving up to an hour of battery life.

As a side contribution, and to foster reproducibility, a benchmark suite containing all subjects and test suites used in the experiments is freely available from <https://www.github.com/luiscruz/greenbenchmark>.

This chapter is structured as follows. Section 5.2 describes how the experiments were conducted and the tools that were used. Section 5.3 presents the achieved results, followed by a discussion in Section 5.4. Threats to the validity are presented in Section 5.5. Finally, we present the related work in Section 5.6 and summarize main contributions in Section 5.7.

5.2 Empirical Study

We conducted a study in which energy consumption was our dependent variable, while Android optimizations were the independent variable. In this section we describe our methodology and empirical study:

- A. Android application selection
- B. Static analysis and refactoring
- C. Generation of automatic UI tests
- D. Energy measurement tools setup
- E. Experiments execution
- F. Data analysis

5.2.1 Android Application Selection

The technique proposed in this work, hence the empirical evaluation, needs the source code of the applications under analysis. For that matter, we use *F-droid*, a free and open source software catalog of Android applications. Currently, *F-droid* offers over 2,300 open source applications¹. The *Google Play Store* was used to obtain further details about the popularity of the application in the Android community. Applications were selected according to the following criteria: 1) open source, 2) active development, 3) not using heavy network operations, since we are not optimizing them. Applications were randomly selected and filtered when no performance issue was found by *lint*. Given the complexity of the experiments for each application we have limited our study to six applications:

Loop - Habit Tracker An application to track habits. Users' rating for version 1.4.1 is 4.7 out of 5.

Writeily Pro A note editor with markdown support. Users' rating for version 1.3.2 is 4.3 out of 5.

Talalarimo Alarm Clock A minimalist alarm clock. Users' rating for version 3.9 is 4.4 out of 5.

¹https://f-droid.org/wiki/page/Repository_Maintenance visited in June 5, 2020

Table 5.1: Metrics of applications used in experiments.

Application	Installs	Rating	# ratings	LOC (Java)	LOC (XML)	Classes	CC
Loop - Habit Tracker	50,000–100,000	★★★★★ 4.7	1,252	28,295	7,302	193	2,471
Writeily Pro	1,000–5,000	★★★★★ 4.3	84	3,251	2,612	86	498
Talalarimo Alarm Clock	1,000–5,000	★★★★★ 4.4	63	1,043	192	26	131
GnuCash	50,000–100,000	★★★★★ 4.3	2,460	26,532	20,757	286	2,846
Acrylic Paint	n.a.	n.a.	n.a.	961	384	18	119
Simple Gallery	1,000–5,000	★★★★★ 4.7	18	2,227	685	37	434

GnuCash A finance application to keep track of personal expenses. Rating for version 2.0.7 is 4.3.

Acrylic Paint A basic drawing application. This application is only available through the *F-droid* application store.

Simple Gallery Application to view pictures stored in the mobile device. Rating for version 1.15 is 4.7.

Table 5.1 shows information regarding the complexity of the applications as well as statistics from Google Play store. It presents number of installs, rating, and number of users that rated the application at Google Play Store, as well as **Lines of Code (LOC)** in Java, LOC in XML, number of classes, and McCabe's Cyclomatic Complexity (CC). *Loop - Habit Tracker* and *GnuCash* are the most complex applications, with over 25,000 LOC. Besides having a large number of LOC in Java, *GnuCash* also has a large number of LOC in XML, more than 20,000, which in Android is used for specification of the UI and other resources. It also has the highest CC value, 2,846. *Talalarimo Alarm Clock* is the simplest application with approximately 1,000 lines of Java code, 1,043 of XML, and a CC of 131. This was expected since it provides a small set of features.

5.2.2 Static Analysis and Refactoring

In order to measure the impact of performance-based guidelines in Android applications it is necessary to systematically detect parts of code that did not comply with those guidelines. We perform static code analysis to automatically detect code smells, such as *Overdraw*, mentioned in Section 5.1. The Android SDK provides a tool for this purpose, *lint*², which detects problems related with the structural quality of the code.

Code smells were chosen by considering performance-related suggestions given by *lint* that (1) are common in Android applications, and (2) potentially modify important parts of the application to fix the problem. Eight patterns resulted from this selection. Below we detail the eight patterns, including a rough estimation of priority provided by *lint* documentation. In *lint*, priority is an integer between 1 and

²*Lint* documentation: <http://developer.android.com/tools/debugging/improving-w-lint.html> (Visited on June 5, 2020).

10, with 10 being the most important — this is merely used to sort issues relative to each other.

DrawAllocation: Allocations within drawing code It is a bad practice allocating objects during a drawing or layout operation. Allocating objects can cause garbage collection operations that will slow down the operation and create a nonsmooth UI. The recommended fix is allocating objects upfront and reusing them for each drawing operation. *Lint priority:*  9/10.

WakeLock: Incorrect wake lock usage Wake locks are mechanisms to control the power state of the mobile device. This can be used to wake up the screen or the CPU when the device is in a sleep state in order to perform tasks. If an application fails to release a wake lock or uses it without being strictly necessary, it can drain the battery. As an example, some applications use a wake lock to keep the screen on. This requires developers to properly release the wake lock when it is no longer necessary. Alternatively, the application can set the flag FLAG_KEEP_SCREEN_ON and the system will properly manage the wake lock, being less prone to errors. *Lint priority:*  9/10.

Recycle: Missing recycle() calls There are collections such as TypedArray that are implemented using singleton resources. Thus, they should be released so that calls to different TypedArray objects can efficiently use these same resources. *Lint priority:*  7/10.

ObsoleteLayoutParams: Obsolete layout params During development, UI views might be refactored several times. In this process, some parameters might be left unchanged even when they have no effect on the view. This causes useless attribute processing at runtime. *Lint priority:*  6/10.

ViewHolder: View Holder Candidates This pattern is used to make a smoother scroll in *List Views*. When in a *List View*, the system has to draw each item. To make this process more efficient, data from the previous drawn item can be reused. The number of calls to the method `findViewById`, which is known for being a very expensive method, decreases with this technique. *Lint priority:*  5/10.

Overdraw: Painting regions more than once Another common inefficiency in Android applications is when views are being overdrawn. This means that the same pixel has to be written several times, leading to unnecessary processing (see Figure 5.1). This can be improved by removing the background of views, or by clipping drawing, when possible. The recommended fix is adding a statement in the view creation that removes the background of the parent view:

`getWindow().setBackgroundDrawable(null);` *Lint priority:*  3/10.

Table 5.2: Anti-patterns found in open source applications.

Anti-Pattern	Loop - Habit Tracker	Writeily Pro	Talalarmo	GnuCash	Acrylic Paint	Simple Gallery
DrawAllocation	n.a.	n.a.	2 □ 8+ 3-	n.a.	3 □ 7+ 2-	n.a.
WakeLock	n.a.	n.a.	1 □ 11+ 4-	n.a.	n.a.	n.a.
Recycle	n.a.	n.a.	n.a.	1 □ 1+	n.a.	n.a.
ObsoleteLayoutParams	n.a.	n.a.	n.a.	2 □ 2-	n.a.	n.a.
ViewHolder	n.a.	1 □ 37+ 21-	n.a.	n.a.	n.a.	n.a.
Overdraw	5 □ 5+ 2-	3 □ 7+ 8-	n.a.	n.a.	3 □ 10+ 7-	4 □ 6+ 3-
UnusedResources	3 □ 0+ 231-	67 □ 1+ 318-	n.a.	n.a.	n.a.	n.a.
UselessParent	n.a.	2 □ 3+ 14-	n.a.	n.a.	n.a.	n.a.

Each refactoring is reported with the number of files changed (□), number of insertions (+), and number of deletions (-). A “n.a.” is present when a given anti pattern was not found in the application.

UnusedResources Resources, such as icons or UI elements, may become obsolete due to changes in the software. However, developers may forget to remove them from the project which makes applications larger, consequently slowing down builds. *Lint* priority:  3/10.

UselessParent: Useless parent layout Since interface layouts suffer several changes throughout the development process, layouts frequently become useless. The latter can eventually be replaced by a descendant layout. *Lint* priority:  2/10.

Static code analysis is performed to automatically detect these patterns in the applications. For each detected pattern, the application was manually refactored and a new version of the application was produced. In addition, a version complying with all the practices was also created. For the sake of comparison, the original version was also used during the experiments. Table 5.2 shows the anti-patterns that were found in each of the analyzed applications. *Writeily Pro* resulted in five new versions, *Simple Gallery* in one version, and the others in three versions.

5.2.3 Generation of Automatic UI tests

The energy consumption of a mobile phone while running an application depends on several conditions (e.g., services that are running in the device, background tasks). To obtain meaningful results, the same execution needs to be replicated several times. The applications used in our study, unfortunately, do not provide test suites that mimic user interaction with the app. Thus, automatic UI test scripts were manually created to replicate user interaction in these applications.

The scripts were built using the *Python* library *Android View Client*³. This library allows the interaction with UI components by querying a view id, description or content, which makes tests compatible across different devices.

⚠ As seen in Chapter 3, *Android View Client* is not the most advisable framework for energy measurements. However, at the time this study was conducted, there was no

³<https://github.com/dtmilano/AndroidViewClient> visited in June 5, 2020.

evidence of this limitation. Still, in this case, as pointed out in Chapter 3, results were not affected by the poor-choice of UI testing framework.

Tests mimic the usual interaction of a user. Algorithms 1 to 6 describe the interaction for the applications *Loop - Habit Tracker*, *Writeily Pro*, *Talalarmo*, *GnuCash*, *Acrylic Paint*, *Simple Gallery*, respectively.

Algorithm 1 *Loop - Habit Tracker* interaction script

```
1: SkipIntroductoryTips()
2: for  $i \leftarrow 1$  to 10 do
3:   for  $i \leftarrow 1$  to 7 do
4:     CreateNewHabit( $i$ )
5:     CheckHabitDetails( $i$ )
6:     ScrollThroughTheReport()
7:     GoBack()
8:   end for
9:   DeleteAllHabits()
10: end for
```

Algorithm 2 *Writeily Pro* interaction script

```
1: GoToSettings()
2: GoBack()
3: for  $i \leftarrow 1$  to 20 do
4:   folderOne  $\leftarrow$  CreateFolderWithFoldersInside()
5:   folderTwo  $\leftarrow$  CreateFolderWithNotesInside()
6:   MoveAllNotesToFirstFolder()
7:   CreateNote()
8:   folderThree  $\leftarrow$  CreateFolder()
9:   MoveItemToFolder(folderOne, folderThree)
10:  DeleteFolder(folderThree)                                ▷ Removes all files
11: end for
```

Algorithm 3 *Talalarmo* interaction script

```
1: SetAlarmOn()                                              ▷ Starts next minute tick
2: Sleep(5.minutes)
3: StopAlarm()
4: for  $i \leftarrow 1$  to 200 do
5:   SwitchAMAndPM()
6: end for
7: for  $i \leftarrow 1$  to 12 do
8:   SetAlarmOn()
9:   SetAlarmOff()
10:  SwitchAMAndPM()
11:  GoToSettings()
12:  SwitchBetweenDarkAndLightTheme()
13:  GoBack()
14: end for
```

For each execution of the test, the application was uninstalled and installed with the APK of the version under analysis. Thus, all user data was erased at the beginning of the experiment, making sure each execution of the test would have a similar initial

Algorithm 4 *GnuCash* interaction script

```
1: SkipIntroductionSteps()
2: for  $i \leftarrow 1$  to 10 do
3:   for all  $account \in \{\text{"Assets"}, \text{"Equity"}\}$  do
4:     for  $i \leftarrow 1$  to 20 do
5:       SelectAccount( $account$ )
6:       EditAccount()
7:       GoBack()
8:     end for
9:   end for
10: end for
```

Algorithm 5 *Acrylic Paint* interaction script

```
1: SkipIntroduction()
2: for  $i \leftarrow 1$  to 20 do
3:   for  $i \leftarrow 1$  to 10 do
4:     DrawLine()
5:   end for
6:   GoToColorMenu()
7:   for  $i \leftarrow 1$  to 10 do
8:     SetColor()
9:   end for
10:  GoBack()
11: end for
```

Algorithm 6 *Simple Gallery* interaction script

```
1: for  $i \leftarrow 1$  to 100 do
2:   SelectAlbum()
3:   SelectPicture()
4:   GoBack()
5:   GoBack()
6: end for
```

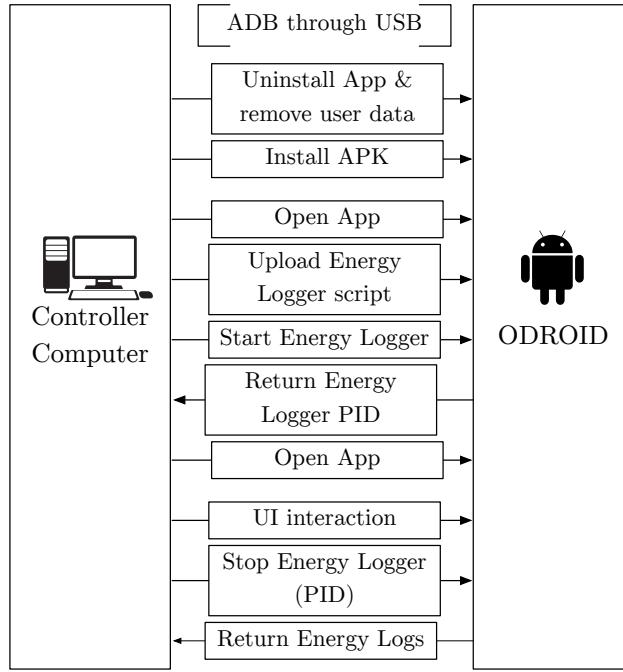


Figure 5.2: Experiments' workflow.

state. On the other hand, cleaning user data requires the application to setup in every experiment. This initial setup is not a real use case scenario, since it would happen only once after installing the app. To ensure that such scenario does not have a significant impact on results, we repeat subsequent scenarios a reasonable number of times — between 10 and 200 — depending on the complexity of the interaction.

5.2.4 Energy measurement tools setup

To measure the energy consumed in each execution, we use the bare-board computer *ODROID-XU* running Android version 4.2.2 - API level 17.

This device is known for having an architecture similar to a smartphone. Components such as cellular, location, accelerometer, and screen can be separately integrated. Nevertheless, these components are not being evaluated since the provided power sensors only report data for the main CPU, the secondary CPU, memory, and GPU.

Power sensors provide data with a sample period of 263,808 microseconds. Since the clock provided by *ODROID* in this setup has a precision of one second, data was down sampled. I.e., different samples with the same timestamp were aggregated using the average to a period of one second.

5.2.5 Experiments Execution

Each experiment was designed to be independent of previous experiments. The execution of a single experiment is illustrated by Figure 5.2. In every experiment, before running the UI interaction script, the energy logger is uploaded to the *ODROID* and set ready to start. The application, if existing, is uninstalled, the given APK is installed and finally the application is automatically opened.

After the execution of the UI interaction script, the energy logger is stopped and the data is collected from *ODROID* storage. This process is repeated 30 times for each different version of the application, as recommended in Chapter 2.

In addition, we measured interaction scripts with a blank application that we developed. The application does nothing and aims to give an idea of the energy consumed with the same UI interaction when the application is in an idle state. This gives an approximate measure of the overhead of energy consumed by the experiment setup and by the Android framework.

5.2.6 Data Analysis

Downsampling As described in Section 5.2.4, data was downsampled to one sample per second in order to synchronize energy logs with *ODROID* timestamps.

Outlier Removal Execution of experiments is prone to failures. This can happen due to a system dialog that popped up during the experiment, or due to a nontrivial bug that stopped the application, or to a slower response of the application that was not expected by the test script. Thus, there are executions that consumed considerably more or less energy. In order to reduce the effect of outliers, experiments with energy consumption outside the range $[\bar{x} - 2s, \bar{x} + 2s]$, where \bar{x} is the sample mean and s is the sample standard deviation, were discarded.

5.3 Results

In the end, 18 different APKs were tested with a total of 900 executions. It took 94 hours (roughly 4 days) and 75MB of raw data was collected.

For each app, Table 5.3 presents the sample size (n), i.e., the number of executions of the interaction script after outlier removal, mean (\bar{x}) and standard deviation (s) of energy consumption, and the p -value for the Shapiro-Wilk test. Shapiro-Wilk test for normality is a statistical test for detecting if the experiments follow a normal distribution. Column *Pattern* expresses the code smell that is fixed in that particular fixed version. *Original* is a version of the application that was not modified, serving as baseline. *All* stands for a version of the application in which all code smells

Table 5.3: Descriptive statistics of experiments

Application	Pattern	n	\bar{x} (J)	s	p-value
Loop - Habit Tracker	Original	28	335.6	35.3	0.76
	Overdraw	29	340.8	34.4	0.34
	UnusedResources	30	343.1	32.9	0.17
	All	29	336.4	34.7	0.58
	Blank	29	86.7	1.5	0.31
Writeily Pro	Original	30	119.7	7.2	0.42
	Overdraw	30	119.8	6.4	0.63
	UnusedResources	30	119.7	6.4	0.43
	ViewHolder	30	114.3	6.7	0.16
	UselessParent	30	119.3	7.1	0.10
Talalarimo	All	30	114.2	7.2	0.06
	Blank	30	87.2	9.9	0.20
	Original	29	58.2	0.8	0.28
	DrawAllocation	28	57.3	0.8	0.29
	WakeLock	28	57.4	0.7	0.59
GnuCash	All	29	57.7	0.9	0.53
	Blank	29	41.8	0.5	0.78
	Original	30	195.6	2.3	0.40
	ObsoleteLayoutParams	29	194.1	2.5	0.89
	Recycle	28	194.3	1.4	0.32
Acrylic Paint	All	30	194.0	2.5	0.89
	Blank	29	71.4	0.8	0.56
	Original	30	62.7	1.0	0.68
	DrawAllocation	29	62.5	1.1	0.12
	Overdraw	28	64.1	0.7	0.36
Simple Gallery	All	29	64.0	0.8	0.85
	Blank	28	52.9	0.6	0.71
	Original	30	145.9	3.7	0.28
Simple Gallery	Overdraw	29	149.0	1.9	0.88
	Blank	29	45.1	0.6	0.71

were fixed. Results for the executions using the blank application, described in Section 5.2.5, are also shown.

Figures 5.3 to 5.8 plot the results for each of the tested applications. As the violin plots show a bell-shaped curve and Shapiro-Wilk test's p-value is greater than 0.05, we conclude that data follows a normal distribution.

To validate changes in energy consumption, we tested the following hypotheses:

$$H_0 : \mu_{fixed} - \mu_{original} = 0$$

$$H_1 : \mu_{fixed} - \mu_{original} \neq 0$$

where $\mu_{original}$ stands for the mean of the energy consumption for the original version, and μ_{fixed} of the fixed versions. Considering the samples as independent

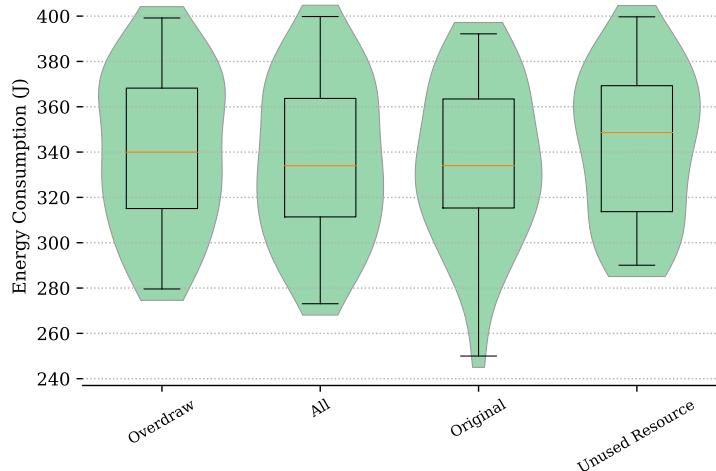


Figure 5.3: Energy consumption for *Loop - Habit Tracker*.

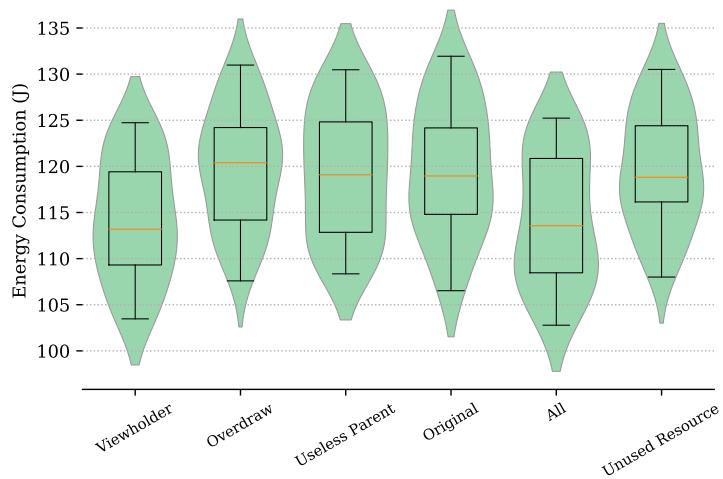


Figure 5.4: Energy consumption for *Writeily Pro*.

and since we have non-paired data in which the standard deviation of populations is not known, we used Welch's two-sample t-test as the most appropriate test for our analysis. A two-tail p -value was used with the critical value of $\alpha = 0.05$. Results are shown in Table 5.4.

Table 5.5 presents the effect size results for the patterns with significant impact on energy. It presents the mean difference (MD) ($\bar{x}_{fixed} - \bar{x}_{original}$), Cohen's d ($\frac{\bar{x}_{fixed} - \bar{x}_{original}}{s}$), a method to indicate a standardized difference between means, improvement (IMP) compared to the original consumption ($\frac{\bar{x}_{fixed} - \bar{x}_{original}}{\bar{x}_{original}}$), and the column *Savings*, which provides the number of minutes of battery life saved after repeating the same usage of the application during 24 hours.

For example, the application *GnuCash* in Table 5.3 is assigned to five rows, each for a different tested version of the application and for the blank application. The

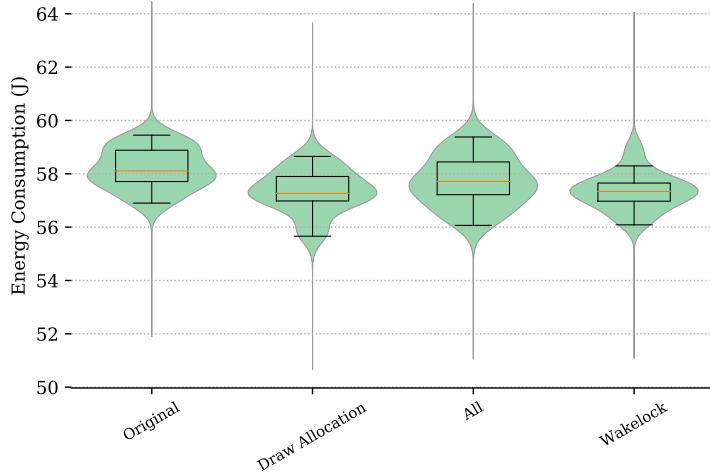


Figure 5.5: Energy consumption for *Talalarmo*.

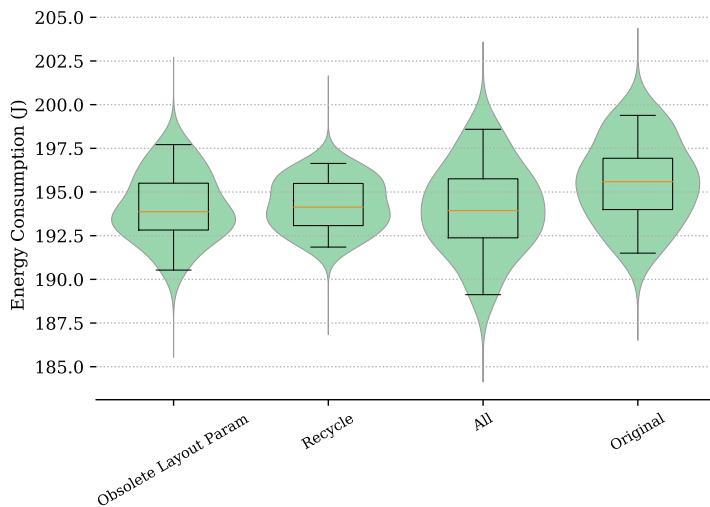


Figure 5.6: Energy consumption for *GnuCash*.

fixed version for the *Recycle* pattern has 28 experiments (n) which on average (\bar{x}) consumed 194.3J with a standard deviation (s) of 1.44 and a p -value for the normality test of 0.32. Significance tests presented in Table 5.4, show that this version of *GnuCash* can significantly reduce energy consumption, since the p -value obtained with the Welch's t-test is 0.0140 which is lower than our significance level $\alpha = 0.05$. All fixed versions that passed the significance level were reported in Table 5.5. The table shows that this version of the application provided an MD of -1.28J , which means that it saved 1.28J , providing an improvement of 0.65% over the original version. This means that after 24 hours of using the app, the battery could last approximately 9 more minutes. The same analysis can be made with the other applications.

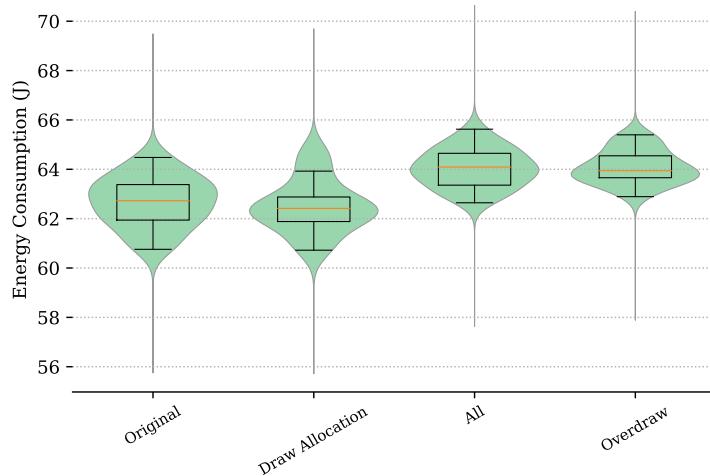


Figure 5.7: Energy consumption for *Acrylic Paint*.

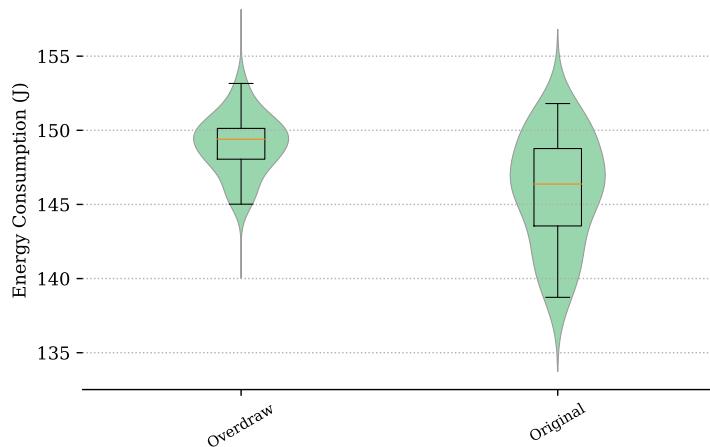


Figure 5.8: Energy consumption for *Simple Gallery*.

5.4 Discussion

Results show that *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParams*, and *Recycle* are patterns that need to be taken into account to develop an energy-efficient mobile application. *ViewHolder* is the pattern with the greatest impact, with an improvement of approximately 5% in *Writeily Pro*. The original version consumed 119.7J while it consumed 114.3J after being modified. This translates into 65 minutes of savings after 1 day of usage (see Table 5.5). When considering a usage of 3.75 hours, this would translate in extra 10 minutes, without affecting user experience.

DrawAllocation also provides an interesting improvement. Although it occurred in a tiny part of the user interaction in *Talalarimo*, we observed an improvement of 1%. *DrawAllocation* was also tested with *Acrylic Paint* but it did not have a statistically significant improvement. The fix affected the color picker redraw routine. Using the

Table 5.4: Significance Welch's t-test results.

Application	Pattern	Test	p-value
Loop - Habit Tracker	Overdraw	-0.56	0.5784
	UnusedResources	-0.83	0.4121
	All	-0.08	0.9362
Writeily Pro	Overdraw	-0.10	0.9180
	UnusedResources	-0.03	0.9790
	ViewHolder	3.02	0.0038
Talalarmo	UselessParent	0.20	0.8434
	All	2.93	0.0049
	DrawAllocation	4.18	0.0001
GnuCash	WakeLock	4.43	< 0.0001
	All	2.16	0.0353
	ObsoleteLayoutParams	2.57	0.0127
AcrylicPaint	Recycle	2.55	0.0140
	All	2.47	0.0164
	DrawAllocation	0.64	0.5221
Simple Gallery	Overdraw	45.88	< 0.0001
	All	-5.84	< 0.0001
	Overdraw	-4.04	0.0010

Table 5.5: Effect size of significant patterns.

Application	Pattern	MD	Cohen's d	IMP (%)	Savings (min)
Writeily Pro	ViewHolder	↓ -5.39	-0.78	4.50	65
	All	↓ -5.42	-0.76	4.53	65
Talalarmo	DrawAllocation	↓ -0.86	-1.11	1.47	21
	WakeLock	↓ -0.85	-1.17	1.46	21
	All	↓ -0.48	-0.57	0.82	12
GnuCash	ObsoleteLayoutParams	↓ -1.41	-0.67	0.72	10
	Recycle	↓ -1.28	-0.66	0.65	9
	All	↓ -1.53	-0.64	0.78	11
Acrylic Paint	Overdraw	↑ 1.42	1.64	-2.26	-33
	All	↑ 1.37	1.51	-2.18	-31
Simple Gallery	Overdraw	↑ 3.08	1.04	-2.11	-30

Android developer options to debug view updates, we can see that redraw is only happening a single time when a new color is chosen. The impact of this fix in the overall execution was minimal, which might have been the reason for not having significant changes in energy consumption.

Fixing incorrect *WakeLock* usage also provided an improvement of 1%. In the original version of *Talalarmo*, the wake lock was not being properly released which could have lead to energy drain in particular cases. For instance, when the application is no longer in the alarm mode and the wake lock was not properly released, the device cannot activate a lower power state. This would consume energy unnecessarily but, given the nature of our tests, such a scenario is not being effectively tested. Thus, the effect size is expected to be higher in a real case scenario.

ObsoleteLayoutParams and *Recycle* anti-patterns were found in the application *GnuCash*. Although results showed a small effect size, with improvements of less than

1% (see Table 5.5), they were statistically significant, as shown in Table 5.4. After analyzing the changes made to fix *ObsoleteLayoutParams*, we saw that it only required removing two obsolete view attributes. One in a list view and another in a list item. Thus, a big effect was not expected from fixing this anti-pattern. Still, for a more energy-efficient practice, this issue should be considered. UI changes during the application development and obsolete attributes can be easily forgotten. This is a common issue, since it does not affect the UI appearance. Regarding *Recycle*, the issue occurred when accessing the color of an account. *GnuCash* allows the user to create several accounts. Each account can be customized with a different color. To get the account's color options a *TypedArray* needs to be accessed. The issue lay in the fact that *TypedArray* was not being closed after it had been accessed. This only happens when a user opens the settings of an account. Regardless, it was able to have significant impact on energy consumption, according to the results of the Welch's t-test in Table 5.4.

Surprisingly, after fixing *Overdraw*, applications ended up consuming more energy. Although *Overdraw* can create a laggy UI, fixing it can lead to more energy consumption. In the applications *Acrylic Paint* and *Simple Gallery*, it decreased battery life approximately 30 minutes after one day of usage. Having a simple UI layout hierarchy is always a good practice, but adding extra code to avoid *Overdraw* requires processing, which might not be worth it, depending on the scenario.

When the application has a view that remains active for a considerable amount of time, this view will have to redraw itself several times. In this case, having an efficient redraw is important, and fixing *Overdraw* is expected to create interesting results. On the other hand, if a view is being created several times but does not remain active for a reasonable amount of time, fixing *Overdraw* might be creating an unnecessary overhead during the creation of the view. Since modeling user behavior is not a trivial task, in our experiments the time a user spends in a view is not being considered. Thus, views with long lifetime were not explored.

UnusedResources and *UselessParent* did not have any significant effect, as showed by the Welch's t-test in Table 5.4. *UnusedResources* was tested in the applications *Loop - Habit Tracker* and *Writeily Pro*. Having unused resources in the application can increase build time, APK size, and complexity of project maintenance. Thus, it is still an anti-pattern to be considered, although it does not affect energy consumption. *UselessParent* was tested in the application *Writeily Pro*. Since the test is focusing in common use case scenarios rather than a particular anti-pattern, it is possible that *UselessParent* was not a relevant issue in this scenario. In other words, although the optimization was not necessary in this particular case, it may be useful in other applications and scenarios.

It is interesting to note that in a few cases, improvements were higher after fixing a single anti-pattern than after fixing all of them (e.g., *Talalarimo*). The main reason

for this lies in the fact that experiences are prone to random variations related with the power meter and the mobile device. Thus, results may change from experiment to experiment, and effect size measures are not very precise. Nevertheless, this does not affect statistical significance, which shows that energy consumption reduces after using these patterns.

The results for the blank application show that in some experiments a great part of energy is consumed in the experimental setup. Analyzing Table 5.3, regarding the application *Acrylic Paint*, the interaction script running with the original version consumed on average 62.68J, whereas on the blank application consumed 52.93J on average. This means that the interaction script consumed 84% of the total energy consumption, leaving only 16% for optimization. The least affected application by the interaction script was *Loop - Habit Tracker*, consuming only 26% of the total energy consumption.

Research Question 5.1

Do best practices for performance improvement also improve energy efficiency?

Our results show that there are performance optimizations that also have an impact on energy consumption. Concretely, energy efficiency was improved after fixing *ViewHolder* (4.5%), *DrawAllocation* (1.5%), *WakeLock* (1.5%), *ObsoleteLayoutParams* (0.7%), and *Recycle* (0.7%). Developers should consider performance anti-patterns when developing energy-efficient applications. This does not conform with previous work [Sahin et al., 2016], which found that mainly because we have studied Android specific optimizations. However, *UnusedResources* and *UselessParent* did not provide any significant change in energy consumption, while *Overdraw* was found to consume more energy (2.2%). Nevertheless, the impact of these patterns depends on the case in which it is being applied, as it was shown with the *Overdraw* pattern. Different applications and use cases might have better or worse results. Further research need to be made to identify optimal or worst-case scenarios for these patterns.

Optimizations analyzed in this work do not affect the feature set of the application. This means that they can be applied without having to deal with tradeoffs between user experience and energy consumption. Furthermore, the instrumentation used for the applications in this study did not require previous knowledge about the project.

Research Question 5.2

Do these best practices actually have an impact on real, mature Android applications?

Six open source Android applications were included in this study. They are available on F-Droid and, with the exception of *Acrylic Paint*, they can also be downloaded

from *Google Play Store*. From these six, we were able to improve energy efficiency in three applications. We observed improvements in *Writeily Pro* (4.5%), *Talalarimo* (1.4%), and *GnuCash* (0.8%). This evidence suggests that any application with patterns *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParams*, and *Recycle* can have improvements in energy efficiency. More applications should however be evaluated to corroborate with this intuition. As said in the Introduction, the test cases used in the study are available online to foster reproducibility.

5.5 Threats to the Validity

Internal Validity The validation of the code changes that were applied in the Android apps in this study is limited to our perception of the application's features and to the impact we expect from our code changes.

Results obtained from energy measurements can be affected by several factors that are not easily controlled. Different devices and different versions of Android may respond in a different way to these optimizations. Background tasks performed by other applications affect energy consumption and are hard to control. Although the performed outlier removal intends to discard these cases, some of them may still have impacted some experiments.

In addition, we did not measure energy consumption of the whole device. Other components such as GPS and accelerometer are known to have a significant impact on energy consumption. The same happens with network operations and screen usage. However, the later is expected to have the same energy consumption in all cases. The automatic interaction with the UI, used in our experiments, may also create overhead on the energy spent. This might affect results in terms of effect size but we do not expect it to affect statistic significance.

External factors, such as temperature can also affect experiment results. If a CPU has a higher temperature, it will consume more energy. However, if a CPU has to do more processing, it will increase its temperature, which means that temperature can also be an effect and not the cause. We have repeated 30 times the experiment for each fixed version in order to have a fairer comparison, although this can still be a threat. In addition, the experiments were alternatively executed with other versions of a given app to ensure similar external factors for all versions.

External Validity Patterns were studied in a small set of applications (six). Since the impact of these patterns heavily depends upon the case in which it is being applied, further experiments should be conducted to understand overall effects. Nevertheless, this study is an initial exploration of the impact of anti-patterns in energy consumption.

5.6 Related Work

Previous work showed that there is a large set of code smells that are appearing in Android applications [Mannan et al., 2016; Hecht et al., 2015]. The analysis was performed by checking the application’s source code, bytecode, and metadata but no refactor was performed, and the impact on energy consumption was not studied.

Energy efficiency might be improved by offloading heavy tasks to the cloud [Kwon and Tilevich, 2015; Qian and Andresen, 2015; D. Li and Halfond, 2015]. Network components often lead to high energy consumption but depending on the complexity of the task, there is a tradeoff between CPU and network operations. Making such optimization would require structural changes in applications. In this chapter, we do not follow the same principles and network operations were not studied.

Previous work identified common energy-greedy sequences of Android API calls in 55 mobile applications [Linares-Vásquez et al., 2014]. Most energy-greedy API calls were found to be related to UI manipulation and database tasks. We take another step further by identifying alternative sequences that might lead to less energy consumption.

Other approaches have used visualization tools to help developers spot high energy-consuming I/O events, guiding developer to fix those events to reduce energy consumption [Pathak et al., 2011b]. Debugging energy-related defects has also been done by relating user reported defects with common energy-inefficient API calls patterns using log files [Banerjee et al., 2016]. Internet, Camera, Database, and UI operations were considered the most energy-intensive components in 405 real-world mobile applications [D. Li et al., 2014a]. Our approach lies on finding common inefficient patterns and providing standard optimizations. Ideally this can help developing energy-efficient applications before having to profile energy bundles.

Some works suggest changing the application’s feature set (e.g., reducing third-party advertisements, different UI colors in **Organic Light-Emitting Diode (OLED)** displays [Chen et al., 2013], etc.) as a solution to optimize energy consumption [Pathak et al., 2011b; Pathak et al., 2012a; D. Li et al., 2015; Linares-Vásquez et al., 2015]. Such optimizations have to be considered when designing an energy-efficient application. Nevertheless, our work intends to preserve original functionality of applications.

Energy efficiency has been improved at UI level by removing unnecessary display updates [Kim et al., 2015]. This approach is slightly different from ours since it requires modifications at the operative system level.

The idea of having energy code smells, plus refactoring fixes, for mobile applications has been studied before. Directed graphs were used to describe and analyze

applications at the source code level, and a graph repository query language was proposed to detect code smells [Gottschalk et al., 2012]. However, the effect of these code smells on energy consumption was not evaluated. Our work presents results of the optimization of mobile application along with statistical significance tests and effect sizes. Previous work reported that, depending on the application, code smells could have opposite impacts on energy consumption [Sahin et al., 2014]. However, as opposed to the work presented in this chapter, the study did not include mobile applications. Other work has used a similar approach to measure the impact of performance tips in Android applications [Sahin et al., 2016]. However, these tips focus on internal aspects of the way Java is assembled in Android, which is expected to have impact in heavy processing tasks rather than in normal android applications. Our work takes it to another level and focuses on the way applications consume the Android framework's APIs.

Previous works have studied the influence of the pattern *Internal Getter/Setter* [Sahin et al., 2016; D. Li and Halfond, 2014; Mundody and Sudarshan, 2014; Tonini et al., 2013; Hecht et al., 2016; Carette et al., 2017] but it has been reported by Google as not having any effect in performance since Android 2.3⁴⁵. Thus, we considered this optimization obsolete and left it out of our study.

Making an efficient use of the resources of a mobile device (e.g., GPS, Camera, Wifi) is a good way of saving energy. The use of these resources can be optimized by creating an **Event-flow Graph (EFG)** that represents UI states of a given Android app [Banerjee and Roychoudhury, 2016]. Defects are detected and refactored by matching expressions with a deterministic finite automaton based on the EFG. Although our work has similar goals, we have focused instead on code smells that directly affect CPU usage, including UI optimizations. In addition, significance tests were performed to consolidate the relevance of our results.

5.7 Summary

In this chapter, motivated by the existing efforts to build efficient mobile applications, we have measured the impact of eight performance-based optimizations on energy efficiency. In sum, in this chapter:

- We design a methodology to measure the impact of eight performance-based anti-patterns on the energy consumption of five real-world Android applications (cf. Section 5.2, page 79).
- We show that the anti-patterns *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParams*, and *Recycle* have a positive impact on the energy efficiency

⁴Clarification on the Android compiler's optimizations from a Google engineer: <http://stackoverflow.com/q/4912695/> (Visited on June 5, 2020).

⁵Description of the *Internal Getter/Setter* optimization and why it is disabled: <http://tools.android.com/tips/lint-checks> (Visited on June 5, 2020).

of Android apps (answering RQ 5.1, page 78). Mobile developers should take these code smells into account when building their mobile apps.

- We show that there is no evidence of the impact of the anti-patterns *Overdraw*, *UnusedResources*, and *UselessParent* in the energy consumption of mobile apps.
- We validate the impact of anti-patterns on the energy efficiency of five real-world mobile applications, using typical use case scenarios (answering RQ 5.2, page 78). With this approach we prevented exploiting the usage of anti-patterns in our experimentation without yielding any practical impact in a real context.
- We motivate the importance of having tools to automatically refactor mobile applications to avoid these code smells. Such tools would improve energy efficiency of already deployed mobile applications (as we will show later in Chapter 6).

Using Automatic Refactoring to Improve Energy Efficiency



Improving Energy Efficiency Through Automatic Refactoring

Luis Cruz and Rui Abreu

Submitted to Journal of Software Engineering Research and Development, 2019.



Using Automatic Refactoring to Improve Energy Efficiency of Android Apps

Luis Cruz and Rui Abreu

In: XXI Ibero-American Conference on Software Engineering (CIBSE, Best Paper Award), 2018.



Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring

Luis Cruz, Rui Abreu, and Jean-Noël Rouvignac

In: Proceedings of the 4th International Conference on Mobile Software Engineering and Systems, 2017.

Abstract

Although energy efficiency is a valuable requirement, developers often lack knowledge to deliver energy-efficient mobile applications. In this chapter, we study how automatic refactoring can aid developers ship energy-efficient apps. We leverage an automatic refactoring tool, Leafactor, with five energy code smells that tend to go unnoticed in Android applications. To evaluate Leafactor, we conduct an empirical study comprehending 140 free and open source apps. Results evince the importance of having tools to help developers adopt energy best practices mobile applications. Code smells in 45 apps were detected and fixed, from which 40% have been successfully merged into the official repositories.

6.1 Introduction

In the past decade, the advent of mobile devices has brought new challenges and paradigms to the existing computing models. One of the major challenges is the fact that mobile phones have limited battery life. As a consequence, users need to frequently charge their devices to prevent their inoperability. Hence, energy efficiency is an important non-functional requirement in mobile software, with a valuable impact on usability.

A study in 2013 reported that 18% of apps have feedback from users that is related to energy consumption [Wilke et al., 2013b]. Other studies have nonetheless found that most developers lack the knowledge about best practices for energy efficiency in mobile applications (apps) [Pang et al., 2016; Sahin et al., 2014]. Hence, it is important to provide developers with actionable documentation and toolsets that aim to help deliver energy-efficient apps.

In Chapter 5, we have identified code optimizations with significant impact on the energy consumption of Android apps. Five code optimizations were found to yield a significant improvement in the energy consumption of mobile apps: *ViewHolder*, *Draw Allocation*, *Wake Lock*, *Recycle*, and *Obsolete Layout Parameter*. However, certify that code is complying with these optimizations is time-consuming and prone to errors. Thus, in this chapter we study how automatic refactor can help develop code that follows energy best practices.

There are state-of-the-art tools that provide automatic refactoring for Android and Java apps (for instance, *AutoRefactor*¹, *Walkmod*², *Facebook pfff*³, *Kadabra*⁴). Although these tools help developers create better code, they do not feature energy-related patterns for Android. Thus, we leverage five energy optimizations in an automatic refactoring tool, *Leafactor*, which is publicly available with an open source license. In addition, the toolset has the potential to serve as an educative tool to aid developers in understanding which practices can be used to improve energy efficiency.

In this chapter, we answer the following research questions:

Research Question 6.1

What is the prevalence of energy code smells in FOSS Android applications?

We analyze the prevalence of five energy code smells in a dataset of 140 FOSS Android apps. We have found that a considerable part (32%) is released with energy inefficiencies.

Research Question 6.2

Is automatic refactoring a feasible approach to improve the energy efficiency of mobile applications?

We study how an automatic refactoring tool would help ship more energy efficient mobile software. While applying *Leafactor* to our dataset of Android apps we have fixed 222 anti-patterns in 45 apps. We have used the results of our tool to

¹*AutoRefactor*: <http://autorefactor.org> (June 5, 2020).

²*Walkmod*: <http://walkmod.com> (June 5, 2020).

³*Facebook pfff*: <https://github.com/facebookarchive/pfff> (June 5, 2020).

⁴*Kadabra*: <http://specs.fe.up.pt/tools/kadabra/> (June 5, 2020).

contribute to projects of Android apps, validating the value of adopting an automatic refactoring tool in the development stack of mobile apps. In the end, we have successfully merged our changes into the official branch of 18 projects. Results show that automatic refactoring tools can be very helpful to improve the energy footprint of apps.

In sum, our work makes the following contributions:

- An automated refactoring tool, *Leafactor*, to improve energy efficiency of Android applications.
- An empirical study of the prevalence of five energy-related code smells in FOSS Android applications.
- The submission of 59 PRs to the official code bases of 45 FOSS Android applications, comprehending 222 energy efficiency refactorings.

The remainder of this chapter is organized as follows: Section 6.2 details energy refactorings and corresponding impact on energy consumption; in Section 6.3, we present the automatic refactor toolset that was implemented; Section 6.4 describes the experimental methodology used to validate our tool, followed by Sections 6.5 and 6.6 with results and discussion; in Section 6.7 we present the related work in this field; and finally Section 6.8 summarizes our findings and contributions.

6.2 Energy Refactorings

We use static code analysis and automatic refactoring to apply Android-specific optimizations of energy efficiency. In this section, we revisit the anti-patterns studied in Chapter 5 that yielded a significant improvement in the energy efficiency of Android apps. In addition to the expected energy efficiency improvement (leaf icon) and the *lint* priority, we provide code examples with step-by-step instructions that show how code refactoring can be applied to fix these code smells.

All refactorings are in Java with the exception *ObsoleteLayoutParams* which is in XML — the markup language used in Android to define the UI.

6.2.1 ViewHolder: View Holder Candidates

Energy efficiency improvement (leaf icon): 4.5%. Lint priority:  5/10.

This pattern is used to make a smoother scroll in *List Views*, with no lags. When in a *List View*, the system has to draw each item separately. To make this process more efficient, data from the previous drawn item should be reused. This technique decreases the number of calls to the method `findViewById()`, which is known for

being a very inefficient method [Linares-Vásquez et al., 2014]. The following code snippet provides an example of how to apply *ViewHolder*.

```
// ...
@Override
public View getView(final int position, View convertView, ViewGroup parent) {
    convertView = LayoutInflater.from(getContext()).inflate(❶
        R.layout.subforsublist, parent, false
    );
    final TextView t = ((TextView) convertView.findViewById(R.id.name)); ❷
// ...
```

Optimized version:

```
// ...
private static class ViewHolderItem { ❸
    private TextView t;
}

@Override
public View getView(final int position, View convertView, ViewGroup parent) {
    ViewHolderItem viewHolderItem;
    if (convertView == null) { ❹
        convertView = LayoutInflater.from(getContext()).inflate(
            R.layout.subforsublist, parent, false
        );
        viewHolderItem = new ViewHolderItem();
        viewHolderItem.t = ((TextView) convertView.findViewById(R.id.name));
        convertView.setTag(viewHolderItem);
    } else {
        viewHolderItem = (ViewHolderItem) convertView.getTag();
    }
    final TextView t = viewHolderItem.t; ❺
// ...
```

- ❶ In every iteration of the method `getView`, a new `LayoutInflater` object is instantiated, overwriting the method's parameter `convertView`.
- ❷ Each item in the list has a view to display text — a `TextView` object. This view is being fetched in every iteration, using the method `findViewById()`.
- ❸ A new class is created to cache common data between list items. It will be used to store the `TextView` object and prevent it from being fetched in every iteration.
- ❹ This block will run only in the first item of the list. Subsequent iterations will receive the `convertView` from parameters.
- ❺ It is no longer needed to call `findViewById()` to retrieve the `TextView` object.

One might argue that the version of the code after refactoring is considerably less intuitive. This is, in fact true, which might be a reason for developers to ignore optimizations. However, regardless of whether this optimization should be taken

care of by the system, it is the recommended approach, as stated in the Android official documentation⁵.

6.2.2 DrawAllocation: Allocations within drawing code

⌚ 1.5%. Lint priority: ||||| 9/10.

Draw operations are very sensitive to performance. It is a bad practice allocating objects during such operations since it can create noticeable lags. The recommended fix is allocating objects upfront and reusing them for each drawing operation, as shown in the following example:

```
public class DrawAllocationSampleTwo extends Button {
    public DrawAllocationSampleTwo(Context context) {
        super(context);
    }
    @Override
    protected void onDraw(android.graphics.Canvas canvas) {
        super.onDraw(canvas);
        Integer i = new Integer(5);❶
        // ...
        return;
    }
}
```

Optimized version:

```
public class DrawAllocationSampleTwo extends Button {
    public DrawAllocationSampleTwo(Context context) {
        super(context);
    }
    Integer i = new Integer(5);❷
    @Override
    protected void onDraw(android.graphics.Canvas canvas) {
        super.onDraw(canvas);
        // ...
        return;
    }
}
```

- ❶ A new instance of Integer is created in every execution of onDraw.
- ❷ The allocation of the instance of Integer is removed from the drawing operation and is now executed only once during the app execution.

6.2.3 WakeLock: Incorrect wakelock usage

⌚ 1.5%. Lint priority: ||||| 9/10.

⁵ViewHolder explanation in the official documentation: <https://developer.android.com/guide/topics/ui/layout/recyclerview> visited in June 5, 2020.

Wakelocks are mechanisms to control the power state of a mobile device. This can be used to prevent the screen or the CPU from entering a sleep state. If an application fails to release a wakelock or uses it without being strictly necessary, it can drain the battery of the device.

The following example shows an Activity that uses a wake lock:

```
extends Activity { private WakeLock wl;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    PowerManager pm = (PowerManager) this.getSystemService(
        Context.POWER_SERVICE
    );
    wl = pm.newWakeLock(
        PowerManager.SCREEN_DIM_WAKE_LOCK | PowerManager.ON_AFTER_RELEASE,
        "WakeLockSample"
    );
    wl.acquire();❶
}
}
```

- ❶ Using the method `acquire()` the app asks the device to stay on. Until further instruction, the device will be deprived of sleep.

Since no instruction is stopping this behavior, the device will not be able to enter a sleep mode. Although in exceptional cases this might be intentional, it should be fixed to prevent battery drain.

The recommended fix is to override the method `onPause()` in the activity:

```
//...
@Override protected void onPause(){
    super.onPause();
    if (wl != null && !wl.isHeld()) {
        wl.release();
    }
}
//...
```

With this solution, the lock is released before the app switches to background.

6.2.4 Recycle: Missing `recycle()` calls

 0.7%. Lint priority:  7/10.

There are collections such as `TypedArray` that are implemented using singleton resources. Hence, they should be released so that calls to different `TypedArray` objects can efficiently use these same resources. The same applies to other classes (e.g., database cursors, motion events, etc.).

The following snippet shows an object of TypedArray that is not being recycled after use:

```
public void wrong1(AttributeSet attrs, int defStyle) {
    final TypedArray a = getContext().obtainStyledAttributes(
        attrs, new int[] { 0 }, defStyle, 0
    );
    String example = a.getString(0);
}
```

Solution:

```
public void wrong1(AttributeSet attrs, int defStyle) {
    final TypedArray a = getContext().obtainStyledAttributes(
        attrs, new int[] { 0 }, defStyle, 0
    );
    String example = a.getString(0);
    if (a != null) {
        a.recycle();❶
    }
}
```

- ❶ Calling the method `recycle()` when the object is no longer needed, fixes the issue. The call is encapsulated in a conditional block for safety reasons.

6.2.5 `ObsoleteLayoutParams (OLP)`: Obsolete layout params

⌚ 0.7%. Lint priority:  6/10.

During development, UI views might be refactored several times. In this process, some parameters might be left unchanged even when they have no effect in the view. This causes useless attribute processing at runtime. As an example, consider the following code snippet (XML):

```
<LinearLayout>
    <TextView android:id="@+id/name"
              android:layout_width="wrap_content"
              android:layout_height="wrap_content"
              android:layout_alignParentBottom="true">❶
    </TextView>
</LinearLayout>
```

- ❶ The property `android:layout_alignParentBottom` is used for views inside a `RelativeLayout` to align the bottom edge of a view (i.e., the `TextView`, in this example) with the bottom edge of the `RelativeLayout`. On contrary, `LinearLayout` is not compatible with this property, having no effect in this example. It is safe to remove the property from the specification.

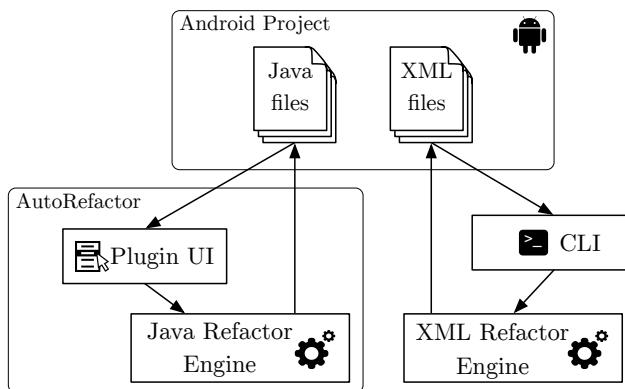


Figure 6.1: Architecture diagram of the automatic refactoring toolset.

6.3 Automatic Refactoring Tool

In the scope of our study, we developed a tool to statically analyze and transform code, implementing Android-specific energy efficiency refactorings — *Leafactor*. The toolset receives a single file, a package, or a whole Android project as input and looks for eligible files, i.e., Java or XML source files. It automatically analyzes those files and generates a new compilable and optimized version.

The architecture of *Leafactor* is depicted in Figure 6.1. There are two separate engines: one to handle Java files and another to handle XML files. The refactoring engine for Java is implemented as part of the open-source project *AutoRefactor* — an *Eclipse* plugin to automatically refactor Java code bases.

6.3.1 AutoRefactor

AutoRefactor is an *Eclipse* plugin that delivers automatic refactoring in Java codebases. It is created as a complement to existing static analyzers such as *SonarQube*, *FindBugs*, *CheckStyle* and *PMD*. Although they provide insightful warnings to developers, they do little in helping developers fixing all the issues lying in legacy codebases.

It provides a comprehensive set of 103 common code cleanups to help deliver “smaller, more maintainable and more expressive code bases”⁶. The list goes from simple rules, such as enforcing the use of the method `isEmpty()` to check whether a collection is empty, instead of checking its size (rule *IsEmptyRatherThanSize*), to more complex ones, such as *SetRatherThanList* choosing a more adequate collection type for specific use cases. In addition, *AutoRefactor* also supports cleanups for code comments, such as removing auto-generated or empty Javadocs from the codebase (rule named by *AutoRefactor* as *Comments*).

⁶As described in the *AutoRefactor*’s official website: <http://autorefactor.org> (Visited on June 5, 2020).

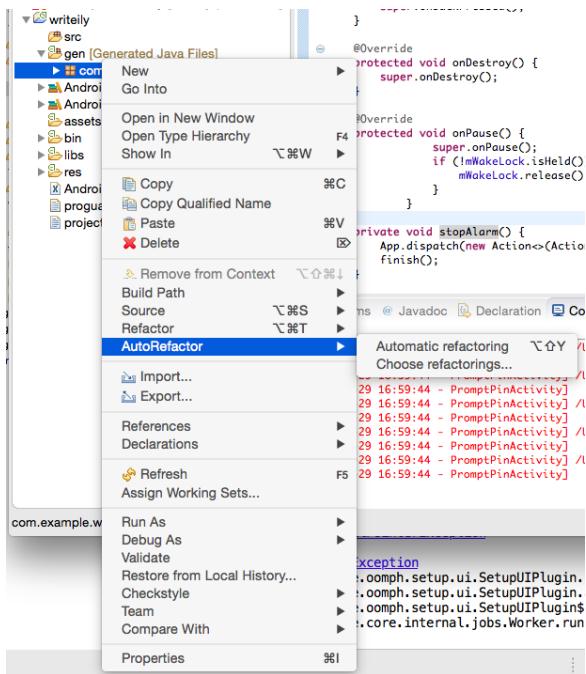


Figure 6.2: Developers can apply refactorings by selecting the “Automatic refactoring” option or by using the key combination $\text{⌘} + \text{⌘} + \text{Y}$.

*Eclipse Marketplace*⁷ — reported 4459 successful installs of *AutoRefactor*.

A common use case of *AutoRefactor* is presented in the screenshot of Figure 6.2. Developers can apply refactorings in single files, packages, or entire projects.

Under the hood, *AutoRefactor* integrates a handy and concise API to manipulate Java *Abstract Syntax Trees* (ASTs). We contributed to the project by implementing the Java refactorings mentioned in Section 6.2.

6.3.2 XML refactorings

Since XML refactorings are not supported by *AutoRefactor*, a separate refactoring engine was developed and integrated into *Leafactor*. The engine features a **Command Line Interface (CLI)**, that can be integrated with continuous integration environments. Optionally, the tool can be set to simply flag warnings, without performing any refactoring transformation. As detailed in the previous section, only a single XML refactoring is offered — *ObsoleteLayoutParams*.

6.4 Empirical evaluation

We designed an experiment with the following goals:

⁷*Eclipse Marketplace* — an interface for browsing and installing plugins for the Java **Integrated Development Environment (IDE)** Eclipse <https://marketplace.eclipse.org> (Visited on June 5, 2020).

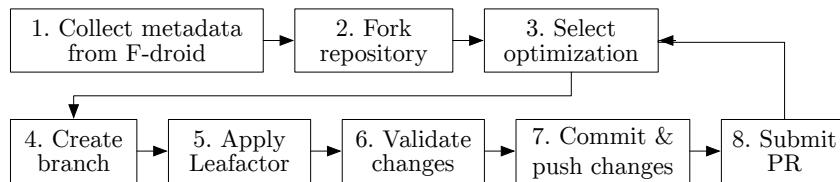


Figure 6.3: Experiment's procedure for a single app.

- Study the benefits of using an automatic refactoring tool within the Android development community.
- Study how FOSS Android apps are adopting energy efficiency optimizations.
- Improve energy efficiency of FOSS Android apps.

We adopted the procedure explained in Figure 6.3. Starting with step 1, we collect data from the *F-droid* app store⁸ — a catalog for FOSS applications for the Android platform. For each mobile application, we collect the git repository location which is used in step 2 to fork the repository and prepare it for a potential contribution to the project's official code repository. Following, in step 3 we select one refactoring to be applied and consequently initiate a process that is repeated for all refactorings (steps 4–8): the project is analyzed and, if any transformation is applied, a new PR is submitted to be considered by the project's integrator. Since we wanted to engage the community and get feedback about the refactorings, we manually created each PR with a personalized message, including a brief explanation of committed code changes.

We analyze 140 free and open-source Android apps collected from *F-droid*⁹. Apps are selected by date of publish (i.e., it was given priority to newly released apps), considering exclusively Java projects (e.g., *Kotlin* projects are filtered out) with a *Github* repository. We select only one git service for the sake of simplicity. Apps in the dataset are spread in 17 different categories, as depicted in Figure 6.4.

Table 6.1 presents descriptive statistics for the source code and repository of the mobile applications in the dataset: number of lines of code LOC, McCabe's Cyclomatic Complexity (CC), mean Weighted Methods per Class¹⁰ (WMC), Lack of Cohesion of Methods¹¹ (LCOM) [Etzkorn et al., 1998], number of Java files, number of XML files, number of Github Forks, Github Stars, and contributors. These metrics were collected using the static analysis tool *Designite*¹² and the *GitHub API v3*¹³.

⁸F-droid repository is available at <https://f-droid.org> (Visited on June 5, 2020).

⁹Data was collected on Nov 27, 2016, and it is available here: <https://doi.org/10.6084/m9.figshare.7637402>.

¹⁰Weighted Methods per Class (WMC) is the sum of the complexity of methods in a class.

¹¹Lack of Cohesion of Methods (LCOM) is a software code metric that measures the correlation between class members and methods. Values fall between 0, indicating perfect cohesion, and 1, indicating a complete lack of cohesion.

¹²Designite's website: <http://www.designite-tools.com> visited in June 5, 2020.

¹³GitHub API v3's website:<https://developer.github.com/v3/> visited in June 5, 2020.

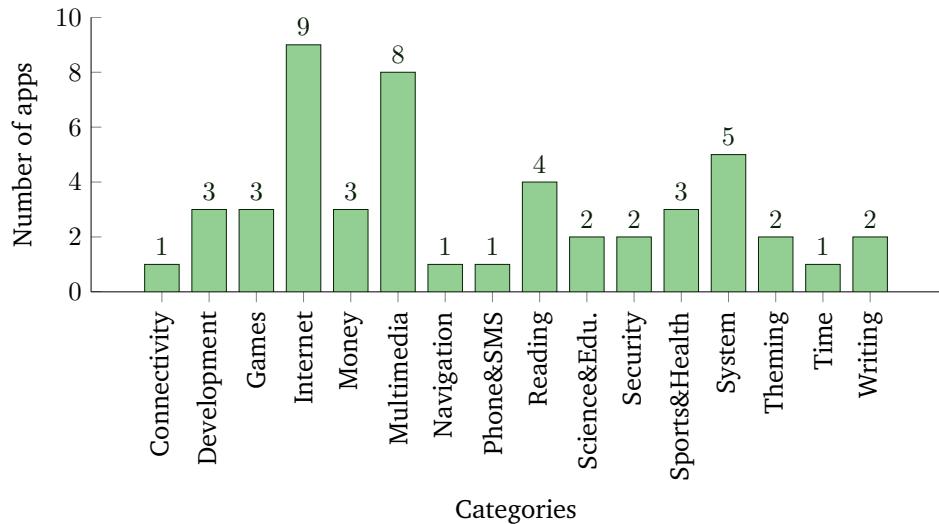


Figure 6.4: Number of apps per category in the dataset.

Table 6.1: Descriptive statistics of projects in the dataset.

	LOC	CC	WMC	LCOM	Java Files	XML Files	Github Forks	Github Stars	Contributors
Mean	20350	3532	17.41	0.29	103	102	65	179	15
Min	13	2	1.00	0.00	0	4	0	0	1
25%	1444	271	11.14	0.23	13	23	3.75	7.75	2
Median	4641	946	15.20	0.27	38	48	9	24	3
75%	14795	3007	21.50	0.34	106	97	39	111	10
Max	388853	77889	82.82	0.67	1678	2109	1483	4488	323
Total	2869394	—	—	—	15308	15103	9547	26484	2162

The dataset comprehends very diverse mobile applications. It goes from very simple apps, such as *Storage-USB*¹⁴, with 13 LOC and complexity CC of 2, to large apps, such as *Slide*¹⁵ with almost 400k LOC and complexity CC of 14631, or *Osmand*¹⁶, with over 300k LOC and complexity CC of 77889. The largest project in terms of Java files is *TinyTravelTracker* (1878), while *NewsBlue* is the largest in terms of XML files (2109). Most apps in the dataset have reasonable cohesion, with LCOM below 0.34 for 75% of the apps; apps with low/moderate cohesion were also analyzed, having LCOM values up to 0.67.

In total, we analyzed 2.8M lines of Java code (LOC) in 6.79GB of Android projects in 4.5 hours — 15103 XML files, and 15308 Java files.

6.5 Results

Our experiment yielded a total of 222 refactorings, which were submitted to the original repositories as PRs. Multiple refactorings of the same type were grouped

¹⁴*Storage-USB* basically launches Storage Settings directly from the apps drawer. Github repository: <https://github.com/enricocid/Storage-USB> visited in June 5, 2020.

¹⁵*Slide* is a browser for the social news forum Reddit. Github Repository: <https://github.com/crama/Slide> visited in June 5, 2020.

¹⁶*Osmand* is a navigation app. Github repository: <https://github.com/osmandapp/Osmand> visited in June 5, 2020.

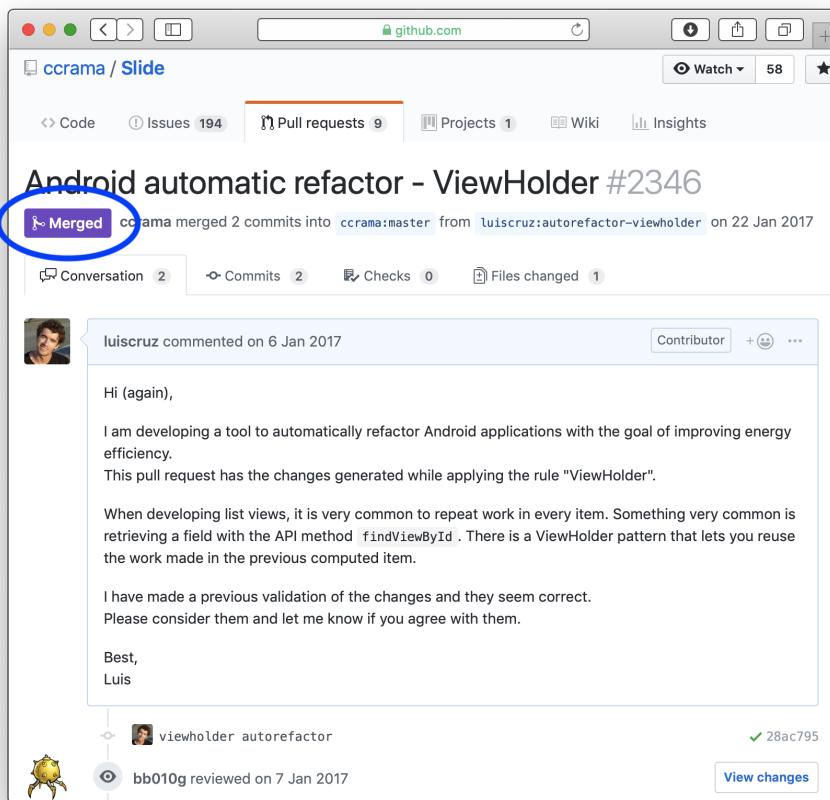


Figure 6.5: An example of a PR containing code changes authored using *Leafactor* that was submitted to the Github project of the Android app *Slide*. The PR was accepted and successfully merged into the app.

Table 6.2: Summary of refactoring results.

Refactoring	ViewHolder	DrawAllocation	Wakelock	Recycle	OLP*	Total
Total Refactorings	7	0	1	58	156	222
Total Projects	5	0	1	23	30	45
Percentage of Projects	4%	0%	1%	16%	21%	32%
Incidence per Project	1.4×	-	1.0×	2.5×	5.2×	4.8×

*OLP — ObsoleteLayoutParams

in a single PR to avoid creating too many PRs for a single app. It resulted in 59 PRs spread across 45 apps. This is a demanding process since each project has different contributing guidelines. Nevertheless, by the time of writing, 18 apps had successfully merged our contributions for deployment.

An example of the PR submitted to the projects is illustrated in Figure 6.5. *Leafactor* performed the refactoring *ViewHolder* in the app *Slide*¹⁷, and developers successfully merged our PR. The full thread can be found in the *Github* project *ccrama/Slide* with reference #2346¹⁸.

¹⁷*Slide*'s website: <http://trikit.co/slide/> (Visited on June 5, 2020).

¹⁸PR of the *ViewHolder* of app *Slide*: <https://github.com/ccrama/Slide/pull/2346> (Visited on June 5, 2020).

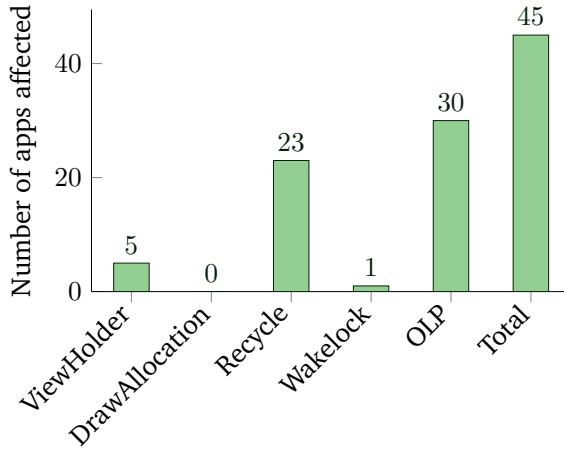


Figure 6.6: Number of apps affected per refactoring.

Table 6.2 presents the results for each refactoring. It shows the total number of applied refactorings, the total number of projects that were affected, the percentage of affected projects, and the average number of refactorings per affected project. In addition, the table presents the combined results for the occurrence of any type of refactoring (*Total*).

ObsoleteLayoutParams was the most frequent pattern. It was applied 156 times in a total of 30 projects out of the 140 in our dataset (21%). In average, each affected project had 5 occurrences of this pattern. *Recycle* comes next, occurring in 23 projects (16%) with 58 refactorings. *DrawAllocation* and *Wakelock* only showed marginal impact.

In addition, Figure 6.6 presents a plot bar summarizing the number of projects affected amongst all the studied refactorings.

The mobile application with a bigger incidence of refactorings was the Android application for the cloud platform *NextCloud*¹⁹. *Leafactor* has refactored two occurrences of *Recycle*, two of *ViewHolder*, and six of *ObsoleteLayoutParams*. In terms of the total number of refactorings, *QR Scanner*²⁰ was the app with a higher number of occurrences, with 35 occurrences of *ObsoleteLayoutParams*.

For reproducibility and clarity of results, all the data collected in this study is publicly available²¹. In addition, all the PRs are public and can be accessed through the official repositories of the apps.

¹⁹NextCloud's website: <https://nextcloud.com> (Visited on June 5, 2020).

²⁰QR Scanner's entry on Google Play:

<https://play.google.com/store/apps/details?id=com.secuso.privacyFriendlyCodeScanner> (Visited on June 5, 2020).

²¹Spreadsheet with all experimental results: <https://doi.org/10.6084/m9.figshare.7637402>.

6.6 Discussion

In this section, we answer the proposed research questions, i.e., RQs 6.1 and 6.2, and discuss potential implications of our results.

Research Question 6.1

What is the prevalence of energy code smells in FOSS Android applications?

We have found energy efficiencies in a considerable number of Android apps (45), representing 32% of the apps in this study.

The code smell *Obsolete Layout Param* was found in 21% of the projects. This relates to the fact that app views are often created in an iterative process with several rounds of trial and error. Since some parameters have no effect under certain contexts, useless UI specification statements can go unnoticed by developers.

Recycle is frequent too, being observed in 16% of projects. This pattern is found in Android API objects that can be found in most projects (e.g., database cursors). Although a clean fix is to use the Java *try-with-resources* statement²², it requires version 19 or earlier of Android SDK (introduced with Android 4.4 Kitkat). However, developers resort to a more verbose approach for backward compatibility which requires explicitly closing resources, hence prone to mistakes.

Our *DrawAllocation* checker did not yield any result. It was expected that developers were already aware of *DrawAllocation*. Still, we were able to manually spot allocations that were happening inside a drawing routine. Nevertheless, those allocations are using dynamic values to initialize the object. In our implementation, we scope only allocations that will not change between iterations. Covering those missed cases would require updating the allocated object in every iteration. While spotting these cases is relatively easy, refactoring would require better knowledge of the class that is being instantiated. Similarly, *WakeLocks* are very complex mechanisms and fixing all misuses still requires further work.

In the case of *ViewHolder*, although it only impacted 4% of the projects, we believe it has to do with the fact that 1) some developers already know this pattern due to its performance impact, and 2) many projects do not implement dynamic list views. *ViewHolder* is the most complex pattern we have in terms of LOC — a simple case can require changes in roughly 35 LOC. Although changes are easily understandable by developers, writing code that complies with *ViewHolder* pattern is not intuitive.

²²Documentation about the Java *try-with-resources* statement: <https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html> (Visited on June 5, 2020).

Research Question 6.2

Is automatic refactoring a feasible approach to improve the energy efficiency of mobile applications?

We were able to improve energy efficiency in the official release of 18 Android apps. Results show that an automatic refactoring tool can help developers ship more energy-efficient apps. Since the identified code smells are only visible after long periods of app activity they can easily go unnoticed. From the feedback developers provided in the PRs, we have noticed that developers are open to recommendations from an automated tool. Only in a few exceptions, developers expressed being unhappy with our contributions. Most developers were curious about the refactorings and they recognized being unaware of their impact on energy efficiency. This is consistent with previous work [Pang et al., 2016; Sahin et al., 2014].

In a few cases, code smells were found in code that does not affect the energy consumption of the app itself (e.g., test code). In those cases, our PRs were not merged. Nevertheless, we recommend consistently complying with these optimizations in all types of code since new developers often use tests to help understand how to contribute to a project.

Gainings on energy efficiency may vary depending on the application and the use cases in which they occur. Measuring the effective impact on energy consumption is not trivial as it requires a complex setup. In Chapter 5, we found these refactorings to improve energy efficiency up to 5% in real use case scenarios. Nonetheless, these refactorings are recommended by the official Android documentation²³ as best practices for performance.

A visible side effect of the refactorings featured by *Leafactor* is the questionable maintainability of the code introduced. Although the refactorings are implemented based on the official Android documentation, the resulting code is considerably longer and less intuitive for patterns such as *ViewHolder* and *Recycle*. This is a threat to the adoption of energy-efficient practices in Android applications. Mobile frameworks should feature coding mechanisms aiming to improve energy efficiency without hindering code maintainability.

6.7 Related Work

Energy efficiency of mobile apps is being addressed with many different approaches. Some works opt by simplifying the process of measuring the energy consumption of mobile apps [Zhang et al., 2010; Pathak et al., 2012a; Pathak et al., 2011b;

²³*ViewHolder* is documented here: <https://developer.android.com/training/improving-layouts/smooth-scrolling> (Visited on June 5, 2020).

Hao et al., 2013; Di Nucci et al., 2017a; Couto et al., 2014]. Alternatively, other works study the energy footprint of software design choices and code patterns that will prevent developers from creating code with poor energy efficiency [D. Li et al., 2014a; D. Li and Halfond, 2014; D. Li and Halfond, 2015; Linares-Vásquez et al., 2017a; Malavolta et al., 2017; R. Pereira et al., 2017a].

The frequency of anti-patterns in Android apps was studied in previous work [Hecht et al., 2015]. Patterns were automatically detected in 15 apps using the tool *Paprika* which was developed to perform static analysis in the bytecode of apps. Although Paprika provides valuable feedback on how to fix their code, developers need to manually apply the refactorings. Our study differs by focusing on energy-related anti-patterns and by applying automatic refactoring to resolve potential issues.

Previous work has also studied the importance of providing a catalog of bad smells that negatively influence the quality of Android applications [Reimann et al., 2014; Reimann and Aßmann, 2013]. Although the authors motivate the importance of using automatic refactoring, their approach lacks an extensive implementation of their catalog. Related work has implemented 15 code-smells from this catalog proposed Reimann et al. (2013) in an automatic refactoring tool, *aDoctor* [Palomba et al., 2017]. In our work, we use this approach to improve the energy efficiency of Android applications.

Another work has focused exclusively on patterns to improve energy efficiency of iOS and Android mobile applications [Cruz and Abreu, 2019a]. However, no efforts were made regarding the automatic refactoring of the cataloged energy patterns. In our work, we implement automatic refactoring for five energy patterns. In addition, we validate our refactorings by applying *Leafactor* in a large dataset of real Android apps. Moreover, we assess how automatic refactoring tools for energy can positively impact the Android FOSS community.

Other works have detected energy-related code smells by analyzing source code as *TGraphs* [Gottschalk et al., 2012; Ebert et al., 2008]. Eight different code smell detectors were implemented and validated with a navigation app. Fixing the code with automatic refactoring was discussed but not implemented. In addition, although studied code smells are likely to have an impact on energy consumption, no evidence was presented.

Previous work has used the EFG of the app to optimize resource usage (e.g., GPS, Bluetooth) [Banerjee and Roychoudhury, 2016]. Results show significant gains in energy efficiency. Nevertheless, although this process provides details on how to fix the code, it is not fully automated yet.

Other works have studied and applied automatic refactorings in Android applications [Sahin et al., 2014; Sahin et al., 2016]. However, these refactorings were not mobile specific.

Besides refactoring source code, other works have focused on studying the impact of UI design decisions on energy consumption [Linares-Vásquez et al., 2017a]. Agolli, T., et al. have proposed a methodology that suggests changes in the UI colors of apps. The new UI colors, despite being different, are almost imperceptible by users and lead to savings in the energy consumption of mobile phones' displays [Agolli et al., 2017]. In our work, we strictly focus on changes that do not change the appearance of the app.

6.8 Summary

In this chapter, we use automatic refactoring to improve the energy efficiency of Android applications. Concretely, in this chapter:

- We present the architecture of the automatic refactoring tool *Leafactor* for Java codebases and XML specifications.
- We provide examples of code changes to fix the energy code smells studied in Chapter 5 and implement them as refactoring rules in the *Leafactor*.
- We use *Leafactor* on 140 FOSS Android apps to study how automatic refactoring can help developers ship energy-efficient applications.
 - We found that a considerable percentage of the Android apps in the dataset (32%) is released with energy code smells (answering RQ 6.1, page 100).
 - As a result, we fixed 222 energy-related anti-patterns and improved the energy footprint of 45 Android applications (answering RQ 6.2, page 100).
- We identified maintainability as a potential issue when improving energy efficiency.

Catalog of Energy Patterns for Mobile Apps



Catalog of energy patterns for mobile applications

Luis Cruz and Rui Abreu

In: *Empirical Software Engineering*, 2019.

Abstract

*Software engineers make use of design patterns for reasons that range from performance to code comprehensibility. Several design patterns capturing the body of knowledge of best practices have been proposed in the past, namely *creational*, *structural* and *behavioral* patterns. However, with the advent of mobile devices, it becomes a necessity a catalog of design patterns for energy efficiency. In this work, we inspect commits, issues and pull requests of 1027 Android and 756 iOS apps to identify common practices when improving energy efficiency. This analysis yielded a catalog, available online, with 22 design patterns related to improving the energy efficiency of mobile apps. We argue that this catalog might be of relevance to other domains such as Cyber-Physical Systems and Internet of Things. As a side contribution, an analysis of the differences between Android and iOS devices shows that the Android community is more energy-aware.*

7.1 Introduction

Design patterns have been formalized to provide general, reusable solutions to recurrent problems in software design. According to their main purpose, design patterns were originally categorized in *creational*, *structural*, and *behavioral* patterns [Vlissides et al., 1995]. Further efforts have leveraged domain-specific catalogs of design patterns to meet non-functional requirements such as security [Yskout et al., 2015]. Design patterns also play an important role in educating developers, since they tend to learn by looking at code examples or using boilerplate code following well defined solutions [Pham et al., 2015; Pham et al., 2013].

There is a number of practices from experienced developers that lie in the history of mobile app projects [Negara et al., 2014; Palomba et al., 2018]. In this work, we collect the set of patterns that developers adopt to improve the energy efficiency of their apps. We analyze 1783 apps from Android (1027) and iOS (756) and compare practices of developers towards energy efficiency amongst the two most popular mobile platforms.

In particular, we aim to answer the following questions:

Research Question 7.1

Which design patterns do mobile app developers adopt to improve energy efficiency?

We describe a catalog of 22 patterns that mobile app developers resort to when addressing energy efficiency. We document these patterns so that other developers can learn more about energy best practices and reuse them in their projects.

Research Question 7.2

How different are mobile app practices addressing energy efficiency across different platforms?

We found that Android developers have higher awareness towards energy efficiency improvement than iOS developers. We show the prevalence of each energy pattern in the two platforms and discuss potential causes.

In this chapter, we make the following contributions:

- We propose a catalog of energy patterns with a detailed description and instructions for mobile app developers and designers. It is available online: <https://tqrg.github.io/energy-patterns>, and we welcome contributions from the community as pull request.
- We provide a dataset with 1563 commits, issues, and pull requests in which mobile app development practitioners address the energy efficiency of their apps. The dataset and collection tools are available online: <https://github.com/TQRG/energy-patterns>.
- We compare energy efficiency awareness in mobile app development in different platforms (viz. Android and iOS).

The remainder of this chapter is organized as follows. Related work is discussed in Section 7.2. Section 7.3 outlines the methodology used to collect data and extract energy patterns in our study, followed by Section 7.4 describing the collection of energy patterns. Section 7.5 summarizes the collected data and discusses implications of the list of proposed patterns. Threats to the validity are discussed in Section 7.6. Finally, we summarize the main contributions in Section 7.7.

7.2 Related Work

Improving energy efficiency of mobile apps has gained the attention of the research community recently, which addressed the challenge in different ways: identifying

energy bugs [Pathak et al., 2011a; Banerjee et al., 2014; Vekris et al., 2012], profiling energy consumption [Wilke et al., 2013a; Y. Liu et al., 2013; Hao et al., 2013; Behrouz et al., 2015; Pathak et al., 2011b; Zhang et al., 2010; S. Chowdhury et al., 2018a; Di Nucci et al., 2017b; Hindle et al., 2014; Romansky et al., 2017], or understanding best coding practices for energy efficiency [Sahin et al., 2016; Cruz and Abreu, 2017; Cruz and Abreu, 2018b; Linares-Vásquez et al., 2014; Pathak et al., 2012a].

In previous work, Moura et al., 2015 have mined 290 energy-saving software commits, identifying 12 categories of source code modification to improve energy usage [Moura et al., 2015]: *Frequency and voltage scaling, Use power efficient library/device, Disabling features or devices, Energy bug fix, Low power idling, Timing out, Avoid polling, Pin management, Display and UI tuning, Avoid unnecessary work, Miscellaneous, and Outlier*. The programming languages used to implement the software systems used in this study were diverse: programming C (158 projects), Java (25 projects), Bourne Shell (17 projects), Arduino Sketch (15 projects), and C++ (12 projects). They found that roughly 50% of energy-saving commits target lower levels of the software stack (e.g., kernels and drivers), which is not a level of abstraction commonly considered during the design of mobile apps. Our work extends this approach to the ecosystem of mobile apps by compiling a set of coding practices that can be used by practitioners across mobile apps on different platforms. Thus, our dataset of apps also includes projects written in *Swift, Objective-C, Java, Kotlin*, and any other language used for mobile app development in iOS or Android. In addition, we detail these and other energy-saving categories with a context and guidelines to help developers decide on the most appropriate pattern. Moreover, we compare the prevalence of these patterns across different mobile platforms.

With a similar approach, Bao et al., 2016 have mined 468 power management commits to find coding practices in Android apps [Bao et al., 2016]. Using a hybrid card sort approach, six different power management practices were identified: *Power Adaptation, Power Consumption Improvement, Power Usage Monitoring, Optimizing Wake Lock, Adding Wake Lock and Bug Fix & Code Refinement*. The study shows that power management activities are more prevalent in navigation apps. Conversely, our work focuses on energy-saving commits, pull requests, and issues. Using the same taxonomy, our work concentrates exclusively on coding practices for *Power Adaptation, and Power Consumption Improvement*. Moreover, rather than analyzing the prevalence of power management activities amongst different app categories, we emphasize on providing actionable findings for mobile app practitioners. Finally, we extend this work to the iOS mobile platform, which shares a big part of the mobile app market.

Previous work studied the views of mobile app developers on energy efficiency improvement by mining *StackOverflow*¹ posts [Pinto et al., 2014]. It was found that developers make interesting questions about energy efficiency problems. However, the answers provided on this topic are often flawed or vague. Our work analyses mobile app projects to collect recurrent solutions adopted by developers.

There is work that studied the impact of performance optimizations on the energy efficiency of mobile apps [Hecht et al., 2016; Cruz and Abreu, 2017; Linares-Vásquez et al., 2014; Sahin et al., 2016]. However, only platform-specific optimizations were addressed – e.g., in early versions of Android using get/set methods internally was less energy-efficient than accessing fields directly. In our work, we focus on patterns that can be applied regardless of the mobile platform being used.

Furthermore, related work studied the impact of high-level coding and design practices. E.g., the use of advertisement increases energy usage of mobile apps [Pathak et al., 2012a], bundling small **HyperText Transfer Protocol (HTTP)** requests can be used to enhance energy efficiency [D. Li and Halfond, 2014]. We assess how mobile app developers are using these and other patterns to improve energy efficiency in real mobile apps. Measuring the effective impact of these optimizations on energy efficiency is out of the scope of our work.

Reimann et al. have published a catalog of quality smell patterns for Android apps [Reimann et al., 2014]. Our work differentiates by focusing on energy efficiency improvements and including iOS apps. Still, there is one pattern that is common in the two catalogs: Reimann et al.’s *Early Resource Binding* and our’s *Open Only When Necessary* (cf. Section 7.4.5).

The impact of general purpose software design patterns on energy efficiency has been studied in previous work [Sahin et al., 2012]. It was shown that design patterns may have different impacts on energy consumption. Related work has also evaluated the impact of different machine learning algorithms [McIntosh et al., 2018]. The most efficient technique algorithm depends on properties such as the size of the dataset, and the number of data attributes. Our work differs as we exclusively focus on patterns that are applied to improve the energy efficiency of mobile apps.

Most of the works described above focus on a single platform, notably Android. In this study, we also consider iOS. Coding practices for iOS development have seldom been studied in related work. In a study with 279 iOS apps and 1551 Android apps, no significant distinction was found in the prevalence of code smells between iOS and Android apps [W. Oliveira et al., 2017]. A different work has studied error handling practices of Swift² [Cassee et al., 2018]. Nearly half of the 2733 Swift projects did

¹*StackOverflow* is a collaborative Web platform for questions and answers on a wide range of topics in computer programming.

²Swift is the official programming language for iOS apps.

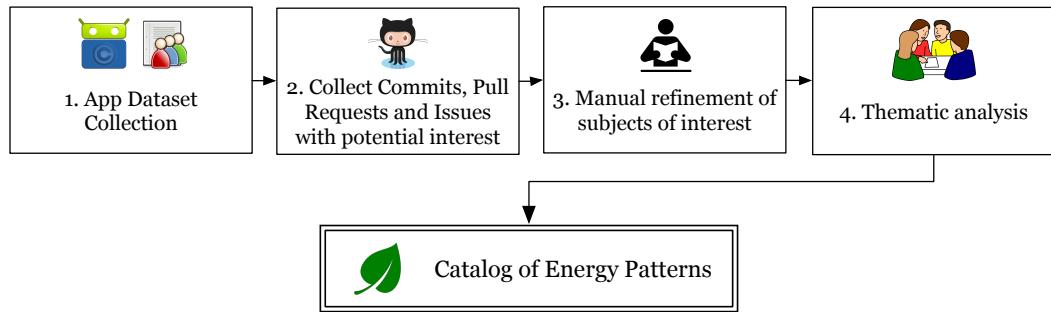


Figure 7.1: Methodology used to extract energy patterns from mobile apps.

not exhibit any error handling code. Our work provides more enlightenment on practices of developers amongst the iOS ecosystem.

7.3 Methodology

We designed a methodology to extract energy patterns from existing mobile apps. In total, we collect a total of 1027 Android apps and 756 iOS apps, including apps designed for smartphones, tablets, wearables, or e-paper devices. Essential to our analysis, apps from both platforms are open source and have their git repositories available on GitHub³. Our methodology is illustrated in Figure 7.1 and comprised the following four main tasks:

- App dataset collection
- Automatic gathering of subjects with potential interest (i.e., commits, issues, and pull requests)
- Manual refinement of subjects of interest
- Thematic analysis (infer energy patterns) according to the solution encountered to improve energy efficiency

7.3.1 App Dataset Collection

Multiple open source mobile app catalogs were combined to collect Android and iOS mobile apps. For Android, we resort to *F-Droid*, a catalog that lists 2800 free and open source Android apps⁴. There are open source apps that are not available in *F-droid* for not fulfilling free software requirements (e.g., *Signal* app⁵). Although these are just a minority of apps we argue that they can provide relevant input on energy efficiency practices. Thus, we included Android apps listed in community-curated

³GitHub is a social coding platform with a git web interface.

⁴F-droid's website: <https://f-droid.org/> (Visited on June 5, 2020).

⁵Signal is an open source messaging app: <https://signal.org> (Visited on June 5, 2020).

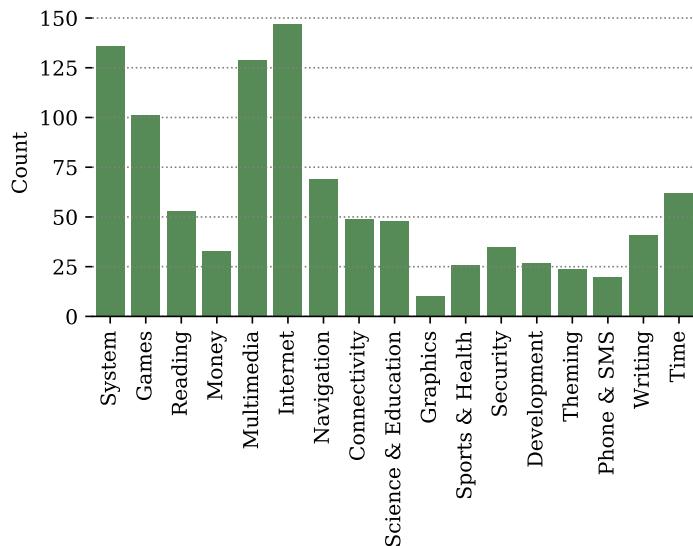


Figure 7.2: Distribution of categories in Android apps.

collections of Android open source apps⁶. This resulted in 1027 apps – 1001 from *F-droid* and 26 from curated lists.

For iOS we use the *Collaborative List of Open-Source iOS Apps*⁷, amounting to 829 apps listed with the help of a community of 195 collaborators. Given our constraint of having a publicly available GitHub repository, we ended up including 756 iOS apps in our study.

The apps used in this study are from a wide range of categories for Android and iOS, as depicted in Figure 7.2 and Figure 7.3, respectively. In addition, Table 7.1 shows the dispersion of apps in terms of popularity metrics: GitHub stars, GitHub forks, number of reviews, and rating. The number of reviews and the ratings are from the apps in the dataset that are published in the Google Play Store or the iOS App Store. Thus, we only provide these metrics for a subset of the apps in this study: 64% of Android apps and 20% of iOS apps. GitHub stars go up to roughly 15K in both platforms, while GitHub forks up to 8K in Android and 5K in iOS. On average, the apps have a rating of 4 out of 5.

7.3.2 Automatic gathering of commits, issues, and pull requests

In this step, we collect from all GitHub repositories in our dataset any commit, issue or pull request that potentially contains energy improvement practices. As done

⁶Amazing open source Android apps curated list available here: <https://github.com/Mybridge/amazing-android-apps> (Visited on June 5, 2020).

⁷The list is available here: <https://github.com/dkhamsing/open-source-ios-apps> (Visited on June 5, 2020)

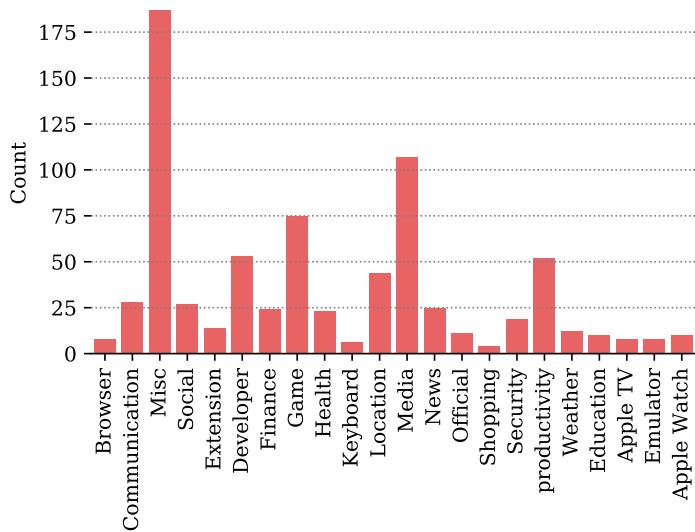


Figure 7.3: Distribution of categories in iOS apps.

Table 7.1: Descriptive statistics of the popularity metrics of the Android and iOS apps in the dataset.

	Platform	Mean	Std	Min	25%	Median	75%	Max
Stars	Android	145	608	0	6	20	68	15159
	iOS	486	1272	0	18	71	329	15318
Forks	Android	75	302	0	4	14	46	7811
	iOS	118	354	0	7	20	71	4820
Number of Reviews*	Android	31855	529676	1	24	138	1087	13080790
	iOS	3241	12597	5	18	75	1038	115011
Rating*	Android	3.8	0.5	1.0	4.0	4.0	4.0	5.0
	iOS	4.1	0.7	2.0	3.6	4.0	4.5	5.0

*as in Google Play Store and iOS App Store.

in previous work [Bao et al., 2016], any instance that mentions the words *energy*, *battery*, or *power* is selected. The following regular expression is used:

```
.*(energy|battery|power).*
```

The *GitHub API v3* was used to automatically collect data from public repositories. Note that we only include commits that were merged in the *default* branch of the projects. In total, we gathered 6028 entries that matched this regular expression.

7.3.3 Manual Refinement

We understand that the regular expression used in the automatic data collection yields many false positives. As an example, consider the following entries collected in the previous step:

- “*Adding a link to the app’s device settings in iOS Settings.app would be great for power users.*” (False positive found in the app *WordPress* for iOS⁸).
- “*(...) recently a lot of issues that the core team does not have the energy to implement themselves have been closed.*” (False positive found in the app *Minetest* for Android⁹).
- “*One thing is really important on mobile devices, and that is power consumption*” (True positive found in the app *ChatSecure* for iOS¹⁰).

Although the first two examples match with the regular expression, they are not expected to deal with energy-related practices. On contrary, the last example is referring to the topic of power consumption. Thus, it is likely to provide useful insights on energy improvement practices.

To filter out unrelated entries, we resort to a manual analysis of each instance, comprising two iterations:

1. Check the line where a match with the regular expression was found. If the sentence does not mention anything related to energy consumption, the subject is discarded from the dataset.

⁸The whole thread can be found here: <https://github.com/wordpress-mobile/WordPress-iOS/issues/6057> (Visited on June 5, 2020).

⁹The whole thread can be found here: <https://github.com/minetest/minetest/issues/6394> (Visited on June 5, 2020).

¹⁰The whole thread can be found here: <https://github.com/ChatSecure/ChatSecure-iOS/issues/31> (Visited on June 5, 2020).

- Check the whole thread in which the mention was found. I.e., open the GitHub page where the commit, issue, or pull request is documented and analyze the context in which the energy topic is being discussed. This step removes cases in which contributors are discussing the topic of energy for contexts that are not related to energy efficiency improvement. E.g., cases were found in which developers were talking about the battery of their own laptops, or in which the app actually had a feature in which it displayed the status of the battery¹¹.

After this step, we ended up with a total of 1563 subjects: 332 commits, 1089 issues, and 142 pull requests.

7.3.4 Thematic Analysis

We resort to a methodology based on Thematic Analysis [Fereday and Muir-Cochrane, 2006] to derive design patterns from commits, issues and pull requests. Thematic Analysis is a widely-used qualitative data analysis method, that focuses on identifying patterned meaning in a dataset. Its hybrid process of deductive and inductive analysis has been successfully used in previous work to categorize software commits [Moura et al., 2015]. We follow a similar approach by adopting a four-stage process:

Familiarization with data We have carefully read the information provided in commits, issues or pull-requests, including comments and descriptions. Relevant advice and reasoning are collected and studied using online documentation.

Generating initial labels For each commit, issue, and pull request, we describe the change in a generic short sentence – i.e., without resorting to specific properties of the app. This process was split into several iterations to discuss amongst both authors in order to refine the labels.

Reviewing themes After having all subjects spread in different labels, we discuss and review them to find themes that should be merged or split. Some themes were discarded for not being present in a sufficient number of subjects. In particular, we filter out themes that did not occur in at least three different apps. In addition, we corroborate and legitimate coded themes, by finding evidence in the literature that supports or discards themes.

Defining and naming themes In this stage, we make a structured description of each theme to provide a set of straight-forward guidelines that can be reused in the design of different mobile app projects. Each theme is now converted into an **Energy Pattern**. Each pattern includes a *name*, a brief *description*, a *context* or problem in which the pattern can be applied, and a *solution* with instructions on how to apply the pattern. The solutions provided are based

¹¹An example can be found here: <https://github.com/hrydgard/ppsspp/issues/7765> (Visited on June 5, 2020).

on a combination of authors' experience, logical arguments, literature, mobile platform documentation, and the data itself.

In total 332 commits, 1089 issues, and 142 pull requests were analyzed using this approach. As a result, we identify and document 22 energy patterns that appear in 431 of the analyzed subjects.

7.3.5 Reproducibility-Oriented Summary

Based on previous guidelines for app store analyses [Martin et al., 2017], we describe ours as follows to help repeat the experimental procedure and reproduce the results:

App Stores used to gather collections of apps We use apps available on *F-Droid*, and on the list *Collaborative List of Open-Source iOS Apps*.

Total number of apps used The study comprises 1783 apps.

Breakdown of free/paid apps used in the study Only non-paid apps are listed in our dataset.

Categories used All categories were included in this study.

API usage GitHub **Representational State Transfer (REST) API v3**¹².

Whether code was needed from apps Source code was required to analyze code changes in commits.

Fraction of open source apps Open source apps are used exclusively.

Static analysis techniques No static analysis was performed. In some tasks, the code was analyzed manually by the authors.

All scripts and tools developed in this work are publicly available with an open source license: <https://github.com/TQRG/energy-patterns>.

7.4 Energy Patterns

In this section, we present the energy patterns collected in this study. As mentioned before, each energy pattern is described by the following entries: context, solution, and an example illustrating a practical usage of the pattern. All these patterns are also available online: <https://tqrg.github.io/energy-patterns>. The website also provides links to the occurrences in the apps, disclosing the discussion performed by developers and practical examples of the patterns in this catalog.

¹²Documentation of GitHub REST API v3 available here: <https://developer.github.com/v3/> (Visited on June 5, 2020).

We summarize the occurrences of these patterns in Table 7.2, presenting their frequency in Android and iOS platforms along with an indication of their prevalence in related work and grey literature (as listed in the Appendix 7.A).

Table 7.2: Energy patterns' occurrences and related work.

Pattern	Android	iOS	Related Work	Grey Literature
Dark UI Colors	28	2	[Agolli et al., 2017; Linares-Vásquez et al., 2017a; D. Li et al., 2014b; D. Li et al., 2015]	-
Dynamic Retry Delay	10	2	-	-
Avoid Extraneous Work	32	9	-	[1]
Race-to-idle	27	5	[Y. Liu et al., 2016; Banerjee and Roychoudhury, 2016; Cruz and Abreu, 2017; Pathak et al., 2012b]	-
Open Only When Necessary	4	3	[Banerjee and Roychoudhury, 2016; Reimann et al., 2014]	-
Push over Poll	13	3	-	[2, 3]
Power Save Mode	24	5	-	[4]
Power Awareness	35	6	[Bao et al., 2016]	[4]
Reduce Size	3	0	[Boonkrong and Dinh, 2015]	[5]
WiFi over Cellular	13	2	[Metri et al., 2012]	[6, 7, 8]
Suppress Logs	7	1	[S. Chowdhury et al., 2018b]	-
Batch Operations	17	1	[D. Li and Halfond, 2014; Corral et al., 2015; Cai et al., 2015]	[9, 10, 11]
Cache	14	4	[Gottschalk et al., 2014]	[10]
Decrease Rate	27	10	-	-
User Knows Best	33	11	-	-
Inform Users	6	4	-	-
Enough Resolution	10	7	-	-
Sensor Fusion	12	3	[Shafer and Chang, 2010]	[12]
Kill Abnormal Tasks	11	1	-	-
No Screen Interaction	8	2	-	-
Avoid Extraneous Graphics and Animations	11	8	[Kim et al., 2016]	[1]
Manual Sync, On Demand	4	5	-	-

7.4.1 Dark UI Colors

Provide a dark UI color theme to save battery on devices with display technology **Active Matrix Organic Light-Emitting Diode (AMOLED)** [Agolli et al., 2017; Linares-Vásquez et al., 2017a; D. Li et al., 2014b; D. Li et al., 2015].

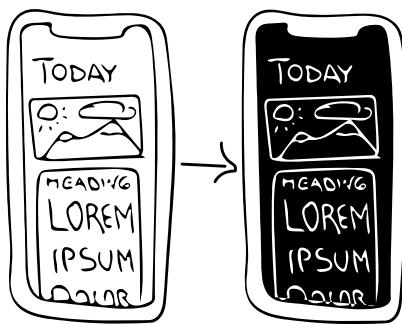


Figure 7.4: UI themes with dark colors are more energy-efficient.

Context Screen is one of the major sources of power consumption in mobile devices.

Apps that require heavy usage of screen (e.g., reading apps) can have a substantial negative impact on battery life.

Solution Provide a UI with dark background colors, as illustrated in Figure 7.4.

This is particularly beneficial for mobile devices with AMOLED screens, which are more energy-efficient when displaying dark colors. In some cases, it might be reasonable to allow users to choose between a light and a dark theme. The dark theme can also be activated using a special trigger (e.g., when battery is running low).

Example In a reading app, provide a theme with a dark background using light colors to display text. When compared to themes using light backgrounds, a dark background will have a higher number of dark pixels.

7.4.2 Dynamic Retry Delay

Whenever an attempt to access a resource fails, increase the time interval before retrying to access the same resource.

Context Mobile apps that need to collect or send data from/to other resources (e.g., update information from a server). Commonly, when the resource is unavailable, the app will unnecessarily try to connect the resource for a number of times, leading to unnecessary power consumption.

Solution Increase retry interval after each failed connection. It can be either a linear or exponential growth. Update interval can be reset upon a successful connection or a given change in the context (e.g., network status).

Example Consider a mobile app that provides a news feed and the app is not able to reach the server to collect updates. Instead of continuously polling the server until the server is available, use the Fibonacci series¹³ to increase the time between attempts.

¹³Fibonacci series is a sequence of numbers in which each number is the sum of the two preceding numbers (e.g., 1, 1, 2, 3, 5, 8, etc.).

7.4.3 Avoid Extraneous Work

Avoid performing tasks that are either not visible, do not have a direct impact on the user experience to the user or quickly become obsolete. This has been documented in the iOS online documentation¹⁴.

Context Mobile apps have to perform a number of tasks simultaneously. There are cases in which the result of those tasks is not visible (e.g., the UI is presenting other pieces of information), or the result is not necessarily relevant to the user. This is particularly critical when apps go to the background. Since the data quickly becomes obsolete, the phone is using resources unnecessarily.

Solution Select a concise set of data that should be presented to the user and enable/disable update and processing tasks depending on their effect on the data that is visible or valuable to the user.

Example Consider a time series plot that displays real-time data. The plot needs to be constantly updated with the incoming stream of data – however, if the user scrolls up/down in the UI view making the plot hidden, the app should cease drawing operations related with the plot.

7.4.4 Race-to-idle

Release resources or services as soon as possible (such as wake locks, screen) [Y. Liu et al., 2016; Banerjee and Roychoudhury, 2016; Cruz and Abreu, 2017; Pathak et al., 2012b].

Context Mobile apps use a number of resources that can be manually closed after use. While active, these resources are ready to respond to requests from the app and require extra power consumption.

Solution Make sure resources are inactive when they are not necessary by manually closing them.

Example Implement handlers for events that are fired when the app goes to background, and release wake locks accordingly.

¹⁴*Energy Efficiency Guide for iOS Apps – Avoid Extraneous Graphics and Animations* available here: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/AvoidExtraneousGraphicsAndAnimations.html> (Visited on June 5, 2020).

7.4.5 Open Only When Necessary

Open/start resources/services only when they are strictly necessary.

Context Some resources require to be opened before use. It might be tempting to open the necessary resources at the beginning of some task (e.g., upon the creation of an activity). However, resources will be actively waiting for requests, and consequently consuming energy.

Solution Open resources only when necessary. This also avoids activating resources that will never be used [Banerjee and Roychoudhury, 2016].

Example In a mobile app for video calls, only start capturing video at the moment that it will be displayed to the user¹⁵.

7.4.6 Push over Poll

Use push notifications to receive updates from resources, instead of actively querying resources (i.e., polling).

Context Mobile apps need to get updates from resources (e.g., from a server). One way of checking for updates is by periodically query those resources. However, this will lead to several requests that will return no update, leading to unnecessary energy consumption.

Solution Use push notifications to get updates from resources. Note – this is a big challenge amongst FOSS apps since there is no good open source alternative for Firebase Cloud Messaging (former Google Cloud Messaging).

Example In a messaging app, instead of actively check for new messages, the app can subscribe push notifications.

7.4.7 Power Save Mode

Provide an energy-efficient mode in which user experience can drop for the sake of better energy usage.

Context Whenever the device battery is running low, users want to avoid losing connectivity before they reach a power charging station. If the device shuts down, users might miss important calls or will not be able to do an important task. Still, apps might be running unimportant tasks that will reduce battery life in this critical context.

¹⁵This is a real example that can be found here: <https://github.com/signalapp/Signal-Android/commit/cb9f225f5962d399f48b65d5f855e11f146cbbcb> (Visited on June 5, 2020).

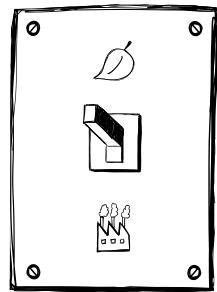


Figure 7.5: Power Save Mode allows to run the app in two different modes: a fully-featured mode and an energy-efficient mode.

Solution The app provides a power save mode in which it uses fewer resources while providing the minimum functionality that is indispensable to the user. It can be activated manually or upon some power events (e.g., when battery reaches a given level). User experience can drop for the sake of energy efficiency. Note, this is enforced in iOS for some use cases if the apps use the BackgroundSync APIs.

Example Deactivate features, reduce update intervals, or deactivate animated effects in the UI.

7.4.8 Power Awareness

Have a different behavior when the device is connected/disconnected to a power station or has different battery levels.

Context There are some features that are not strictly necessary to users although they improve user experience (e.g., UI animations). Moreover, there are operations that do not have high priority and do not need to execute immediately (e.g., backup data in the cloud).

Solution Enable or disable tasks or features according to power status. Even when the device is connected to power, the battery might still be running low, it might be advisable to wait until a pre-defined battery level is reached (or the power save mode is deactivated).

Example Delay intensive operations such as cloud syncing or image processing until the device is connected to a charger.

7.4.9 Reduce Size

When transmitting data, reduce its size as much as possible.

Context Data transmission is a common operation in mobile apps. However, such operations are energy greedy and the time of transmission should be reduced as much as possible.

Solution Exchange only what is strictly necessary, avoiding sending unnecessary data. Use data compression when possible.

Example When performing HTTP requests, use gzip content encoding to compress data.

7.4.10 WiFi over Cellular

Delay or disable heavy data connections until the device is connected to a WiFi network.

Context Data needs to be synchronized with a server but it is not urgent and can be postponed.

Solution Data connections using cellular networks are usually more battery intensive than connections using WiFi [Metri et al., 2012]. Low priority operations that require a data connection to exchange considerable amounts of data should be delayed until a WiFi connection is available.

Example Consider a mobile app to organize photos that allows users to backup their photos in a cloud server. Use an API to check the availability of a WiFi connection and postpone cloud synchronizing in case it cannot be reached.

7.4.11 Suppress Logs

Avoid using intensive logging. Previous work has found that logging activity at rates above one message per second significantly reduces energy efficiency [S. Chowdhury et al., 2018b].

Context Developers resort to logging in their mobile apps to ensure their correct behavior and simplify bug reporting. However, logging operations create overhead on energy consumption without creating value to the end user.

Solution Avoid using intensive logging, keeping rates below one message per second.

Example Disable logging when processing real-time data. If necessary enable only during debugging executions.

7.4.12 Batch Operations

Batch multiple operations, instead of putting the device into an active state many times.

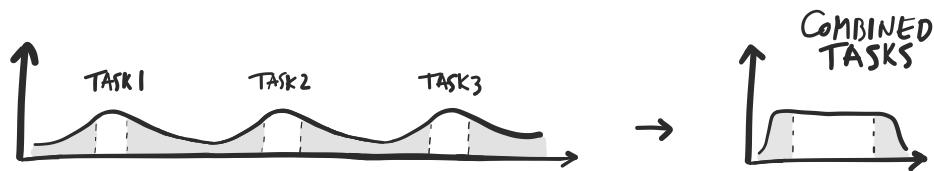


Figure 7.6: Illustration of the energy pattern Batch Operation. Energy usage can be reduced by combining the execution of different tasks.

Context Executing operations separately leads to extraneous tail energy consumptions [D. Li and Halfond, 2014; Corral et al., 2015; Cai et al., 2015]. As illustrated in Figure 7.6, executing a task often induces tail energy consumptions related with starting and stopping resources (e.g., starting a cellular connection).

Solution Bundle multiple operations in a single one. By combining multiple tasks, tail energy consumptions can be optimized. Although background tasks can be expensive, very often they have flexible time constraints. I.e., a given task needs to be eventually executed, but it does not need to be executed in a specific time.

Example Use Job Scheduling APIs (e.g., ‘`android.app.job.JobScheduler`’, ‘`Firebase JobDispatcher`’) that manage multiple background tasks occurring in a device. These APIs will guarantee that the device will exit sleep mode only when there is a reasonable amount of work to do or when a given task is urgent. It combines several multiple tasks to prevent the device from constantly exiting sleep mode (or doze mode). Other examples: execute low priority tasks only if another task is using the same required resources; try to collect location data when other apps are collecting it as well.

7.4.13 Cache

Avoid performing unnecessary operations by using cache mechanisms.

Context Typically mobile apps present data to users that is collected from a remote server. However, it may happen that the same data is being collected from the server multiple times.

Solution Implement caching mechanisms to temporarily store data from a server [Gottschalk et al., 2014]. In addition, verify whether there is an update before downloading all data.

Example Considering a social network app that shows other users’ profiles. Instead of downloading basic information and profile pictures every time a given profile is opened, the app can use data that was locally stored from earlier visits.

7.4.14 Decrease Rate

Increase time between syncs/sensor reads as much as possible.

Context Mobile apps have to periodically perform operations. If the time between two executions is small, the app will be executing operations more often. In some cases, even if operations are executed more often, it will not affect users' perception.

Solution Increase the delay between operations to find the minimal interval that does not compromise user experience. This delay can be manually tuned by developers or defined by users. More sophisticated solutions can also use context (e.g., time of the day, history data, etc.) to infer the optimal update rate.

Example Consider a news app that collects news from different sources, each one having its own thread. Some news sources might have new content only once a week, while others might be updated every hour. Instead of updating all threads at the same rate, use data from previous updates to infer the optimal update rate of these threads. Connect to the news source only if new updates are expected.

7.4.15 User Knows Best

Allow users to enable/disable certain features in order to save energy.

Context Energy efficiency solutions often provide a tradeoff between features and power consumption. However, this tradeoff is different for different users – some users might be okay with fewer features but better energy efficiency, and vice versa.

Solution Allow users to customize their preferences regarding energy critical features. Since this might be more intuitive for power users, mobile apps should provide optimal preferences by default for regular users.

Example Consider a mail client for POP3 accounts as an example. In some cases, users are not expecting any urgent message and may find acceptable checking for new mail in no less than 10 minutes for the sake of energy efficiency. On the other hand, there are cases in which users are waiting for urgent messages and would like to check for messages every two minutes. Since there is no automatic mechanism to infer the optimal update interval, the best option is to allow users to define it.

7.4.16 Inform Users

Let the user know if the app is doing any battery intensive operation.

Context There are specific use cases in mobile apps that can be energy greedy. On the other hand, some features might be dropping user experience in order to improve energy efficiency. If users do not know what to expect from the mobile app, they might think it is not behaving correctly.

Solution Let users know about battery intensive operations or energy management features. Properly flag this information in the user interface (e.g., alerts).

Example Alert users when a power saving mode is active, or alert when a battery intensive operation is about to be executed.

7.4.17 Enough Resolution

Collect or provide high accuracy data only when strictly necessary.

Context When collecting or displaying data, it is tempting to use high resolutions. The problem of using data with high resolution is that its collection and manipulation require more resources (e.g., memory, processing capacity, etc.). As a consequence, energy consumption increases unnecessarily.

Solution For every use case, find the optimal resolution value that is required to provide the intended user experience.

Example Consider a running app that is able to record running sessions. While the user is running, the app presents the current overall distance in real-time. While calculating the most accurate value of the total distance would provide the correct information, it would require precise real-time processing of GPS or accelerometer sensors, which can be energy greedy. Instead, a lightweight method could be used to estimate this information with lower but reasonable accuracy. At the end of the session, the accurate results would still be processed, but without real-time constraints.

7.4.18 Sensor Fusion

Use data from low power sensors to infer whether new data needs to be collected from high power sensors

Context Mobile apps provide features that require reading data or executing operations in different sensors or components. Such operations can be energy greedy, causing high power consumption. Thus, they should be called as fewer times as possible.

Solution Use complementary data from low power sensors to assess whether a given energy-greedy operation needs to be executed.

Example Use the accelerometer to infer whether the user has changed location. In the case that the user is in the same location, data from GPS does not need to be updated.

7.4.19 Kill Abnormal Tasks

Provide means of interrupting energy greedy operations (e.g., using timeouts, or users input).

Context Mobile apps might feature operations that can be unexpectedly energy greedy (e.g., taking a long time to execute).

Solution Provide a reasonable timeout for energy greedy tasks or wake locks. Alternatively, provide an intuitive way of interrupting those tasks.

Example In a mobile app that features an alarm clock, set a reasonable timeout for the duration of the alarm. In case the user is not able to turn it off it will not drain the battery.

7.4.20 No Screen Interaction

Whenever possible allow interaction without using the display.

Context There are apps that require a continuous usage of the screen. However, there are use cases in which the screen can be replaced by less power intensive alternatives.

Solution Allow users to interact with the app using alternative interfaces (e.g., audio).

Example In a navigation app, there are use cases in which users might be only using audio instructions and do not need the screen to be on all the time. This pattern is commonly adopted by audio players that use the earphone buttons to play/pause or skip songs.

7.4.21 Avoid Extraneous Graphics and Animations

Graphics and animations are really important to improve the user experience. However, they can also be battery intensive – use them with moderation [Kim et al., 2016]. This is also a recommendation in the official documentation for iOS developers¹⁶

¹⁶*Energy Efficiency Guide for iOS Apps – Avoid Extraneous Graphics and Animations* available here: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/AvoidExtraneousGraphicsAndAnimations.html> (Visited on June 5, 2020).

Context Mobile apps often feature impressive graphics and animations. However, they need to be properly tuned in order to prevent battery drain of users' devices. This is particularly critical in e-paper devices.

Solution Study the importance of graphics and animations to the user experience. The improvement in user experience may not be sufficient to cover the overhead on energy consumption. Avoid using graphics animations or high-quality graphics. Resort to low frame rates for animations when possible.

Example For example, a high frame rate may make sense during game play, but a lower frame rate may be sufficient for a menu screen. Use a high frame rate only when the user experience calls for it.

7.4.22 Manual Sync, On Demand

Perform tasks exclusively when requested by the user.

Context Some tasks can be energy intensive, but not strictly necessary for some use cases of the app.

Solution Provide a mechanism in the UI (e.g., button) that allows users to trigger energy intensive tasks.

Example In a beacon monitoring app, there are occasions in which the user does not need to keep track of her/his beacons. Allow the user to start and stop monitoring manually.

7.5 Data Summary and Discussion

In this section, we report and discuss findings regarding the presence of energy patterns in the studied mobile applications. In particular, we study differences between Android and iOS platforms and how often energy patterns co-occur within the same app.

7.5.1 Energy Patterns: Android vs. iOS

Next, we assess whether energy efficiency is addressed in a different way in Android and iOS environments.

Energy Efficiency Changes Per mobile app

From the 1783 apps used in this study, we have found 332 (19%) with at least one commit, issue, or pull request related to energy efficiency. In Android we have found 256 out of the 1021 apps (25%), while in iOS we have found 76 out of 756 apps (10%). Congruent results are observed in the extraction of energy patterns: we were able to extract energy patterns in 133 Android apps (13%) and 28 iOS apps (4%). In general, Android developers put more effort into improving the energy efficiency of their mobile apps than iOS developers.

However, research is necessary to explain why this is the case. First, our data comprises only development activities that consciously address energy efficiency. We understand that there are many factors that can affect these results: power management mechanisms implemented by the system, documentation of the frameworks, differences in targeted users, differences in targeted devices, etc. For instance, the iOS platform provides APIs with more strict power management rules, and developers are enforced to use energy best practices beforehand even though they were not addressing energy efficiency per se.

Developers even express their concern on not having their apps accepted in the iOS App store for certain practices such as having tasks periodically running in background¹⁷. Although in this case the main goal is having the app accepted in the app store, energy efficiency is addressed indirectly. On the contrary, Google Play store, the official store for Android apps, is known to have less strict policies [Cuadrado and Dueñas, 2012].

Prevalence of Patterns in Android and iOS

We compare the prevalence of each energy pattern in the two platforms, Android and iOS. Since our app dataset comprises a different number of apps for the two platforms, we use the ratio of the number of occurrences $N_{p|X}$ of a given pattern p divided by the total number of apps (M_X) studied for a given platform X :

$$R(p, X) = \frac{N_{p|X}}{M_X}, \quad X = \{\text{iOS, Android}\} \quad (7.1)$$

Figure 7.7 shows the values of R for every pattern and platform. In general, energy patterns are more prevalent in Android rather than iOS apps. Only two energy patterns were more frequent in iOS: *Open Only When Necessary* and *Manual Sync, On Demand*. In addition, *Inform Users*, *Enough Resolution*, and *Avoid Extraneous*

¹⁷An example of developers dealing with the strict policy of the iOS app store: <https://github.com/owncloud/ios/issues/13> (Visited on June 5, 2020).

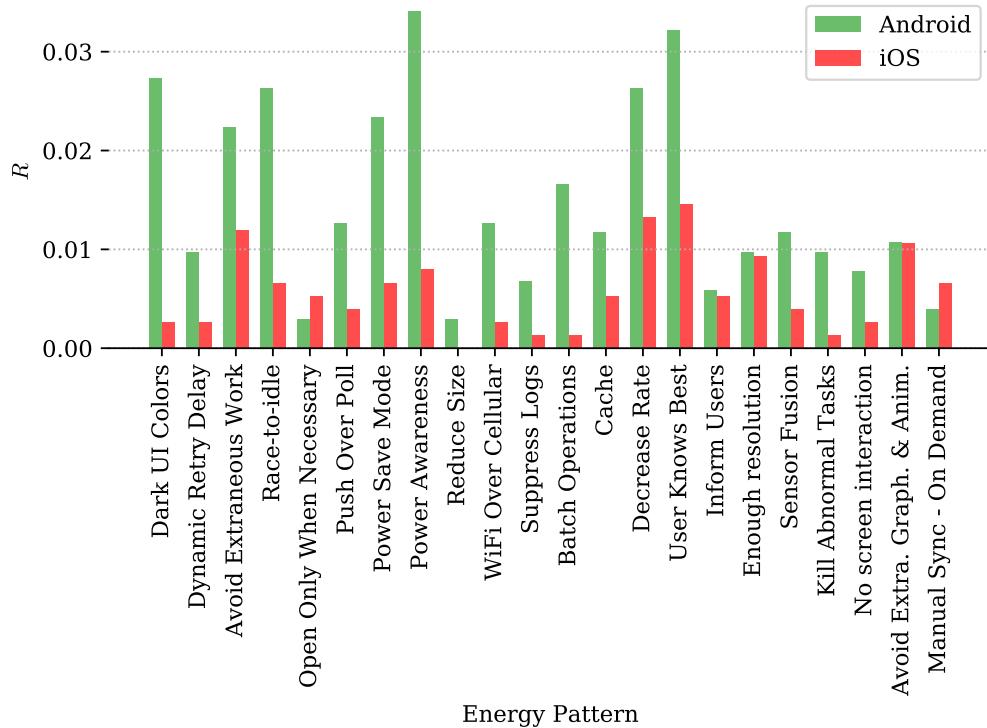


Figure 7.7: Comparison of the usage of energy patterns between Android and iOS mobile apps.

Graphics And Animations had a similar ratio of occurrences in both platforms. The remaining 18 patterns were notably more frequent in Android apps.

The two platforms differ in the patterns that have the highest number of occurrences. In Android, the most frequent patterns were *Power Awareness*, *User Knows Best*, *Dark UI Colors*, and *Race-to-idle*. In iOS, the most frequent patterns were *Avoid Extraneous Work*, *Decrease Rate*, *User Knows Best*, and *Avoid Extraneous Graphics And Animations*. These patterns are being mentioned in the official iOS documentation for developers, reinforcing the importance of documentation to help developers build energy-efficient software [Manotas et al., 2016; Sousa et al., 2018].

The pattern *Dark UI Color* is considerably more popular amongst Android apps than in iOS apps. This is an expected observation: only recently, iOS devices started to feature AMOLED displays that reduce energy consumption when using dark colors¹⁸.

7.5.2 Co-occurrence of Patterns

We analyze which patterns tend to appear together within the same mobile app. We resort to the chord diagram in Figure 7.8. Each pattern is connected to another

¹⁸Until mid 2018, iPhone X was the only iOS smartphone with an AMOLED screen.

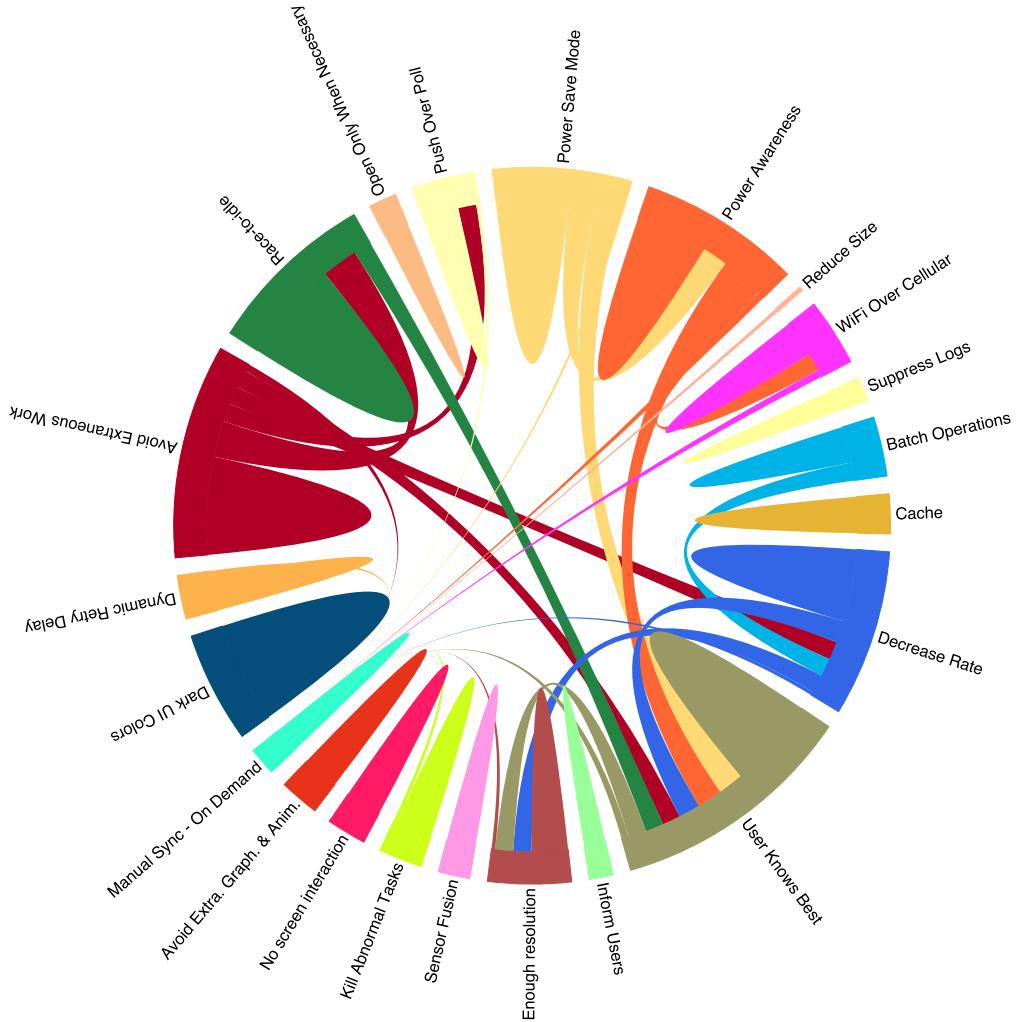


Figure 7.8: Co-occurrence of energy patterns in the same mobile application.

pattern if found in the same app. The thicker the edge the more frequent the pair of patterns co-occur.

To improve the interpretability of the chord diagram we have filtered out cases in which two patterns co-occur less than five times. An interactive version of the diagram containing all data is available in the online catalog: <https://tqrg.github.io/energy-patterns>.

The chord diagram of Figure 7.8 reveals the following relationships between patterns:

- *Dark UI Colors* and *Race-to-idle* are the most prevalent patterns (28 occurrences), followed by *Avoid Extraneous Work* (27 occurrences), *User Knows Best* (25 occurrences), and *Decrease Rate* (24 occurrences).
- *User Knows Best* is a dominant pattern. It co-occurs with *Power Save Mode*, *Power Awareness*, *Decrease Rate*, *Avoid Extraneous Work*, and *Race To Idle*.

- *Avoid Extraneous Work* is also a dominant pattern. It appears in apps that also use *Race-to-Idle*, *Push Over Poll*, *Decrease Rate*, and *User Knows Best*.
- *Power Save Mode*, *Power Awareness*, and *User Knows Best* are typically used together.

7.5.3 Implications

This catalog wraps up the techniques used by developers of open source mobile apps to address energy efficiency. It helps developers understand how to design energy-efficient mobile apps by looking at solutions from other projects. Most techniques are spread out in the literature, making energy efficiency a problem that requires specialized developers. We mitigate this barrier by providing common approaches to solve typical problems in the energy efficiency of mobile apps.

Although we create energy patterns as a platform to share knowledge from experienced developers, it can also serve as the base to future work on automated tools to improve energy efficiency. Patterns such as *Enough resolution* may be challenging, requiring developers to have a deep understanding of the devices they are targeting. Tools or APIs aiding to address this issue would significantly decrease the efforts to adopt this pattern. This is already the case for patterns such as *Batch Operations* that are featured in the platform's API: Android provides the JobScheduler that schedules the jobs to execute at the most efficient times. We understand that similar approaches should be leveraged to other patterns. For instance, *Power Save Mode* is a common pattern that often requires re-implementing existing features.

We also find some remarks of interest to the research community. While some techniques have been widely studied in previous work (e.g., *Dark UI Colors* and *Race-to-idle*), many remain unnoticed. Patterns such as *Dynamic Retry Delay* or *Kill Abnormal Tasks* can be argued based on logical arguments. However, there is no empirical study that has evaluated the cost and benefit of applying these patterns.

This catalog can also help educators include the topic of energy efficiency in Mobile Software Engineering courses. Students can reach to this catalog to seek more information on energy patterns and find examples of their application in real Android and iOS apps.

7.6 Threats to validity

In this section, we now outline the potential threats to the validity of the experiment detailed above.

Internal validity We select all commits, issues and pull requests that contain the words *energy*, *battery*, or *power*. While we understand that this covers the large majority of mentions of improvements in energy usage, some mentions may have been missed. There are commits that may have been created to improve energy efficiency, but the message may not mention it explicitly. Moreover, we only use mentions written in English, which is the most common language amongst developer communities.

In our methodology, we adopt a manual filtering of collected commits, issues, and pull requests. While we understand that this was the safest approach to avoid missing important mentions, human error can be expected from this process. False positives may have been left in the dataset – we argue that during the thematic analysis these subjects are easily discarded. False negatives (i.e., subjects of interest that were filtered out) may also occur due to misinterpretation of subjects by authors during the selection. However, we expect this to comprise a negligible number of cases.

Only commits merged with the *default* branch of projects were considered. Energy improvements that were implemented in different branches of the project were not considered in our study. Although some interesting patterns may lie in some of these branches we cannot guarantee their quality: changes that have not been merged are still lacking the validation from the development team.

In addition, we only collect energy patterns that were used by mobile app development practitioners and are available on open source projects. There might be other patterns that improve energy efficiency but that are not being used by the community. Those patterns are out of the scope of this study and are left for future work. Moreover, patterns that also improve other properties besides energy usage might occur in the projects in this study.

We solely list cases that occur with the main goal of improving energy efficiency. In this study, we only address energy improvements that are clearly described in the description of the respective commit, issue, or pull request. However, the classification of developers' intent in code changes is a non-trivial open problem [Pascarella et al., 2018]. Thus, less obvious energy improvements were not studied. In addition, this catalog is based on developers common approaches to address energy efficiency. We do not measure the magnitude of potential gains of these patterns work. An empirical study would help to assess these gains, as done in previous work [Carette et al., 2017; Palomba et al., 2019].

Finally, we have discarded less significant patterns by filtering out patterns with less than three occurrences. We understand that some of them may hide interesting energy efficiency strategies. However, no literature was found to support these patterns, and assessing their impact is out of the scope of this work.

External validity The data analyzed in this work is typically private for commercial apps. Thus, we focus exclusively on open source mobile apps. Development of commercial apps is usually driven by different goals and budgets. Hence, energy usage may be targeted in a different way in these apps. However, energy patterns collected in this study can be applied in any mobile app project regardless of its license.

We only analyze iOS and Android mobile apps. While this comprises most of the mobile apps in the market, other mobile platforms also have their share (Windows Phone, BlackBerry OS, Firefox OS, Ubuntu Touch, etc.). We discarded coding practices for energy efficiency that we understood being specific to the platform in which they were implemented (e.g., using certain API methods). Unless the mobile operative system or the hardware device deals with the contexts identified in our work under the hood, we expect these patterns to be useful in other platforms.

7.7 Summary

In this chapter, we have studied typical solutions that developers resort to when improving the energy efficiency of their apps. Concretely, in this chapter:

- We conducted an empirical study on 1021 Android apps and 756 iOS apps to identify design practices that improve the energy efficiency of mobile apps (replying RQ 7.1, page 118).
- We catalogued the most common practices in a set of 22 design patterns, coined as *Energy Patterns*.
- We identified related work from academic and grey literature that validate the efficacy of the energy patterns.
- We compared how developers address energy efficiency in Android and iOS – energy efficiency activities are more common amongst Android projects (replying RQ 7.2, page 118).

7.A Grey Literature

- [1] Apple Developer Documentation Archive. Energy Efficiency Guide for iOS Apps – Avoid Extraneous Graphics and Animations. URL: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/AvoidExtraneousGraphicsAndAnimations.html>
- [2] Daniel Gultsch. The State of Mobile XPP in 2016. URL: https://gultsch.de/xmpp_2016.html
- [3] Alternative Push Notification Transport. URL: <https://github.com/matrix-org/GSoC/blob/master/IDEAS.md#alternative-push-notification-transport>

- [4] Apple Developer Documentation Archive. Energy Efficiency Guide for iOS Apps – React to Low Power Mode on iPhones. URL: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/LowPowerMode.html>
- [5] Stackoverflow. Is gzip compression useful for mobile devices?. URL: <https://stackoverflow.com/questions/3065920/is-gzip-compression-useful-for-mobile-devices>
- [6] Mitch Bartlett. Does Wi-Fi Consume More Battery Power Than 3G or 4G/LTE?. URL: <https://www.technipages.com/does-wi-fi-consume-more-battery-power-than-3g-or-4glte>
- [7] Android SDK Documentation. Modifying your Download Patterns Based on the Connectivity Type. URL: https://developer.android.com/training/efficient-downloads/connectivity_patterns
- [8] Apple Developer Documentation Archive. Energy Efficiency Guide for iOS Apps – Energy and Networking. URL: <https://developer.apple.com/library/archive/documentation/Performance/Conceptual/EnergyGuide-iOS/EnergyandNetworking.html>
- [9] Android SDK Documentation. Optimizing for Doze and App Standby. URL: <https://developer.android.com/training/monitoring-device-state/doze-standby>
- [10] Android SDK Documentation. Android Developer Guides — Optimizing for Battery Life. URL: <https://developer.android.com/topic/performance/power/>
- [11] Android Developers Youtube Channel. DevBytes: Efficient Data Transfers - Batching, Bundling, and SyncAdapters. URL: <https://www.youtube.com/watch?v=5onKZcJyJwI>
- [12] Apple's Core Location Documentation. CLLocationManager. URL: <https://developer.apple.com/documentation/corelocation/cllocationmanager>

Impact of Energy Patterns on Mobile App Maintainability



Do Energy-oriented Changes Hinder Maintainability?

Luis Cruz, Rui Abreu, John Grundy, Li Li, and Xin Xia

Submitted to ICSME, 2019.

Abstract

Energy efficiency is a crucial quality requirement for mobile applications. However, improving energy efficiency is far from trivial as developers lack the knowledge and tools to aid in this activity. In this chapter we study the impact of changes to improve energy efficiency on the maintainability of Android applications. Using a dataset containing 539 energy efficiency-oriented commits, we measure maintainability – as computed by the Software Improvement Group’s web-based source code analysis service Better Code Hub (BCH) – before and after energy efficiency-related code changes. Results show that in general improving energy efficiency comes with a significant decrease in maintainability. This is particularly evident in code changes to accommodate the Power Save Mode and Wakelock Addition energy patterns. In addition, we perform manual analysis to assess how real examples of energy-oriented changes affect maintainability. Our results help mobile app developers to 1) avoid common maintainability issues when improving the energy efficiency of their apps; and 2) adopt development processes to build maintainable and energy-efficient code. We also support researchers by identifying challenges in mobile app development that still need to be addressed.

8.1 Introduction

Modern mobile applications, popularly known as apps, provide users with a number of features in multi-purpose mobile computing devices – smartphones. The convenience of using smartphones to pervasively accomplish important daily tasks has a big limitation: smartphones have a limited battery life. Apps that drain battery life of smartphones can ruin user experience, and are likely to be uninstalled unless they offer a key feature.

Thus, it is critically important that apps efficiently use the battery of smartphones. However, many developers still lack knowledge about best practices to deliver energy-efficient mobile applications [Pang et al., 2015; Sahin et al., 2014]. Important efforts have been carried out to help developers ship energy-efficient mobile apps [Kong et al., 2018]. Novel tools have been built to suggest energy improvements to the

codebases of mobile apps [Cruz and Abreu, 2018b; Cruz and Abreu, 2018a; Linares-Vásquez et al., 2018; D. Li et al., 2016] and to help developers measure the energy consumption of their apps [S. Chowdhury et al., 2018a; Boonkrong and Dinh, 2015; S. A. Chowdhury and Hindle, 2016; Di Nucci et al., 2017a; L. Li et al., 2017].

Despite these efforts, improving the energy efficiency of mobile applications is not a trivial task. It requires implementing new features and refactoring existing ones [Cruz and Abreu, 2019a], only for the sake of better energy usage, i.e., predominantly a non-functional rather than functional change. However, the extent to which these changes affect the maintainability of the mobile app software has not yet been studied. In this work, we are interested in studying the trade-off between the energy efficiency and the maintainability of mobile applications.

The International Standards on software quality ISO/IEC 25010 define software maintainability as “the degree of effectiveness and efficiency with which a software product or system can be modified to improve it, correct it or adapt it to changes in environment, and in requirements” [International Organization for Standardization, 2011]. The standard defines five core sub-characteristics of maintainability: modularity, reusability, analyzability, modifiability, and testability. The Software Improvement Group (SIG) has developed a web-based source code analysis toolset *Better Code Hub* (BCH) [Visser et al., 2016] that maps the ISO/IEC 25010 standard on maintainability into a set of 10 guidelines, such as *write short units of code* and *write code once*, derived from static analysis [Kuipers et al., 2007; Baggen et al., 2012; Visser et al., 2016; Olivari, 2018]. The code metrics used by the SIG model were empirically validated in previous work [Bijlsma et al., 2012]. We use this toolset in our work to provide an assessment of maintainability in mobile app codebases.

Specifically, we want to explore whether there is a trade-off between applying energy patterns and keeping the maintainability of the apps, i.e., does improving energy efficiency have a negative impact on code maintainability? In this chapter, we present the results of our analysis on the maintainability using 539 energy commits harvested from open source Android applications.

The key contributions of this work are:

- An empirical investigation of the impact of energy patterns in code maintainability.
- A dataset of energy commits and respective impact on maintainability.
- A software package with all scripts used in our experiments and a dataset of energy commits with respective impact on maintainability, for reproducibility.
Available here: <https://figshare.com/s/989e5102ae6a8423654d>.

Our empirical study finds evidence that energy efficiency-oriented code changes have a negative impact on code maintainability. In particular, careful thinking is required to implement the energy patterns *Power Save Mode* and *Wakelock Addition*. Furthermore, we show that energy patterns are more likely to require maintenance than regular code changes.

This chapter is structured as follows. In Section 8.2, we introduce an example of an energy improvement from a real-world mobile application. Section 8.3 describes the methodology we use to answer the research questions. We present the results in Section 8.4 and discuss their implications in Section 8.5. In section 8.6, we enumerate the threats to the validity of our work. Section 8.7 describes the differences between our work and existing literature. Finally, in Section 8.8 we summarize the main contributions.

8.2 Motivating Example & Research Questions

Improving energy efficiency of apps revolves around changing their codebases. Previous work has studied existing energy patterns for mobile applications [Cruz and Abreu, 2019a]. It cataloged typical coding practices developers adopt to address energy efficiency. An example of an energy pattern is the *Power Save Mode*: the app features a mode that can be activated upon low battery and uses fewer resources while providing the minimum functionality that is indispensable to the user.

An instance of this pattern can be found in the app *NetGuard*¹ – an Android app that provides a firewall and monitors network traffic across other apps.

To improve energy efficiency, *NetGuard*'s developers decided to implement the pattern *Power Save Mode* [Cruz and Abreu, 2019a]. The following snippet presents the code changes that were implemented²:

```
public class SinkholeService extends VpnService {
    private boolean powersaving = false;
    // [snip]

    public void handleMessage(Message msg) {
        if (powersaving) return;❶
        switch (msg.what) {
            case MSG_PACKET:
                log((Packet) msg.obj, msg.arg1, msg.arg2 > 0);
        // [snip]
        }
    }

    // [snip]
```

¹More information about the app *NetGuard* on Google Play app store: <https://play.google.com/store/apps/details?id=eu.faircode.netguard&hl=en> (Visited on June 5, 2020).

²Commit taken from *NetGuard* project's Github repository, available at: <https://github.com/M66B/NetGuard/commit/2e70a038970d6efe9f74e5719e7648f91de30498> (Visited on June 5, 2020)

```

private BroadcastReceiver interactiveStateReceiver =
    new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
// [snip]
        statsHandler.sendMessage(
-            Util.isInteractive(this) ? STATS_START : STATS_STOP
+            Util.isInteractive(this) && !powersaving ?
+                STATS_START : STATS_STOP ②
        );
    }
};

// [snip]

+ private BroadcastReceiver powerSaveReceiver = new BroadcastReceiver() { ③
+     @Override
+     @TargetApi(Build.VERSION_CODES.LOLLIPOP) ④
+     public void onReceive(Context context, Intent intent) {
+         Log.i(TAG, "Received" + intent);
+         Util.logExtras(intent);
+         PowerManager pm = getSystemService(
+             Context.POWER_SERVICE);
+         powersaving = pm.isPowerSaveMode();
+         Log.i(TAG, "Power" + "saving=" + powersaving);
+         statsHandler.sendMessage(
+             Util.isInteractive(this) && !powersaving ?
+                 STATS_START : STATS_STOP ⑤
+         );
+     }
};

// [snip]

@Override
public void onCreate() {
    // [snip]
+    if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) { ⑥⑦
+        PowerManager pm = getSystemService(POWER_SERVICE);
+        powersaving = pm.isPowerSaveMode();
+        IntentFilter ifPower = new IntentFilter();
+        ifPower.addAction(ACTION_POWER_MODE_CHANGED);
+        registerReceiver(powerSaveReceiver, ifPower);
+    }
// [snip]
}

// [snip]

@Override
public void onDestroy() {
    // [snip]
+    if (VERSION.SDK_INT >= VERSION_CODES.LOLLIPOP) ⑧
+        unregisterReceiver(powerSaveReceiver); ⑨
// [snip]
}
}

```

- ❶ Suppress the behavior of data logging methods to suppress output.

- ❷ Deactivate network speed statistics when *Power Save Mode* is activated.
- ❸ An instance of `BroadcastReceiver` is created to implement the handler of Power Save Mode events.
- ❹ A decorator is used to make sure *Power Save Mode* changes are only applied to a compatible Android version.
- ❺ Network speed statistics have to be deactivated upon different events. This is a duplicate of ❷.
- ❻ Subscribe event to be notified when *Power Save Mode* is activated.
- ❼❼ A conditional statement is used to make sure *Power Save Mode* changes are only applied to a compatible Android version.
- ❼ Subscribe *Power Save Mode* event.

Although the concept of creating a *Power Save Mode* is relatively simple, this example illustrates that a number of code changes have to be made that have an adverse impact on code maintainability. For instance, it requires adding duplicated code and adding conditional statements to check the version of Android, increasing cyclomatic complexity. This form of coding goes against some of the guidelines for building maintainable software [Visser et al., 2016].

We are concerned that, while improving energy efficiency, developers are decreasing the maintainability of their projects, and consequently increasing technical debt. In this work, we use a dataset of energy efficiency-oriented changes to measure the difference in maintainability incurred in Android applications when those changes were applied. Therefore, in this work, we want to answer the following research questions.

Research Question 8.1

What is the impact of making code changes to improve energy efficiency on the maintainability of mobile apps?

Why: Energy efficiency often requires to change codebases and even the features of a mobile application. If maintainability is not addressed, these improvements may significantly increase technical debt and require rework during the lifetime of the project.

How: We analyze a combination of previous datasets with 539 energy-oriented commits. We compute the maintainability score of these commits using the online tool BCH. We apply the same approach to a dataset of regular commits to use as baseline and compare results.

Research Question 8.2

Which energy efficiency-oriented code change patterns are more likely to affect the maintainability of mobile apps?

Why: Some energy patterns might be more complex to implement than others. By understanding which patterns are more likely to introduce maintainability issues, we bring awareness to mobile app and mobile SDK developers of code changes that require more attention.

How: We use the classification of developers activities made in previous work [Cruz and Abreu, 2019a; Moura et al., 2015; Bao et al., 2016; Cruz and Abreu, 2018b] to group energy-oriented commits and analyze maintainability independently.

Research Question 8.3

What are typical maintainability issues introduced by energy-oriented code changes?

Why: By using examples of typical maintainability issues in real energy-oriented commits, practitioners and researchers will have a more tangible concept of how energy efficiency may hinder maintainability.

How: First, we select energy-oriented commits that yielded low maintainability. Then, we manually inspect these commits and discuss the potential issues entailed by energy efficiency improvements.

8.3 Methodology

We use the approach illustrated in Fig. 8.1 to analyze how energy commits affect the maintainability of Android applications. It comprises the following steps:

1. Combine the datasets from related work that classifies the activities of developers addressing energy efficiency in mobile apps [Bao et al., 2016; Moura et al., 2015; Cruz and Abreu, 2018b; Cruz and Abreu, 2019a].
2. Collect regular commits from Android apps to be used as baseline.
3. Compute the impact of energy-oriented commits on maintainability, using BCH.

8.3.1 Dataset

Our work uses the data collected in four previous studies [Cruz and Abreu, 2019a; Moura et al., 2015; Bao et al., 2016; Cruz and Abreu, 2018b] to assess the impact of

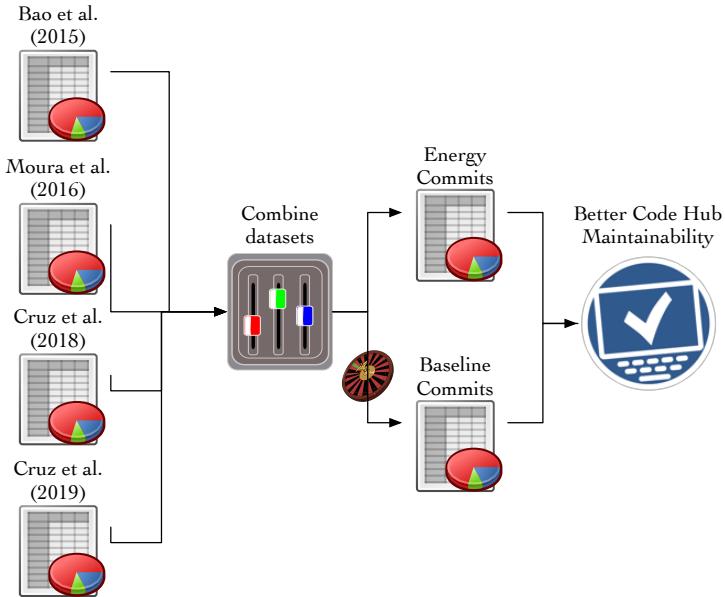


Figure 8.1: Methodology for data collection.

Table 8.1: Datasets of energy-oriented changes that were combined from previous work.

Authors	Ref.	# Commits	Platforms
Moura et al. (2015)	[Moura et al., 2015]	2188	Android, iOS, non-mobile
Bao et al. (2016)	[Bao et al., 2016]	468	Android
Cruz et al. (2018)	[Cruz and Abreu, 2018b]	59	Android
Cruz et al. (2019)	[Cruz and Abreu, 2019a]	431	Android, iOS

energy management-oriented changes on the maintainability of Android software. The datasets are summarized in Table 8.1 and explained below.

Moura et al. (2015) mined more than 2000 commits to understand energy management activities in general-purpose applications [Moura et al., 2015]. Their findings suggest that energy efficiency techniques have to be carefully chosen to ascertain that the correctness of the software remains intact. In an extension of this work [Bao et al., 2016], Bao et al. (2016) used a similar approach to focus exclusively on Android apps, having mined 468 energy management commits. They found that apps in different categories typically have different approaches to energy efficiency.

Cruz et al. (2018) have provided energy patterns in an automatic refactoring tool [Cruz and Abreu, 2018b]. The tool was used to analyze 140 open-source Android apps. As an outcome, the authors submitted 59 pull-requests containing energy improvements to the official repositories of open-source Android applications. In another work, Cruz et al. (2019) proposed a catalog with 22 energy patterns to help developers design energy-efficient mobile applications [Cruz and Abreu, 2019a]. The authors mined the commits, issues, and pull requests of 1027 Android apps and 726 iOS apps to understand how developers address energy efficiency issues. The

catalog can be used to help novice developers learn advanced energy management techniques from existing practices.

From all the data collected, we only select commits from Android projects. Changes from other platforms, such as iOS and Desktop software, were filtered out. Moreover, we cleansed the dataset by filtering out projects that have been deleted and by updating projects that have moved their repositories to a different location. In addition, datasets [Moura et al., 2015] and [Bao et al., 2016] include commits that have not been manually validated – we only include commits that the authors manually ascertained as proper energy changes.

In addition, we reuse the categorization of the energy changes defined in the original datasets. Despite similar, different datasets use different labels to indicate the same pattern. For example, the same pattern is labeled as *PowerConditionalStrategy:PowerSaveMode* by Bao et al. (2016) [Bao et al., 2016] and as *Power Save Mode* by Cruz et al. (2019) [Cruz and Abreu, 2019a]. We map these and other identical categories into unique labels³. In sum, energy commits are classified into seven categories:

- **Bug Fix & Code Refinement.** Changes related to fixing energy bugs, or refactoring code that already implements energy management features.
- **Power Awareness.** Have a different behavior when the device is connected/disconnected to a power station or has different battery levels.
- **Power Save Mode.** Implementation of an energy-efficient mode in which some features are deactivated to improve better energy usage.
- **Power Usage Monitoring.** Developers add UIs or configurations to inform users about the status of the battery and let them make informed decisions about their interaction with the application.
- **Wakelock Addition.** Wakes are used when apps execute tasks that may take longer to execute and need to prevent resources from getting into a sleep state (e.g., screen, network, audio, etc.).
- **Wakelock Optimization.** Inappropriate usage of wakes may incur into unnecessary energy usage. Thus, often developers have to optimize wakelock behavior, or even replace them with other techniques (e.g., event handlers).
- **Miscellaneous.** This comprises several categories of energy commits. Since we perform hypothesis tests to statistically validate results, we need to have at least 20 commits per category. Thus, when a category comprises less than 20 commits, we label it as *Miscellaneous*.

³The whole set of identical categories can be found in the replication package:<https://figshare.com/s/16397140e8183708d248> (Visited on June 5, 2020).

8.3.2 Baseline Commits

Although we want to assess the maintainability of energy commits, there is no evidence in previous work on how regular commits affect the maintainability of Android projects. E.g., if energy-oriented commits hinder maintainability, we need to understand whether this result is in fact different from general purpose commits. Thus, in parallel with energy commits, we also analyze the maintainability of all other commits and use these as a baseline to answer RQ8.1 and RQ8.2.

The baseline dataset is collected as follows: for each energy commit, we obtain all the commits of the respective project and randomly select one. In addition, we randomly select 20 commits to validate that commits are similar in terms of complexity. By using the dataset of energy commits as input for our baseline dataset we make sure that differences in maintainability in the two datasets are not originated by the specificities of different Android projects (e.g., different contribution policies, coding guidelines, app categories, etc.).

8.3.3 Maintainability Analysis

We make use of the Software Improvement Group's web-based source code analysis service *Better Code Hub* (BCH for short⁴) to collect maintainability reports of the projects. BCH delivers a maintainability model based on 10 guidelines [Visser et al., 2016]:

1. **Write short units of code.** Long units are hard to test, reuse, and understand.
2. **Write simple units of code.** Keeping the number of branch points low makes units easier to modify and test.
3. **Write code once.** When code is duplicated, bugs need to be fixed in multiple places, which is inefficient and prone to errors.
4. **Keep unit interfaces small.** Keeping the number of parameters low makes units easier to understand and reuse.
5. **Separate concerns in modules.** Changes in a loosely coupled codebase are much easier to oversee and execute than changes in a tightly coupled codebase. This is computed based on the total fan-in of all methods in a module. Note that a module in Java and other object-oriented languages translates to a class.
6. **Couple architecture components loosely.** Independent components ease isolated maintenance.
7. **Keep architecture components balanced.** Balanced components ease locating code and foster isolation, improving maintenance activities.

⁴Better Code Hub's website available at <https://www.bettercodehub.com/> (Visited on June 5, 2020)

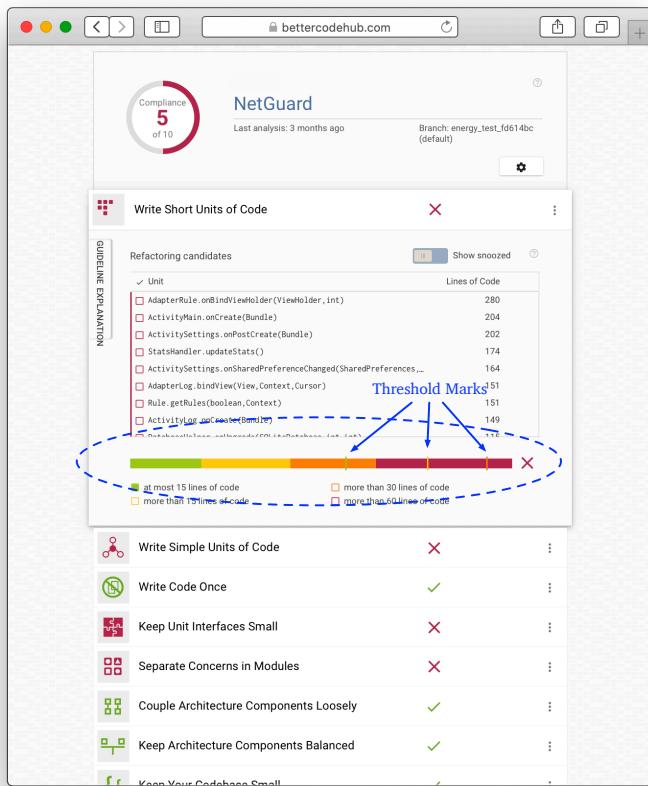


Figure 8.2: BCH's maintainability report of the app *NetGuard* for the guideline *Write short units of code*. The app does not comply with the guideline because the bars are not reaching the threshold marks.

8. **Keep your codebase small.** Small systems are easier to search through, analyze, and understand code.
9. **Automate tests.** Automated testing makes development predictable and less risky.
10. **Write clean code.** Code without code smells is less likely to bring maintainability issues.

For each guideline, BCH evaluates the compliance against a particular guideline by setting boundaries for the percentage of code allowed to fall in each of the four risk severity categories (*low risk*, *medium risk*, *high risk*, and *very high risk*). If the thresholds are not violated, the project is considered to be compliant with the guideline. According to BCH, the guideline thresholds are calibrated yearly based on a representative benchmark of closed and open source software systems. Being compliant with a guideline means that the project under analysis is at least better than 65% of the software systems in BCH's benchmark.

Table 8.2: Example of a BCH report for a non-compliant case.

Level	Threshold (%)	LOC	Percentage of Code (%)
(Low)	(56.3)	(1353)	(18.6)
Medium	43.7	1683	23.2
High	22.3	1622	22.4
Very High	6.9	2588	35.7

The BCH report of the app *NetGuard* for a non-compliant guideline can be seen in Fig. 8.2. This was extracted from the report of the app *NetGuard*, used in the motivating example of Section 8.2. The green bar represents the percentage of compliant lines of code. These lines of code are considered to be compliant with ISO 25010 standard for maintainability [internationale de normalisation, 2011]. The yellow, orange and red bars represent non-compliant lines of code with *medium*, *high*, and *very high* severity levels, respectively. Along the bars, there are also marks that refer to the compliance thresholds for each severity level. The report is equivalent to the information reported in Table 8.2: a set of thresholds, number of lines of code (LOC), and percentage of the project for each severity level. Nonetheless, thresholds provided by BCH do not sum to 100%: non-compliant levels are provided in a cumulative way (e.g., the threshold for the medium level includes high and very high levels); the compliant-level threshold is the complement of the medium-level threshold.

Since we want to analyze maintainability regression, we use BCH to compute maintainability in two different versions of the Android app: a) the version of the project before the energy commit (v_{E-1}) and b) the version immediately after the energy commit (v_E). This is illustrated in Fig. 8.3.

Although BCH provides a detailed report of the maintainability of the project, it does not compute a final score that we can use to compare maintainability amongst different projects. Thus, based in previous work [Olivari, 2018], we designed an equation to capture the distance between the current state of the project and the standard thresholds. We have adjusted the equation to meet the following requirements:

- **The maintainability difference between two versions of the same project is not affected by its size.** In this work, we want to evaluate the identical energy patterns occurring in different projects. Thus, the metric cannot use normalization based on its size – we convert percentage data to the respective number of lines of code.
- **Distance to the thresholds in high severity levels is more penalized than in low severity levels.** We use weights based on the severity level to count lines of code that violate maintainability guidelines.

We compute the mean average of the maintainability score $M(v)$ for all the selected guidelines, as follows:

$$M(v) = \sum_{g \in G} M_g(v) \quad (8.1)$$

where:

G = selected maintainability guidelines from BCH (e.g., *Write short units of code, etc.*)

v = version of the app under analysis.

The maintenance M based on the guideline g for a given version of a project is computed with the following equation:

$$M_g = \frac{1}{|L|} \sum_{l \in L} C(l), \quad L = \{\text{medium, high, veryHigh}\} \quad (8.2)$$

where:

C = compliance with the maintainability guideline for the given severity level (medium, high, and very high)

L = severity levels of maintainability infractions.

The compliance C for a given severity level l is derived by:

$$C(l) = LOC_{compliant}(l) - w(l) \cdot LOC_{\neg compliant}(l) \quad (8.3)$$

where:

$LOC_{compliant}(l)$ = lines of code that comply with the guideline at the given severity level l

$LOC_{\neg compliant}(l)$ = lines of code that do not comply with the guideline at the given severity level l

$w(l)$ = weight factor to boost the impact of non-compliant lines in comparison to compliant lines.

Finally, the term $w(l)$ is calculated as follows:

$$w(l) = \frac{1 - T(l)}{T(l)} \quad (8.4)$$

where:

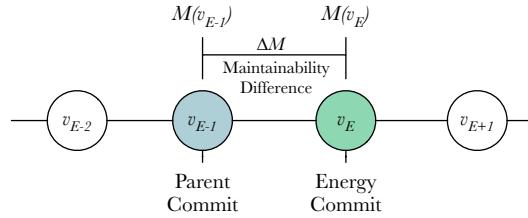


Figure 8.3: Maintainability difference for the energy commit v_E .

$T(l)$ = threshold in percentage of the lines of code that are accepted to be non-compliant with the guideline for the severity level l . This is a standard value defined by BCH, as illustrated in Fig. 8.2 and Table 8.2.

In other words, the factor w is used in Eq. 8.3 to highlight the lines of code that are not complying with the guideline. For instance, the threshold for the severity level *veryHigh* is defined in Table 8.2 as $T(\text{veryHigh}) = 6.9\%$, which derives to a weight of $w(\text{veryHigh}) = 13.5$. This means that, in this example, one non-compliant guideline is decreasing maintainability score by 13.5 points while a compliant guideline is increasing by 1.0 point. In addition, a version that is perfectly aligned with the standard thresholds has a maintainability score of zero.

Then, we compute the difference of maintainability (ΔM) between the energy commit (v_E) and its parent commit (v_{E-1}), as illustrated in Fig. 8.3.

Statistical validation To validate the maintainability differences in different groups of commits (e.g., baseline and energy commits) we use the Paired Wilcoxon signed-rank test with the significance level $\alpha = 0.05$. In other words, we test the null hypothesis that the maintainability difference between pairs of versions v_{E-1} , v_E (i.e., before and after an energy-commit) follows a symmetric distribution around 0. This test does not capture the absolute value of the maintainability differences. Thus, it is not affected by confounding factors, such as the size of the code changes in different groups.

To understand the effect-size, as advocated by the Common-language effect sizes [McGraw and Wong, 1992], we compute the mean difference, the median of the difference, and the percentage of cases that reduce maintainability.

8.3.4 Typical Maintainability Issues

From the results collected in our dataset, we select the most evident examples of maintainability issues that arise from improving energy efficiency. We manually analyze these energy-oriented commits by examining its message and code changes. The most evident cases are then discussed and presented to illustrate common maintainability issues and bring awareness on how to avoid common issues.

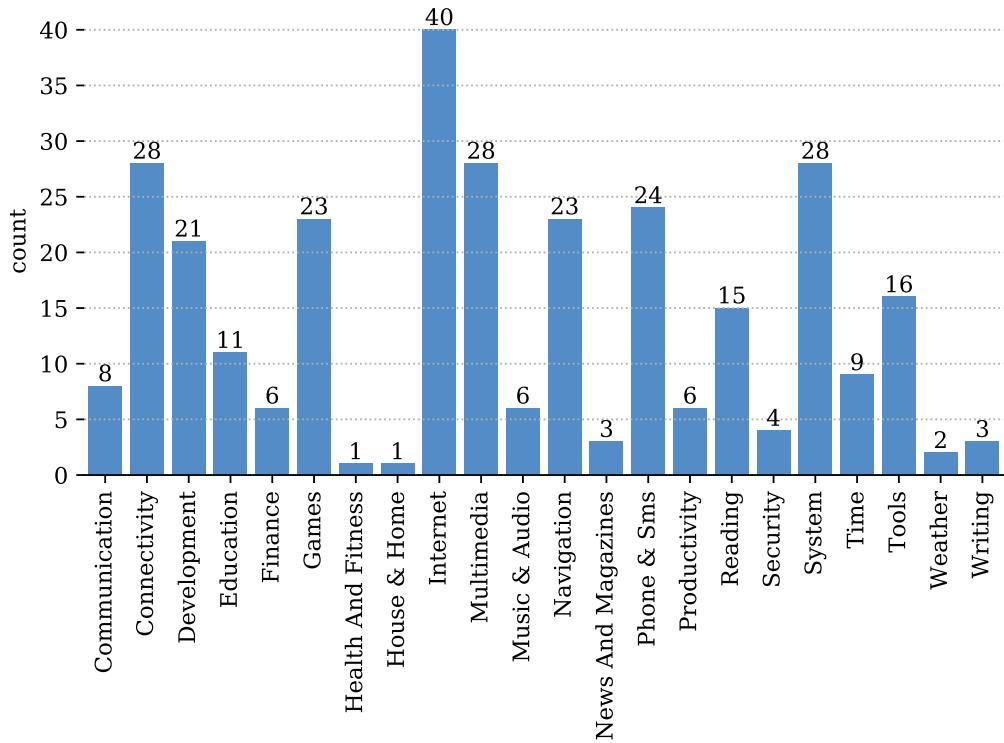


Figure 8.4: Categories of apps included in our study with the corresponding app count for each category.

8.4 Results

We evaluated a total of 539 energy commits and 539 baseline commits. These commits comprise 306 apps distributed among 22 categories, as depicted in Fig. 8.4. In this section, we present the results for each proposed research question.

Research Question 8.1

What is the impact of making code changes to improve energy efficiency on the maintainability of mobile apps?

The results on the impact of different categories of commits in software maintainability are present in the plot bar of Fig. 8.5. The plot presents the results for two groups of software changes: **energy commits**, and **baseline commits**. For each group, the figure provides three bars with the percentage of commits which 1) decrease maintainability (on top, colored in red), 2) do not change maintainability (in the middle, colored in yellow), and 3) increase maintainability (in the bottom, colored in green). In addition, the figure provides, for each group, the mean (\bar{x}) and the median (Md) of the maintainability difference, and the p -value of the Wilcoxon signed-rank test (p).

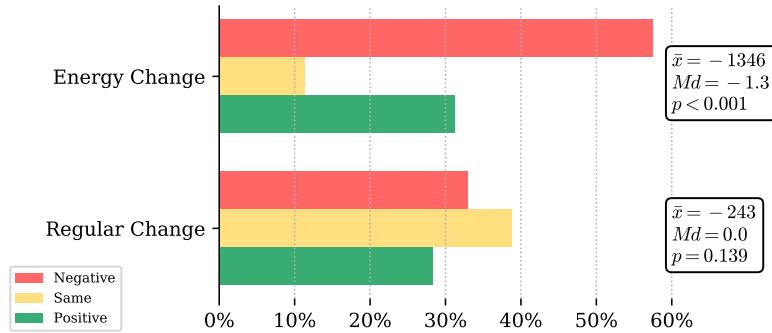


Figure 8.5: Maintainability differences for energy commits and baseline commits.

In the case of the regular commits, used as a baseline, 33.0% decrease maintainability (183 cases), 38.7% do not change maintainability (215 cases), and 28.3% improve maintainability (157 cases). Since the p-value of the Wilcoxon signed-rank test ($p = 0.139$) is not below the significance level ($\alpha = 0.05$), there is no statistical significance of the impact of regular commits on maintainability.

On contrary, we observe clear changes for energy commits: 57.1% (310 cases) decrease software maintainability, 10.7% do not change maintainability (61 cases), and 31.2% improve maintainability (168 cases). The results for the Wilcoxon signed-rank test show statistical significance that energy commits decrease the maintainability of Android applications ($p < 0.001$).

Research Question 8.2

Which energy efficiency-oriented code change patterns are more likely to affect the maintainability of mobile apps?

Results of the maintainability impact per category of energy changes are presented in Fig. 8.6. The Wilcoxon signed-rank test yields statistical evidence that the categories *Miscellaneous* ($p = 0.021$), *Power Save Mode* ($p = 0.012$), and *Wakelock Addition* ($p = 0.003$) significantly decrease the maintainability of Android projects.

The remaining patterns, (i.e., *Bug Fix & Code Refinement*, *Power Awareness*, *Power Usage Monitoring*, and *Wakelock Optimization*) yielded more cases in which maintainability was negatively affected. However, for these patterns, results are not statistically significant.

In the category *Miscellaneous*, 53.1% of changes (77 cases) have decreased maintainability, while 15.9% (23 cases) did not bring any impact, and 31.0% (45 cases) have improved maintainability. The impact is more evident in the category *Power Save Mode*, decreasing maintainability in 78.3% of changes (18 cases), leaving 4.3% unaffected (1 case), and 17.4% (4 cases) with an observed improvement in maintainability. Finally, in the category *Wakelock Addition*, 65.5% have hindered maintainability (55

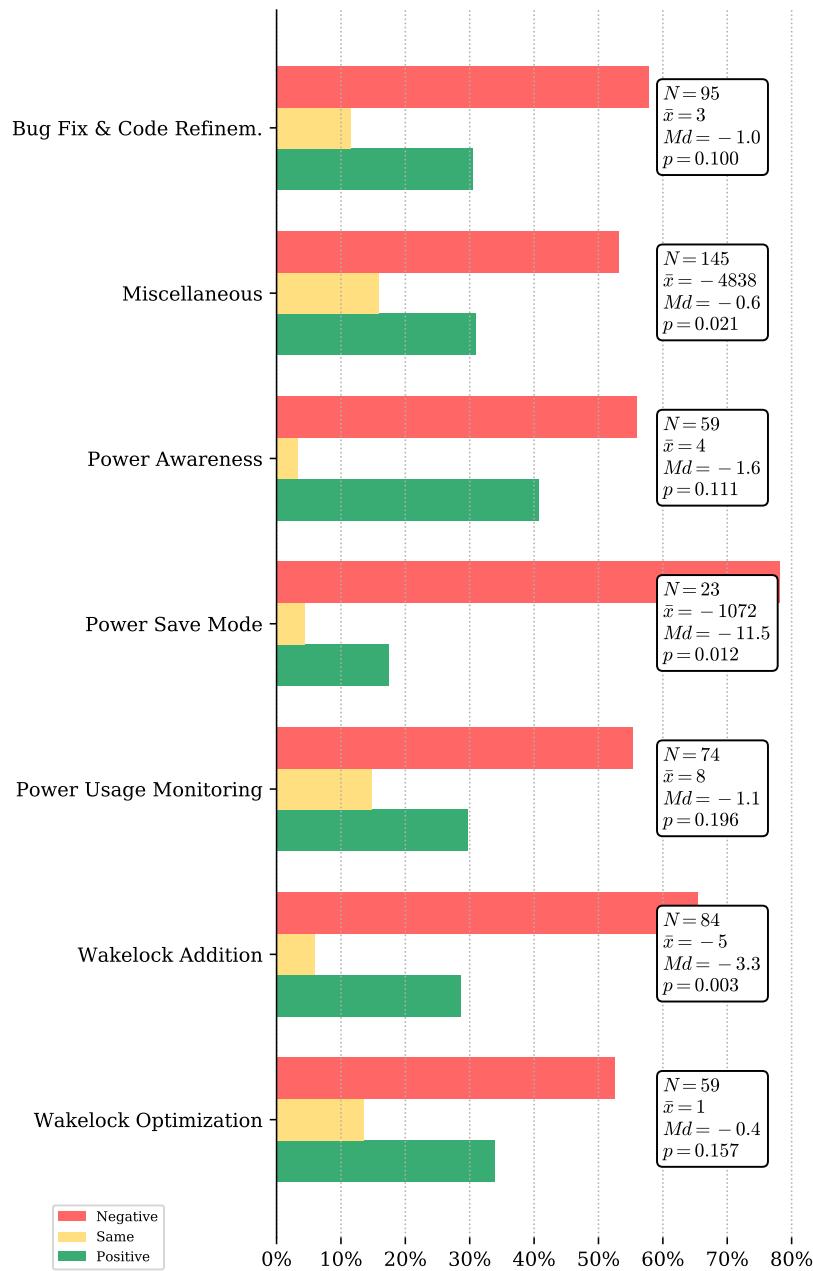


Figure 8.6: Maintainability differences among different types of energy commits.

cases), 6.0% (5 cases) have not yielded any difference, and 28.6% (24 cases) have registered an improvement.

Research Question 8.3

What are typical maintainability issues introduced by energy-oriented code changes?

The following examples illustrate a subset of the maintainability issues we encounter that originate from energy-oriented changes (Maintainability Instances 1–5).⁵

MAINTAINABILITY INSTANCE 1

Git Repository: <https://github.com/ccrama/Slide>

Commit: 070c2c6

Change: Merge two different categories of notifications in the same operation. This is a common approach to improve energy efficiency, coined as *Batch Operations*⁶ [Cruz and Abreu, 2019a].

Maintainability issue ($\Delta M = -949$): While coalescing different tasks, methods ended up being extremely large. As a best practice, Java methods should not go over 15 lines of code [Visser et al., 2016]. Thus, the guideline *Write Short Units of Code* was violated in method `SubredditView.onOptionsItemSelected()`, which ended with 209 lines of code. Several small helper methods should have been implemented to keep this method short.

Maintainability Instance 1 shows an example of maintainability issues that were likely introduced by the lack of awareness by developers on best practices for maintainability. Before applying the code change, the project already had 30 methods with over 200 lines of code. Extracting issues that are strictly related to energy-efficiency improvements is not straightforward. Thus, we skip examples in which this distinction was not clear and opted for selecting maintainability issues that arise from improving energy efficiency in projects with a positive maintainability score.

MAINTAINABILITY INSTANCE 2

Git Repository: <https://github.com/mozilla/MozStumbler>

Commit: 37819d9

Change: New behavior to update the current GPS location. When the user is not moving – i.e., the accelerometer is not sensing any movement – the GPS is turned off and the location is assumed to be constant. When the user moves again, the GPS location updater is reactivated. This instance is an example of energy pattern and *Sensor Fusion* [Cruz and Abreu, 2019a].

Maintainability issue ($\Delta M = -60$): Although this new behavior for GPS updates was added by default in the mobile application, the previous behavior remained as an option in the codebase. This entailed some code duplications: the logic needed to read data from GPS satellites is exactly the same in both behaviors. This violates the *Write Code Once* guideline.

⁵The whole instances can be found in the replication package: <https://figshare.com/s/16397140e8183708d248> (Visited on June 5, 2020)

⁶More information about *Batch Operations* and other energy patterns https://tqrg.github.io/energy-patterns/#/patterns/Batch_Operation (Visited on June 5, 2020).

MAINTAINABILITY INSTANCE 3

Git Repository: <https://github.com/mozilla/MozStumbler>

Commit: 6ea0268

Change: Added support for a power save mode [Cruz and Abreu, 2019a], in which the app stops scanning cell towers and Wi-Fi networks. This change required adding extra logic in the `onCreate` method of `MainActivity` class. In short, the method was changed to verify whether the battery level was low and whether the *Power Save Mode* was enabled in the app.

Maintainability issue ($\Delta M = -20$): Although the idea seems trivial, developers had to add 18 extra lines of code to the already existing `MainActivity.onCreate()` method. The method ended up with 45 lines of code, violating the *Write Short Units of Code* guideline.

MAINTAINABILITY INSTANCE 4

Git Repository: <https://github.com/einmalfel/PodListen>

Commit: 2ed5a65

Change: Add a preference in which users can opt to download new content (i.e., podcasts) only when the smartphone is connected to the charger. This is an implementation of the energy patterns *User Knows Best* and *Power Awareness* [Cruz and Abreu, 2019a].

Maintainability issue ($\Delta M = -20$): By adding this new user-defined setting, conditional logic was added to the beginning of the affected methods (e.g., method `DownloadReceiver.updateDownloadQueue`) to verify the preferences and the phone charging status. This leads to a higher number of branch points per method (maximum recommended of four [Visser et al., 2016]), violating the maintainability guideline *Write Simple Units of Code*. In these cases, the recommended approach is to split the method into simpler ones.

MAINTAINABILITY INSTANCE 5

Git Repository: <https://github.com/horn3t/PerformanceControl>

Commit: cb3080e

Change: Based on the battery level of the smartphone, adjust the power leveraged to CPU and GPU cores. This a very low level code change that resorts to the execution of bash commands to control the hardware of a smartphone device. This example does not implement a documented energy pattern.

Maintainability issue ($\Delta M = -37$): Although the nature of the change implies adding code with poor readability, there are other maintainability issues that should have been avoided. In particular, the class `GPUClass`, which was added to control GPU power, violates the guideline *Separate Concerns in Modules*. The methods of this class have a high number of references through the code (i.e., high fan-in). A typical approach to address this issue is to split the class in separate concerns [Visser et al., 2016].

8.5 Discussion

In this section, we answer our research questions, discussing the implications of the analysis of results.

Research Question 8.1

What is the impact of making code changes to improve energy efficiency on the maintainability of mobile apps?

The majority of energy efficiency-oriented changes hinder the maintainability of Android projects. Results presented in Fig. 8.5 shows a decrease in maintainability in 57% of the cases. This raises a new tradeoff when developers need to address energy efficiency in their projects.

Previous work found evidence that developers struggle to improve the energy efficiency of their software, lacking the knowledge and tools to aid in this problem [Pang et al., 2015]. Our work corroborates by showing that developers may have to reduce maintainability for the sake of energy efficiency.

In our perspective, developers need to be able to create energy-efficient code without potentially ruining the maintainability of their projects. Otherwise, they may not apply such fixes or come with too many negative code maintenance consequences. We understand that this problem needs to be addressed at several levels:

- **Mobile frameworks** need to feature energy patterns out-of-the-box without requiring too many changes in the software codebases.
- **Documentation** of mobile libraries and frameworks need to provide developers with the best practices to implement energy patterns.
- **Programming languages** should provide coding mechanisms to easily implement energy patterns without compromising maintainability. Previous work has already started addressing energy-efficiency concerns in programming languages [R. Pereira et al., 2017b; W. Oliveira et al., 2017]. Hopefully, these

efforts can be ported to the official mobile programming languages (e.g., Java, Kotlin, Swift, etc.).

- **Mobile Developers** have to look out for maintainability issues when implementing energy patterns. Online services such as BCH, that play well in a continuous integration pipeline, can help developers to be more aware of the maintainability issues introduced by their changes. By bringing awareness, developers can put more effort on improving the maintainability of their code and avoid common issues (e.g., code duplication).

Research Question 8.2

Which energy efficiency-oriented code change patterns are more likely to affect the maintainability of mobile apps?

Energy patterns Power Save Mode, and Wakelock Addition significantly decrease the maintainability of Android projects. The same is observed for small energy patterns grouped as Miscellaneous. Although the remaining patterns Bug Fix & Code Refinement, Power Awareness, Power Usage Monitoring, and Wakelock Optimization seem to reduce maintainability, no statistical evidence was found.

This is particularly disconcerting because *Power Save Mode* and *Wakelock Addition* are recommended as power management solutions in the official documentation of the Android SDK⁷. However, it seems that more support is needed in order to implement patterns without compromising the maintainability of Android projects.

Documentation should be enriched with more examples and best practices to implement these patterns. We were not able to find those in the official Android documentation. Moreover, the documentation does not consistently refer to the *Power Save Mode* pattern by this name, referring to it as *Battery Saver* in a few cases⁸.

In addition, different Android versions feature different mechanisms to these patterns. However, developers need to make sure their software runs efficiently in different versions of Android [Muccini et al., 2012; An et al., 2018]. Thus, this requires adding specific logic for each API level, adding more complexity to the code and making it less maintainable.

Along with the implications from RQ8.1, we find that improving support to *Power Save Mode* and *Wakelock Addition* would immediately help developers ship maintainable and energy-efficient mobile software. Actually, tools providing support to

⁷Documentation for *Power Save Mode* and *Wakelocks*: <https://developer.android.com/reference/android/os/PowerManager> (Visited on June 5, 2020).

⁸Android documentation using inconsistent names for *Power Save Mode*: <https://developer.android.com/about/versions/pie/power#battery-saver> (Visited on June 5, 2020).

automatically apply these patterns while preserving maintainability would be of great benefit.

Research Question 8.3

What are typical maintainability issues introduced by energy-oriented code changes?

Although cases with the highest maintainability difference have clear examples of bad maintainability, they are not entirely affected by the energy improvement per se. That is, other factors, such as the low experience level of the developer, may be the cause of the maintainability issues. This problem is illustrated in the Maintainability Instance 1.

On the contrary, the examples presented in Maintainability Instances 2–5 reveal maintainability issues that are intrinsically related to the strategy used to improve energy efficiency. E.g., in the Maintainability Instance 2, developers created an additional approach to collect sensor data but left the original one as an option. Since the efficacy of the two approaches is different, developers decided to feature both approaches in their app: one more effective but less efficient and the other less effective but more efficient. Given that the app needs to run under many different scenarios with different constraints, mobile apps often support different approaches to the same feature. While this decision may be necessary, the nature of these changes is prone to maintainability issues.

In the Maintainability Instance 4, a number of contextual pre-conditions related to the battery level of the smartphone were checked before granting the execution of particular actions. Mobile development frameworks should provide mechanisms to support typical battery-level scenarios out of the box. For instance, using Java annotations, particular actions could be postponed until power-related requirements are met. This would help keep the codebase small and easy to read.

Preliminary related work has proposed programming environments that address energy efficiency [Yıldırım et al., 2018; Zhu et al., 2015]. We show that such solutions are relevant in the context of mobile app development. Moreover, related work has improved the specification of data types to select the most energy-efficient type for a given context [Cohen et al., 2012; Sampson et al., 2011]. Nevertheless, these solutions address energy efficiency decisions at low-level, lacking support for typical design patterns to address the energy efficiency of mobile apps [Cruz and Abreu, 2019a].

The analyzed examples show that maintainability issues lie mainly on the lack of awareness by developers and the insufficient support of energy-efficiency patterns from mobile platforms. New approaches ought to be delivered to help developers assess the maintainability of their code changes when tackling energy-efficiency

requirements. For instance, CI/CD is a promising approach to address this issue. Although it is known to promote software best practices [Zhao et al., 2017], they are rare in the mobile app world [Cruz et al., 2019b; Kochhar et al., 2015]. In addition, results suggest that energy-related changes ought to be tackled by developers with additional care (e.g., code reviews).

8.6 Threats to Validity

In this section, we discuss potential threats to the validity of our work.

Construct Validity We use metrics derived from static code analysis to assess software maintainability. However, this is a broad-scoped attribute that may not be fully capture maintainability in its five sub-characteristics: modularity, reusability, analyzability, modifiability, and testability. Nonetheless, previous work has found high correlation between maintainability sub-characteristics and BCH guidelines [Bijlsma et al., 2012].

In addition, different projects and contexts may require different maintainability standards. Nonetheless, we use statistical hypothesis testing to mitigate confounding factors. Moreover, BCH uses a representative benchmark of closed and open source software systems to compute the thresholds used in each maintainability guideline [Visser et al., 2016; Baggen et al., 2012]. This benchmark is updated every year [Visser et al., 2016].

Internal Validity Maintainability may be affected by different coding styles and experience level from developers of the same project. We do not evaluate differences at that level. In addition, we do not evaluate the maintainability difference for all regular commits in a project. Evaluating all the commits in a project would not be feasible using our methodology. Thus, we assume that the size of the dataset (539 commits) is enough to mitigate random variations in the maintainability differences of the baseline.

The nature of baseline commits scopes general-purpose commits that may be different to energy-oriented commits in a number of characteristics (e.g., lines of code). We ensure the two datasets are comparable by collecting the baseline set using a random selection. Moreover, we do not analyze the maintainability difference in terms of absolute values. In other words, we only evaluate whether the maintainability was improved, not changed, or worsen. In future work, we plan to address specific categories of changes in mobile apps.

External Validity The collection of energy-oriented commits used in this work comprises open source apps. Our methodology requires access to data that is not publicly available for commercial apps. The extent to which this findings generalize to commercial apps with non-open source licenses is not assessed. Still, the maintainability challenges pinpointed in our work are relevant to mobile app projects regardless of their license.

We only analyze Android apps. Different platforms and programming languages may require different coding practices to address energy efficiency. We did not study how our findings generalize to other mobile platforms.

We resort to a set of energy changes that were collected from four previous works [Moura et al., 2015; Bao et al., 2016; Cruz and Abreu, 2018a; Cruz and Abreu, 2019a]. These works use the commit message provided by developers to classify a given commit as an energy change. This approach discards energy changes that did not have a commit message describing them as such. Since extending our datasets to these commits is not trivial, we limit the scope of this study to energy-oriented commits with an explicit commit message. Finally, all the energy commits in this work are described in English.

8.7 Related Work

In this section, we discuss related works on code maintainability, energy patterns, and anti-pattern detection.

8.7.1 Code maintainability

Previous work has studied the evolution of maintainability issues during the development of Android apps [Malavolta et al., 2018]. The authors have observed that maintainability decreases over time, being code duplication the most common maintainability issue. In addition, they found evidence for the fact that maintainability issues in Android apps occur independently of the type of development activities performed by developers. Their work uses a dataset from related work [Pascarella et al., 2018] with an under-represented sample of energy activities, counting with only 12 occurrences. In this work, we focus on a larger sample, counting with 539 energy activities to analyze how energy activities affect the maintainability of Android projects.

A use case study on the Java framework *JHotDraw* suggests that the adoption of design patterns is highly correlated with the maintainability of a project – i.e., the usage of design patterns do improve code maintainability [Hegedűs et al., 2012]. On contrary, related work shows that some design patterns should be used with caution, since they may bring maintainability and evolution issues to software

projects [Khomh and Gueheneuce, 2008]. Our work studies how these findings apply in the case of energy patterns, for open-source Android apps.

Previous work studied the effect of programming languages in the quality of code found [Ray et al., 2014]. It was found that language design has a significant yet moderate impact on software quality. The authors have used the number of defects as a construct of software quality. Our work analyzes software quality in terms of code maintainability to study how it is affected by energy efficiency-oriented commits.

8.7.2 Energy patterns

Previous works have studied the impact of different energy patterns on mobile apps. Offloading heavy computation tasks to a cloud server was found to reduce energy consumption up to 50% in mobile apps [Kwon and Tilevich, 2013]. Other patterns comprise featuring dark user interface themes achieve better energy usage on mobile devices [Agolli et al., 2017; Linares-Vásquez et al., 2017a]. Other approaches have improved energy efficiency by finding the optimal number of display updates in a mobile app [Kim et al., 2016]. Another work has used regular expression representations to ensure an optimal usage of the energy intensive resources of mobile devices [Banerjee and Roychoudhury, 2016].

The impact of logging practices of developers on the energy consumption of Android apps has also been studied [S. Chowdhury et al., 2018b]. From the 24 Android apps in this study, 19 exhibited at least one version in which logging statements had a medium or large effect size on energy consumption.

Energy patterns for mobile apps have been widely studied in the literature [Cruz and Abreu, 2019a; Bao et al., 2016; Moura et al., 2015]. Our work acknowledges the importance of using energy patterns to improve energy efficiency. However, we take a step further and study the impact of these patterns on the quality of the app in terms of code maintainability. In addition, we study the change-proneness of these techniques in mobile app codebases.

8.7.3 Detecting Anti-patterns in Mobile Apps

Related work has studied how anti-patterns affect the overall energy consumption of Android apps. Previous work on 60 Android apps have studied the influence of 9 Android-specific code smells on energy efficiency. Results showed energy savings up to 87 times after fixing all code smells [Palomba et al., 2019]. Another work has studied the impact of eight performance-based code smells on the energy efficiency of mobile apps [Cruz and Abreu, 2017]. It was found significant differences, up to 5%, on energy consumption by fixing five of the studied code smells. Not only code smells have been studied in this context. The impact of picture smells on

energy usage has also been assessed [Carette et al., 2017]. It was found evidence that significant energy savings incur from using an optimal image compression and format.

These works endorse the importance of using refactoring techniques to improve energy efficiency. In fact, anti-pattern detectors and automatic refactoring tools have been delivered to help developers ship energy-efficient code [Morales et al., 2018]. Cruz et al. have implemented an automatic refactoring tool for Android apps to fix five performance issues that also increase energy usage [Cruz and Abreu, 2018b]. Palomba et al. proposed an automated tool to identify 15 Android-specific code smells [Palomba et al., 2017]. These code smells had been flagged by previous work as a potential threat to the maintainability and the efficiency of Android apps [Reimann et al., 2014]. Our work differs by 1) identifying code changes that hinder maintainability and 2) using code changes that already been labeled has an energy improvement.

8.8 Summary

In this chapter, we have studied what is the impact of improving energy efficiency in the maintainability of mobile software projects. Concretely, in this chapter:

- We merged a dataset of Android app energy commits.
- We investigated the impact of applying energy patterns in the code maintainability of open-source Android apps.
 - We have found evidence that energy-oriented commits significantly decrease software maintainability in open source Android apps: 57% of energy commits were observed to reduce code maintainability.
 - We show that the change on maintainability is more evident for the patterns *Power Save Mode* and *Wakelock Addition*, in which maintainability decreases in 78% and 66% of cases.
- We perform manual analysis to assess how real examples of energy-oriented changes affect maintainability.
- We discuss practical implications of the maintainability issues entailed by energy changes.
- We delivered a reproducibility package, available here: <https://figshare.com/s/989e5102ae6a8423654d>.

Conclusions

“One never notices what has been done; one can only see what remains to be done.”

— Marie Curie

Throughout this thesis, we have studied and extended the state-of-the-art approaches to create energy-efficient mobile software. We have detailed existing energy measurement techniques, defined energy efficiency best practices, developed tools to help practitioners and researchers, and studied potential shortcomings of delivering energy-efficient software. Our contributions in this thesis were motivated by the lead research question proposed in (cf. Section 1.2):

Main Research Question

What are the inherent limitations of state-of-the-art approaches to improving the energy efficiency of mobile applications, and what can be done to help developers address it?

We have found that existing mobile apps lack adequate testing strategies, implying extra challenges for energy tests. In addition, we leveraged new best practices to develop energy-efficient mobile software and delivered tools to aid developers in adopting them.

We answer the main research question by dividing it into the five research goals presented earlier in this thesis (cf. Section 1.2). In this chapter, we summarize the work performed for each research goal, with respective main findings. Then, we outline the main contributions of this thesis. Finally, based on the main implications of our work, we make recommendations on future work that is worth investigating.

9.1 Research Goals

In this section, we go over each proposed research goals and summarize our main findings with respective implications.

Research Goal I

Describe the state-of-the-art approaches to measure the energy efficiency of mobile applications

Measuring the energy consumption of mobile apps is a complex task that requires a thorough analysis to ensure the reliability of results. In Chapter 2 we describe the state-of-the-art approaches to accomplish this task. We pinpoint the benefits and drawbacks of using energy profilers or alternatively power monitors. As a contribution, this chapter aims to be a guide to researchers and practitioners that need to create energy tests. In addition, we use this guide to design the experimental setup throughout this thesis.

Research Goal II

Study which UI testing frameworks can provide reliable energy efficiency assessments and check if existing mobile application projects are ready for automated energy testing approaches

In Chapter 3, we present a thorough analysis of the limitations of the state-of-the-art UI frameworks when used to measure the energy consumption of mobile apps. Despite being a fundamental tool for energy tests, we have found that most frameworks are not ready yet. Significant overheads are being imposed by UI frameworks in the energy consumption collected during measurements. This is a fundamental threat to energy test results. As a rule of thumb, we recommend using the framework *Espresso*. Nevertheless, we show that an analysis of the characteristics of the project should be considered before selecting a framework.

As a side contribution, in Chapter 4, we study how existing FOSS Android apps are testing their apps. We assess what practices these apps use, to understand how energy testing can be integrated into these projects. We have found that these apps lack adequate testing practices. Roughly 60% of the apps are not being tested. The upside is that the most popular UI framework is *Espresso*, being used by 15% of the projects. Moreover, we show that, as in other types of software, testing increases the quality of apps (demonstrated in the number of code issues). Amongst the implications of these findings, we highlight the importance of providing clear up-to-date documentation of testing frameworks, and the influence of *Google* on the technologies adopted by Android developers.

Research Goal III

Study and document best practices and recurrent solutions that can be reused to improve the energy efficiency of mobile apps.

We address this research goal with two different strategies: 1) we study existing best practices to improve the performance of mobile apps (cf. Chapter 5), and 2) we empirically assess recurrent strategies created by experienced developers to manage the energy efficiency of their apps (cf. Chapter 7).

In Chapter 5, we analyze the impact of eight performance-based code smells in the energy efficiency of Android apps. We show that fixing five of these code smells, viz. *ViewHolder*, *DrawAllocation*, *WakeLock*, *ObsoleteLayoutParams*, and *Recycle*, leads to more energy-efficient mobile applications. Savings go up to an hour of battery life.

In Chapter 7, we create a catalog of 22 energy patterns, as a result of analyzing recurrent fixes for energy efficiency in 1027 Android and 756 iOS apps. This catalog will help mobile app designers and developers make educated decisions when building (energy-efficient) apps, regardless of the target platform. As a side contribution, we have found that the Android community is more energy-aware. The catalog is available online for all the community: <https://tqrg.github.io/energy-patterns/>.

Research Goal IV

Develop static analysis and automatic refactoring tools to help practitioners create energy-efficient mobile apps

In Chapter 6, we propose the automatic refactoring tool *Leafactor* to apply energy-efficiency refactorings. We implement refactorings for the five code smells discovered in Chapter 5 and validate *Leafactor* with 140 FOSS apps. Results show the importance of using tools to aid developers in complying with energy best practices: using *Leafactor* we were able to fix energy code smells in 45 out of the 140 (32%) Android apps in the dataset.

Research Goal V

Assess the impact of improving energy efficiency in the maintainability of mobile software projects

In Chapter 8, we study the potential effect of addressing energy efficiency in code maintainability. We have found that, while improving energy efficiency, practitioners tend to decrease the maintainability of their projects. This is particularly evident for the patterns *Power Save Mode* and *Wakelock Addition*. We manually investigate and discuss typical maintainability issues entailed by improving the energy efficiency of their apps. Our findings show that mobile frameworks ought to feature energy patterns without entangling difficult code changes. In addition, we suggest that code auditing tools should be used during development activities to prevent such issues. For instance, we recommend adopting a CI/CD pipeline.

9.2 Recommendations for Future Research

Energy Measurements as a Service In Chapter 2, we presented the challenges of creating reliable energy tests. It serves as clarification of the methodology used in

the energy measurements performed throughout this thesis. Moreover, it provides a self-contained guide for researchers and developers that want to test the energy consumption of their apps. As future research, we propose a cloud-based service to simplify energy tests for developers and researchers. The idea is to use peer-to-peer cloud computing to deliver reliable energy measurements as a service.

The system would allow users with power monitors to share their setup with other developers through the cloud. To make efficient use of power monitors, the system would also resort to estimations from energy profilers. Power monitors would only be used whenever the estimations were found to be unreliable. In other words, the system would combine energy models with power monitors to provide the most accurate energy measurements without requiring a cumbersome setup of power tools. We provide more details about this research direction in our ICSE paper [Cruz and Abreu, 2019b].

Mobile Testing Culture In Chapter 4 we studied the prevalence of automated tests in Android FOSS projects. As future work, our empirical study can be expanded in several ways: 1) study how mobile app projects address tests for particular types of requirements (e.g., security, privacy, energy efficiency, etc.); 2) based on the test practices collected from mobile app repositories, provide a set of best practices to serve as rule of thumb for other developers; and 3) verify that these findings also hold for other platforms.

Automatic Refactoring for Energy Efficiency The automatic refactoring approach used in Chapter 6 showed great potential to help developers improve the energy efficiency of their apps. As future work, it would be interesting to support energy patterns from related work [Reimann et al., 2014; Reimann and Aßmann, 2013]. In addition, we plan to study how the energy patterns studied in Chapter 7 can be applied using automated tools.

Moreover, we would like to explore how automatic refactoring can be applied using CI/CD pipelines. The integration would require two distinct steps: 1) one for the detection and 2) another for the code refactoring which would only be applied upon a granting action by a developer. This idea could also be applied for educational purposes. A detailed explanation of the code transformation along with its impact on energy efficiency would be provided whenever a developer pushes new changes to the repository.

Energy Patterns A substantial contribution of this thesis is the catalog of 22 energy patterns. We would like to study these patterns in the context of Cyber-Physical Systems and **Internet of Things (IoT)** applications, in which power consumption has been identified as a challenge to be addressed [White et al., 2010; Palattella et al., 2016]. Finally, we plan to continuously extend the catalog with a broader set

of energy patterns. We welcome contributions from the mobile app development community as pull requests in our online repository.

Impact of Energy Patterns in Software Quality Our empirical study on 539 energy-oriented changes of open source Android apps shows that these types of changes can hinder maintainability (cf. Chapter 8). As future work, this study can be extended in different ways: analyze which maintainability guidelines are more affected from energy commits; analyze how results stand for different categories of mobile apps; expand our methodology with other software quality properties (e.g., reliability). Furthermore, it would be interesting to validate our findings with other mobile platforms (e.g., iOS), and also with desktop and server applications.

Bibliography

- Abdulsalam, Sarah, Ziliang Zong, Qijun Gu, and Meikang Qiu (2015). “Using the Greenup, Powerup, and Speedup metrics to evaluate software energy efficiency”. In: *Green Computing Conference and Sustainable Computing Conference (IGSC), 2015 Sixth International*. IEEE, pp. 1–8 (Cited on page 26).
- Abogharaf, Abdulhakim, Rajesh Palit, Kshirasagar Naik, and Ajit Singh (2012). “A methodology for energy performance testing of smartphone applications”. In: *Proceedings of the 7th International Workshop on Automation of Software Test*. IEEE Press, pp. 110–116 (Cited on page 13).
- Aditya, Paarijaat, Viktor Erdélyi, Matthew Lentz, Elaine Shi, Bobby Bhattacharjee, and Peter Druschel (2014). “Encore: Private, context-based communication for mobile social apps”. In: *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, pp. 135–148 (Cited on page 13).
- Agolli, Tedis, Lori Pollock, and James Clause (2017). “Investigating decreasing energy usage in mobile apps via indistinguishable color changes”. In: *Mobile Software Engineering and Systems (MOBILESoft), 2017 IEEE/ACM 4th International Conference on*. IEEE, pp. 30–34 (Cited on pages 115, 127, 168).
- Amalfitano, Domenico, Nicola Amatucci, Atif M Memon, Porfirio Tramontana, and Anna Rita Fasolino (2017). “A general framework for comparing automatic testing techniques of Android mobile apps”. In: *Journal of Systems and Software* 125, pp. 322–343 (Cited on page 50).
- An, Kijin, Na Meng, and Eli Tilevich (2018). “Automatic Inference of Java-to-swift Translation Rules for Porting Mobile Applications”. In: *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. MOBILESoft ’18. Gothenburg, Sweden: ACM, pp. 180–190 (Cited on page 164).
- Baggen, Robert, José Pedro Correia, Katrin Schill, and Joost Visser (2012). “Standardized code quality benchmarking for improving software maintainability”. In: *Software Quality Journal* 20.2, pp. 287–307 (Cited on pages 146, 166).
- Banerjee, Abhijeet, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury (2014). “Detecting energy bugs and hotspots in mobile apps”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, pp. 588–598 (Cited on page 119).

- Banerjee, Abhijeet, Hai-Feng Guo, and Abhik Roychoudhury (2016). “Debugging energy-efficiency related field failures in mobile apps”. In: *IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft*. Vol. 16 (Cited on pages 23, 95).
- Banerjee, Abhijeet and Abhik Roychoudhury (2016). “Automated Re-factoring of Android Apps to Enhance Energy-efficiency”. In: *Proceedings of the International Conference on Mobile Software Engineering and Systems*. MOBILESoft ’16. Austin, Texas: ACM, pp. 139–150 (Cited on pages 12, 23, 96, 114, 127, 129, 130, 168).
- Bao, Lingfeng, David Lo, Xin Xia, Xinyu Wang, and Cong Tian (2016). “How android app developers manage power consumption?: An empirical study by mining power management commits”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, pp. 37–48 (Cited on pages 119, 124, 127, 150–152, 167, 168).
- Bavani, Raja (2012). “Distributed agile, agile testing, and technical debt”. In: *IEEE software* 29.6, pp. 28–33 (Cited on page 57).
- Beck, Kent (2000). *Extreme programming explained: embrace change*. addison-wesley professional (Cited on page 67).
- Behrouz, Reyhaneh Jabbarvand, Alireza Sadeghi, Joshua Garcia, Sam Malek, and Paul Ammann (2015). “Ecodroid: An approach for energy-based ranking of android apps”. In: *Green and Sustainable Software (GREENS), 2015 IEEE/ACM 4th International Workshop on*. IEEE, pp. 8–14 (Cited on page 119).
- Bijlsma, Dennis, Miguel Alexandre Ferreira, Bart Luijten, and Joost Visser (2012). “Faster issue resolution with higher technical quality of software”. In: *Software quality journal* 20.2, pp. 265–285 (Cited on pages 146, 166).
- Bird, Christian, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu (2009). “The promises and perils of mining git”. In: *Mining Software Repositories, 2009. MSR’09. 6th IEEE International Working Conference on*. IEEE, pp. 1–10 (Cited on page 50).
- Boonkrong, Sirapat and Pham Cao Dinh (2015). “Reducing battery consumption of data polling and pushing techniques on Android using GZip”. In: *Information Technology and Electrical Engineering (ICITEE), 2015 7th International Conference on*. IEEE, pp. 565–570 (Cited on pages 127, 146).
- Bragazzi, Nicola Luigi and Giovanni Del Puente (2014). “A proposal for including nomophobia in the new DSM-V”. In: *Psychology research and behavior management* 7, p. 155 (Cited on page 2).
- Brooks, Margaret E, Dev K Dalal, and Kevin P Nolan (2014). “Are common language effect sizes easier to understand than traditional effect sizes?” In: *Journal of Applied Psychology* 99.2, p. 332 (Cited on page 64).

- Brouwers, Niels, Marco Zuniga, and Koen Langendoen (2014). “NEAT: a novel energy analysis toolkit for free-roaming smartphones”. In: *Proceedings of the 12th ACM conference on embedded network sensor systems*. ACM, pp. 16–30 (Cited on page 13).
- Cai, Huaqian, Ying Zhang, Zhi Jin, Xuanzhe Liu, and Gang Huang (2015). “Delay-Droid: Reducing Tail-Time Energy by Refactoring Android Apps”. In: *Proceedings of the 7th Asia-Pacific Symposium on Internetware*. ACM, pp. 1–10 (Cited on pages 127, 133).
- Cao, Yi, Javad Nejati, Muhammad Wajahat, Aruna Balasubramanian, and Anshul Gandhi (2017). “Deconstructing the Energy Consumption of the Mobile Page Load”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 1.1, 6:1–6:25 (Cited on pages 19, 39, 42).
- Carette, Antonin, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy (2017). “Investigating the Energy Impact of Android Smells”. In: *24th International IEEE Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, p. 10 (Cited on pages 19, 39, 42, 96, 142, 169).
- Cassee, Nathan, Gustavo Pinto, Fernando Castor, and Alexander Serebrenik (2018). “How Swift Developers Handle Errors”. In: *15th International Conference on Mining Software Repositories (MSR 2018)* (Cited on page 120).
- Chen, Xiang, Yiran Chen, Zhan Ma, and Felix CA Fernandes (2013). “How is energy consumed in smartphone display applications?” In: *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*. ACM, p. 3 (Cited on page 95).
- Choudhary, Shauvik Roy, Alessandra Gorla, and Alessandro Orso (2015). “Automated test input generation for Android: Are we there yet? (E)”. In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, pp. 429–440 (Cited on pages 23, 43, 50).
- Chowdhury, Shaiful Alam and Abram Hindle (2016). “Greenoracle: Estimating software energy consumption with energy measurement corpora”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. ACM, pp. 49–60 (Cited on pages 11, 13, 146).
- Chowdhury, Shaiful, Stephanie Borle, Stephen Romansky, and Abram Hindle (2018a). “GreenScaler: training software energy models with automatic test generation”. In: *Empirical Software Engineering*, pp. 1–44 (Cited on pages 4, 11, 119, 146).
- Chowdhury, Shaiful, Silvia Di Nardo, Abram Hindle, and Zhen Ming Jack Jiang (2018b). “An exploratory study on assessing the energy impact of logging on Android applications”. In: *Empirical Software Engineering* 23.3, pp. 1422–1456 (Cited on pages 127, 132, 168).

- Cohen, Michael, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu (2012). “Energy types”. In: *ACM SIGPLAN Notices*. Vol. 47. 10. ACM, pp. 831–850 (Cited on page 165).
- Coppola, Riccardo (2017). “Fragility and evolution of Android test suites”. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 405–408 (Cited on pages 42, 57).
- Coppola, Riccardo, Maurizio Morisio, and Marco Torchiano (2017). “Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility”. In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, pp. 22–32 (Cited on page 50).
- Coppola, Riccardo, Emanuele Raffero, and Marco Torchiano (2016). “Automated mobile UI test fragility: an exploratory assessment study on Android”. In: *Proceedings of the 2nd International Workshop on User Interface Test Automation*. ACM, pp. 11–20 (Cited on page 42).
- Corral, Luis and Ilenia Fronza (2015). “Better code for better apps: a study on source code quality and market success of Android applications”. In: *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press, pp. 22–32 (Cited on page 50).
- Corral, Luis, Anton B Georgiev, Andrea Janes, and Stefan Kofler (2015). “Energy-aware performance evaluation of Android custom kernels”. In: *Green and Sustainable Software (GREENS), 2015 IEEE/ACM 4th International Workshop on*. IEEE, pp. 1–7 (Cited on pages 127, 133).
- Cosentino, Valerio, Javier Luis Cánovas Izquierdo, and Jordi Cabot (2016). “Findings from GitHub: methods, datasets and limitations”. In: *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*. IEEE, pp. 137–141 (Cited on page 50).
- Couto, Marco, Tiago Carção, Jácome Cunha, João Paulo Fernandes, and João Saraiva (2014). “Detecting anomalous energy consumption in Android applications”. In: *Brazilian Symposium on Programming Languages*. Springer, pp. 77–91 (Cited on pages 11, 114).
- Couto, Marco, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva (2015). “GreenDroid: A tool for analysing power consumption in the android ecosystem”. In: *2015 IEEE 13th International Scientific Conference on Informatics*. IEEE, pp. 73–78 (Cited on page 11).
- Cruz, Luis and Rui Abreu (2017). “Performance-based Guidelines for Energy Efficient Mobile Applications”. In: *IEEE/ACM International Conference on Mobile Software Engineering and Systems, MobileSoft*, pp. 46–57 (Cited on pages 8, 13, 37, 42, 77, 119, 120, 127, 129, 168).

- (2018a). “Measuring the Energy Footprint of Mobile Testing Frameworks”. In: *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE*, pp. 400–401 (Cited on pages 8, 12, 15, 19, 57, 146, 167).
- (2018b). “Using Automatic Refactoring to Improve Energy Efficiency of Android Apps”. In: *XXI Ibero-American Conference on Software Engineering (CIBSE, Best Paper Award)* (Cited on pages 8, 42, 99, 119, 146, 150, 151, 169).
- (Mar. 2019a). “Catalog of energy patterns for mobile applications”. In: *Empirical Software Engineering* (Cited on pages 8, 114, 117, 146, 147, 150–152, 161, 162, 165, 167, 168).
- (2019b). “EMaaS: Energy Measurements as a Service for Mobile Applications”. In: *ICSE (NIER)* (Cited on page 174).
- (2019c). “Improving Energy Efficiency Through Automatic Refactoring”. In: *Submitted to Journal of Software Engineering Research and Development* (Cited on page 99).
- (2019d). “On the Energy Footprint of Mobile Testing Frameworks”. In: *Submitted to IEEE Transactions on Software Engineering* (Cited on page 19).

Cruz, Luis, Rui Abreu, John Grundy, Li Li, and Xin Xia (2019a). “Do Energy-oriented Changes Hinder Maintainability?” In: *Submitted to ICSME* (Cited on page 145).

Cruz, Luis, Rui Abreu, and David Lo (Apr. 2019b). “To the attention of mobile software developers: guess what, test your app!” In: *Empirical Software Engineering* (Cited on pages 8, 45, 166).

Cruz, Luis, Rui Abreu, and Jean-Noël Rouvignac (2017). “Leafactor: Improving Energy Efficiency of Android Apps via Automatic Refactoring”. In: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems. MOBILESoft ’17*. Buenos Aires, Argentina: IEEE Press, pp. 205–206 (Cited on pages 8, 99).

Cuadrado, Félix and Juan C Dueñas (2012). “Mobile application stores: success factors, existing approaches, and future developments”. In: *IEEE Communications Magazine* 50.11, pp. 160–167 (Cited on page 138).

Das, Teerath, Massimiliano Di Penta, and Ivano Malavolta (2016). “A Quantitative and Qualitative Investigation of Performance-Related Commits in Android Apps”. In: *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, pp. 443–447 (Cited on page 49).

Di Nucci, Dario, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia (2017a). “PETrA: a software-based tool for estimating the energy profile of android applications”. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 3–6 (Cited on pages 11, 114, 146).

- Di Nucci, Dario, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia (2017b). “Software-based energy profiling of Android apps: Simple, efficient and reliable?” In: *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, pp. 103–114 (Cited on pages 19, 23, 119).
- Diop, Tahir, Natalie Enright Jerger, and Jason Anderson (2014). “Power modeling for heterogeneous processors”. In: *Proceedings of workshop on general purpose processing using GPUs*. ACM, p. 90 (Cited on page 9).
- Dong, Mian and Lin Zhong (2011). “Self-constructive high-rate system energy modeling for battery-powered mobile systems”. In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM, pp. 335–348 (Cited on page 12).
- Ebert, Jürgen, Volker Riediger, and Andreas Winter (2008). “Graph Technology in Reverse Engineering—The TGraph Approach”. In: *Proc. 10th Workshop Software Reengineering. GI Lecture Notes in Informatics*. Citeseer (Cited on page 114).
- Etzkorn, Letha, Carl Davis, and Wei Li (1998). “A practical look at the lack of cohesion in methods metric”. In: *Journal of Object-Oriented Programming*. Citeseer (Cited on page 108).
- Fereday, Jennifer and Eimear Muir-Cochrane (2006). “Demonstrating rigor using thematic analysis: A hybrid approach of inductive and deductive coding and theme development”. In: *International journal of qualitative methods* 5.1, pp. 80–92 (Cited on page 125).
- Ferrari, Alan, Dario Gallucci, Daniele Puccinelli, and Silvia Giordano (2015). “Detecting energy leaks in android app with poem”. In: *2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops)*. IEEE, pp. 421–426 (Cited on page 13).
- Ferreira, Denzil, Anind K Dey, and Vassilis Kostakos (2011). “Understanding human-smartphone concerns: a study of battery life”. In: *International Conference on Pervasive Computing*. Springer, pp. 19–33 (Cited on page 2).
- Fowler, Martin (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional (Cited on page 4).
- Gamma, Erich (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India (Cited on page 4).
- Gao, Jerry, Xiaoying Bai, Wei-Tek Tsai, and Tadahiro Uehara (2014). “Mobile application testing: a tutorial”. In: *Computer* 47.2, pp. 46–55 (Cited on page 73).
- Gao, Zebao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon (2016). “Sitar: GUI test script repair”. In: *IEEE Transactions on Software Engineering* 42.2, pp. 170–186 (Cited on pages 43, 57).

- Geiger, Franz-Xaver, Ivano Malavolta, Luca Pascarella, Fabio Palomba, Dario Di Nucci, and Alberto Bacchelli (2018). “A Graph-based Dataset of Commit History of Real-World Android apps”. In: *Proceedings of the 15th International Conference on Mining Software Repositories, MSR*. ACM, New York, NY (Cited on page 49).
- Gomez, Lorenzo, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein (2013). “Reran: Timing-and touch-sensitive record and replay for Android”. In: *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, pp. 72–81 (Cited on pages 23, 42).
- Gottschalk, Marion, Jan Jelschen, and Andreas Winter (2014). “Saving Energy on Mobile Devices by Refactoring.” In: *EnviroInfo*, pp. 437–444 (Cited on pages 127, 133).
- Gottschalk, Marion, Mirco Josefiok, Jan Jelschen, and Andreas Winter (2012). “Removing Energy Code Smells with Reengineering Services.” In: *GI-Jahrestagung* 208, pp. 441–455 (Cited on pages 96, 114).
- Gousios, Georgios, Margaret-Anne Storey, and Alberto Bacchelli (2016). “Work practices and challenges in pull-based development: The contributor’s perspective”. In: *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, pp. 285–296 (Cited on pages 58, 67).
- Grier, David Alan (2017). *Is the Smartphone a Real Innovation?* <https://www.computer.org/publications/tech-news/closer-than-you-might-think/is-the-smartphone-a-real-innovation->. Accessed: 2019-04-30 (Cited on page 1).
- Gunasekaran, S and V Bargavi (2015). “Survey on automation testing tools for mobile applications”. In: *International journal of Advanced Engineering Research and Science (IJIAERS)* 2.11 (Cited on page 43).
- Hao, Shuai, Ding Li, William GJ Halfond, and Ramesh Govindan (2013). “Estimating mobile application energy consumption using program analysis”. In: *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE Press, pp. 92–101 (Cited on pages 12, 114, 119).
- Hao, Shuai, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan (2014). “Puma: Programmable UI-automation for large-scale dynamic analysis of mobile apps”. In: *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. ACM, pp. 204–217 (Cited on page 43).
- Hecht, Geoffrey, Naouel Moha, and Romain Rouvoy (2016). “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. Austin, Texas: ACM, pp. 59–69 (Cited on pages 42, 96, 120).

- Hecht, Geoffrey, Romain Rouvoy, Naouel Moha, and Laurence Duchien (2015). “Detecting antipatterns in Android apps”. In: *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems*. IEEE Press, pp. 148–149 (Cited on pages 77, 95, 114).
- Hegedűs, Péter, Dénes Bán, Rudolf Ferenc, and Tibor Gyimóthy (2012). “Myth or reality? analyzing the effect of design patterns on software maintainability”. In: *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*. Springer, pp. 138–145 (Cited on page 167).
- Hilton, Michael, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig (2016). “Usage, costs, and benefits of continuous integration in open-source projects”. In: *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, pp. 426–437 (Cited on pages 48, 51, 54, 70, 72, 75).
- Hindle, Abram (2015). “Green mining: a methodology of relating software change and configuration to power consumption”. In: *Empirical Software Engineering* 20.2, pp. 374–409 (Cited on page 23).
- Hindle, Abram, Alex Wilson, Kent Rasmussen, E Jed Barlow, Joshua Charles Campbell, and Stephen Romansky (2014). “Greenminer: A hardware based mining software repositories software energy consumption framework”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, pp. 12–21 (Cited on pages 13, 119).
- Hoque, Mohammad Ashraful, Matti Siekkinen, Kashif Nizam Khan, Yu Xiao, and Sasu Tarkoma (2016). “Modeling, profiling, and debugging the energy consumption of mobile devices”. In: *ACM Computing Surveys (CSUR)* 48.3, p. 39 (Cited on page 12).
- Hosio, Simo, Denzil Ferreira, Jorge Goncalves, Niels van Berkel, Chu Luo, Muzamil Ahmed, Huber Flores, and Vassilis Kostakos (2016). “Monetary assessment of battery life on smartphones”. In: *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 1869–1880 (Cited on page 3).
- Hu, Cuixiong and Iulian Neamtiu (2011). “Automating GUI testing for Android applications”. In: *Proceedings of the 6th International Workshop on Automation of Software Test*. ACM, pp. 77–83 (Cited on page 46).
- Imes, Connor and Henry Hoffmann (2015). “Minimizing energy under performance constraints on embedded platforms: resource allocation heuristics for homogeneous and single-ISA heterogeneous multi-cores”. In: *ACM SIGBED Review* 11.4, pp. 49–54 (Cited on page 13).
- International Organization for Standardization (2011). *Systems and software engineering: Systems and software Quality Requirements and Evaluation (SQuaRE): System and software quality models*. Standard. Geneva, Switzerland: ISO/IEC (Cited on pages 4, 146).

- Joorabchi, Mona Erfani, Ali Mesbah, and Philippe Kruchten (2013). “Real challenges in mobile app development”. In: *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*. IEEE, pp. 15–24 (Cited on pages 38, 75).
- Jung, Wonwoo, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha (2012). “DevScope: a nonintrusive and online power analysis tool for smartphone hardware components”. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, pp. 353–362 (Cited on page 12).
- Kaasila, Jouko, Denzil Ferreira, Vassilis Kostakos, and Timo Ojala (2012). “Testdroid: Automated Remote UI Testing on Android”. In: *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*. MUM ’12. Ulm, Germany: ACM, 28:1–28:4 (Cited on page 53).
- Karvonen, Teemu, Woubshet Behutiye, Markku Oivo, and Pasi Kuvaja (2017). “Systematic literature review on the impacts of agile release engineering practices”. In: *Information and Software Technology* (Cited on page 57).
- Kerby, Dave S. (2014). “The Simple Difference Formula: An Approach to Teaching Nonparametric Correlation”. In: *Comprehensive Psychology* 3, 11.IT.3.1. eprint: <https://doi.org/10.2466/11.IT.3.1> (Cited on page 63).
- Khalid, Hammad, Meiyappan Nagappan, Emad Shihab, and Ahmed E Hassan (2014). “Prioritizing the devices to test your app on: A case study of Android game apps”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 610–620 (Cited on page 46).
- Khomh, Foutse and Yann-Gael Gueheneuce (2008). “Do design patterns impact software quality positively?” In: *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*. IEEE, pp. 274–278 (Cited on page 168).
- Kim, Dongwon, Nohyun Jung, Yohan Chon, and Hojung Cha (2015). “Content-Centric Energy Management of Mobile Displays”. In: *IEEE Transactions on Mobile Computing* 15, pp. 1925–1938 (Cited on page 95).
- (2016). “Content-centric energy management of mobile displays”. In: *IEEE Transactions on Mobile Computing* 15.8, pp. 1925–1938 (Cited on pages 127, 136, 168).
- Kjærgaard, Mikkel Baun and Henrik Blunck (2011). “Unsupervised power profiling for mobile devices”. In: *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*. Springer, pp. 138–149 (Cited on page 11).
- Kochhar, Pavneet Singh, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang (2013a). “Adoption of Software Testing in Open Source Projects — A Preliminary Study on 50,000 Projects”. In: *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*. IEEE, pp. 353–356 (Cited on page 51).

- Kochhar, Pavneet Singh, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang (2013b). “An empirical study of adoption of software testing in open source projects”. In: *Quality Software (QSIC), 2013 13th International Conference on*. IEEE, pp. 103–112 (Cited on page 51).
- Kochhar, Pavneet Singh, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo (2015). “Understanding the test automation culture of app developers”. In: *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, pp. 1–10 (Cited on pages 46, 49, 58, 166).
- Kong, Pingfan, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein (2018). “Automated Testing of Android Apps: A Systematic Literature Review”. In: *IEEE Transactions on Reliability* (Cited on page 145).
- Krutz, Daniel E, Mehdi Mirakhori, Samuel A Malachowsky, Andres Ruiz, Jacob Peterson, Andrew Filipski, and Jared Smith (2015). “A dataset of open-source Android applications”. In: *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, pp. 522–525 (Cited on pages 46, 51, 75).
- Kuipers, T., I. Heitlager, and J. Visser (Sept. 2007). “A Practical Model for Measuring Maintainability”. In: *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)(QUATIC)*. Vol. 00, pp. 30–39 (Cited on page 146).
- Kulkarni, Madhuri Kishan and A Soumya (2016). “Deployment of Calabash Automation Framework to Analyze the Performance of an Android Application”. In: *Journal 4 Research* 2.03, pp. 70–75 (Cited on page 43).
- Kwon, Young-Woo and Eli Tilevich (2013). “Reducing the energy consumption of mobile applications behind the scenes”. In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, pp. 170–179 (Cited on page 168).
- (2015). “Facilitating the implementation of adaptive cloud offloading to improve the energy efficiency of mobile applications”. In: *Mobile Software Engineering and Systems (MOBILESoft), 2015 2nd ACM International Conference on*. IEEE, pp. 94–104 (Cited on page 95).
- de Langhe, Bart, Philip M. Fernbach, and Donald R. Lichtenstein (2016). “Navigating by the Stars: Investigating the Actual and Perceived Validity of Online User Ratings”. In: *Journal of Consumer Research* 42.6, pp. 817–833. eprint: /oup/backfile/content_public/journal/jcr/42/6/10.1093_jcr_ucv047/3/ucv047.pdf (Cited on page 67).
- Lee, Seokjun, Wonwoo Jung, Yohan Chon, and Hojung Cha (2015). “EnTrack: a system facility for analyzing energy consumption of Android system services”. In: *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, pp. 191–202 (Cited on page 19).
- Leech, Nancy L and Anthony J Onwuegbuzie (2002). “A Call for Greater Use of Nonparametric Statistics.” In: (Cited on page 63).

- Li, Ding and William GJ Halfond (2014). “An investigation into energy-saving programming practices for Android smartphone app development”. In: *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM, pp. 46–53 (Cited on pages 3, 5, 12, 23, 96, 114, 120, 127, 133).
- (2015). “Optimizing energy of http requests in Android applications”. In: *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*. ACM, pp. 25–28 (Cited on pages 95, 114).
- Li, Ding, Shuai Hao, Jiaping Gui, and William GJ Halfond (2014a). “An empirical study of the energy consumption of Android applications”. In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, pp. 121–130 (Cited on pages 12, 19, 38, 95, 114).
- Li, Ding, Shuai Hao, William GJ Halfond, and Ramesh Govindan (2013). “Calculating source line level energy information for Android applications”. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, pp. 78–89 (Cited on page 12).
- Li, Ding, Yingjun Lyu, Jiaping Gui, and William GJ Halfond (2016). “Automated energy optimization of http requests for mobile applications”. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE)*. ACM, pp. 249–260 (Cited on page 146).
- Li, Ding, Angelica Huyen Tran, and William GJ Halfond (2014b). “Making web applications more energy efficient for OLED smartphones”. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, pp. 527–538 (Cited on pages 12, 23, 127).
- (2015). “Nyx: A display energy optimizer for mobile web apps”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, pp. 958–961 (Cited on pages 95, 127).
- Li, Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon (2017). “Static Analysis of Android Apps: A Systematic Literature Review”. In: *Information and Software Technology* (Cited on page 146).
- Li, Xiao, Nana Chang, Yan Wang, Haohua Huang, Yu Pei, Linzhang Wang, and Xuandong Li (2017). “ATOM: Automatic Maintenance of GUI Test Scripts for Evolving Mobile Applications”. In: *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*. IEEE, pp. 161–171 (Cited on page 57).
- Linares-Vásquez, Mario (2015). “Enabling testing of Android apps”. In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*. Vol. 2. IEEE, pp. 763–765 (Cited on page 43).

- Linares-Vásquez, Mario, Gabriele Bavota, Carlos Bernal-Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk (2014). “Mining Energy-greedy API Usage Patterns in Android Apps: An Empirical Study”. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, pp. 2–11 (Cited on pages 15, 19, 23, 41, 42, 95, 102, 119, 120).
- Linares-Vásquez, Mario, Gabriele Bavota, Carlos Bernal-Cárdenas, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk (Sept. 2018). “Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps”. In: *ACM Trans. Softw. Eng. Methodol.* 27.3, 14:1–14:47 (Cited on page 146).
- Linares-Vásquez, Mario, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk (2015). “Optimizing energy consumption of GUIs in Android apps: a multi-objective approach”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, pp. 143–154 (Cited on page 95).
- Linares-Vásquez, Mario, Carlos Bernal-Cárdenas, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk (2017a). “GEMMA: multi-objective optimization of energy consumption of GUIs in Android apps”. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 11–14 (Cited on pages 114, 115, 127, 168).
- Linares-Vásquez, Mario, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk (2017b). “How do Developers Test Android Applications?” In: *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME’17)*, page to appear (Cited on pages 43, 50).
- Linares-Vásquez, Mario, Kevin Moran, and Denys Poshyvanyk (2017c). “Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing”. In: *33rd IEEE International Conference on Software Maintenance and Evolution (ICSME’17)*, page to appear (Cited on pages 43, 45, 50, 53, 58).
- Liu, Chien-Hung, Chien-Yu Lu, Shan-Jen Cheng, Koan-Yuh Chang, Yung-Chia Hsiao, and Weng-Ming Chu (2014). “Capture-replay testing for Android applications”. In: *Computer, Consumer and Control (IS3C), 2014 International Symposium on*. IEEE, pp. 1129–1132 (Cited on page 43).
- Liu, Kuei-Chun, Yu-Yu Lai, and Ching-Hong Wu (2017). “A Mechanism of Reliable and Standalone Script Generator on Android”. In: *Software Testing, Verification and Validation Workshops (ICSTW), 2017 IEEE International Conference on*. IEEE, pp. 372–374 (Cited on page 43).
- Liu, Yepang, Chang Xu, and Shing-Chi Cheung (2013). “Where has my battery gone? Finding sensor related energy black holes in smartphone applications”. In: *Pervasive Computing and Communications (PerCom), 2013 IEEE International Conference on*. IEEE, pp. 2–10 (Cited on page 119).

- Liu, Yepang, Chang Xu, Shing-Chi Cheung, and Valerio Terragni (2016). “Understanding and detecting wake lock misuses for android applications”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. ACM, pp. 396–409 (Cited on pages 127, 129).
- Machiry, Aravind, Rohan Tahiliani, and Mayur Naik (2013). “Dynodroid: An input generation system for Android apps”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, pp. 224–234 (Cited on page 43).
- Mahmood, Riyadh, Nariman Mirzaei, and Sam Malek (2014). “Evodroid: Segmented evolutionary testing of Android apps”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 599–609 (Cited on page 43).
- Maji, Amiya Kumar, Kangli Hao, Salmin Sultana, and Saurabh Bagchi (2010). “Characterizing failures in mobile OSes: A case study with Android and Symbian”. In: *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, pp. 249–258 (Cited on page 46).
- Malavolta, Ivano, Giuseppe Procaccianti, Paul Noorland, and Petar Vukmirović (2017). “Assessing the impact of service workers on the energy efficiency of progressive web apps”. In: *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*. IEEE Press, pp. 35–45 (Cited on pages 12, 42, 114).
- Malavolta, Ivano, Roberto Verdecchia, Bojan Filipovic, Magiel Bruntink, and Patricia Lago (2018). “How Maintainability Issues of Android Apps Evolve”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 334–344 (Cited on page 167).
- Mannan, Umme Ayda, Iftekhar Ahmed, Rana Abdullah M Almurshed, Danny Dig, and Carlos Jensen (2016). “Understanding code smells in Android applications”. In: *Proceedings of the International Workshop on Mobile Software Engineering and Systems*. ACM, pp. 225–234 (Cited on page 95).
- Manotas, Irene, Christian Bird, Rui Zhang, David Shepherd, Ciera Jaspan, Caitlin Sadowski, Lori Pollock, and James Clause (2016). “An empirical study of practitioners’ perspectives on green software engineering”. In: *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, pp. 237–248 (Cited on pages 3, 139).
- Martin, William, Federica Sarro, Yue Jia, Yuanyuan Zhang, and Mark Harman (2017). “A survey of app store analysis for software engineering”. In: *IEEE transactions on software engineering* 43.9, pp. 817–847 (Cited on pages 49, 55, 75, 126).
- McGraw, Kenneth O and SP Wong (1992). “A common language effect size statistic.” In: *Psychological bulletin* 111.2, p. 361 (Cited on pages 63, 157).

- McIntosh, Andrea, Safwat Hassan, and Abram Hindle (2018). “What can Android mobile app developers do about the energy consumption of machine learning?” In: *Empirical Software Engineering*, pp. 1–40 (Cited on page 120).
- Metri, Grace, Abhishek Agrawal, Ramesh Peri, and Weisong Shi (2012). “What is eating up battery life on my SmartPhone: A case study”. In: *Energy Aware Computing, 2012 International Conference on*. IEEE, pp. 1–6 (Cited on pages 127, 132).
- Morales, Rodrigo, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol (2018). “Earmo: an energy-aware refactoring approach for mobile apps”. In: *IEEE Transactions on Software Engineering* 44.12, pp. 1176–1206 (Cited on page 169).
- Moran, Kevin, Mario Linares-Vásquez, and Denys Poshyvanyk (2017). “Automated GUI testing of Android apps: from research to practice”. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 505–506 (Cited on page 46).
- Moreira, Rodrigo MLM, Ana CR Paiva, and Atif Memon (2013). “A pattern-based approach for GUI modeling and testing”. In: *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, pp. 288–297 (Cited on page 19).
- Morgado, Ines Coimbra and Ana CR Paiva (2015). “The iMPAcT tool: testing UI patterns on mobile applications”. In: *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, pp. 876–881 (Cited on page 19).
- Moura, Irineu, Gustavo Pinto, Felipe Ebert, and Fernando Castor (2015). “Mining energy-aware commits”. In: *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, pp. 56–67 (Cited on pages 119, 125, 150–152, 167, 168).
- Muccini, Henry, Antonio Di Francesco, and Patrizio Esposito (2012). “Software testing of mobile applications: Challenges and future research directions”. In: *7th International Workshop on Automation of Software Test, AST 2012, Zurich, Switzerland, June 2-3, 2012*, pp. 29–35 (Cited on pages 5, 46, 58, 164).
- Mundody, Sona and K Sudarshan (2014). “Evaluating the Impact of Android Best Practices on Energy Consumption”. In: *IJCA Proceedings on International Conference on Information and Communication Technologies*. Vol. 8, pp. 1–4 (Cited on page 96).
- Nagappan, Meiyappan and Emad Shihab (2016). “Future trends in software engineering research for mobile apps”. In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 5. IEEE, pp. 21–32 (Cited on page 75).

- Nayebi, Maleknaz, Bram Adams, and Guenther Ruhe (2016). “Release Practices for Mobile Apps—What do Users and Developers Think?” In: *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*. Vol. 1. IEEE, pp. 552–562 (Cited on page 46).
- Nayebi, Maleknaz, Henry Cho, and Guenther Ruhe (2018). “App store mining is not enough for app improvement”. In: *Empirical Software Engineering*, pp. 1–31 (Cited on page 62).
- Negara, Stas, Mihai Codoban, Danny Dig, and Ralph E Johnson (2014). “Mining fine-grained code changes to detect unknown change patterns”. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM, pp. 803–813 (Cited on page 117).
- internationale de normalisation, Organisation (2011). *Systems and Software Engineering: Systems and Software Quality Requirements and Evaluation (SQuaRE): System and Software Quality Models*. ISO/IEC (Cited on page 155).
- Olivari, M.A. (2018). “Maintainable Production: Mapping Software Quality Change to Source Code Contributions”. Master’s thesis. University of Amsterdam (Cited on pages 146, 155).
- Oliveira, Wellington, Renato Oliveira, and Fernando Castor (2017). “A study on the energy consumption of Android app development approaches”. In: *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*. IEEE, pp. 42–52 (Cited on pages 120, 163).
- Palattella, Maria Rita, Mischa Dohler, Alfredo Grieco, Gianluca Rizzo, Johan Torsner, Thomas Engel, and Latif Ladid (2016). “Internet of things in the 5G era: Enablers, architecture, and business models”. In: *IEEE Journal on Selected Areas in Communications* 34.3, pp. 510–527 (Cited on page 174).
- Palomba, Fabio, Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia (2017). “Lightweight detection of Android-specific code smells: The aDoctor project”. In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 487–491 (Cited on pages 77, 114, 169).
- (2019). “On the impact of code smells on the energy consumption of mobile applications”. In: *Information and Software Technology* 105, pp. 43–55 (Cited on pages 142, 168).
- Palomba, Fabio, Damian A Tamburri, Alexander Serebrenik, Andy Zaidman, Francesca Arcelli Fontana, and Rocco Oliveto (2018). “How do community smells influence code smells?” In: *Proceedings of the 40th International Conference on Software Engineering Companion*. ACM, pp. 240–241 (Cited on page 117).
- Pang, Candy, Abram Hindle, Bram Adams, and Ahmed E Hassan (2015). “What do programmers know about the energy consumption of software?” In: *PeerJ PrePrints* 3, e886v1 (Cited on pages 145, 163).

Pang, Candy, Abram Hindle, Bram Adams, and Ahmed E. Hassan (May 2016). “What Do Programmers Know About Software Energy Consumption?” In: *IEEE Software* 33.3, pp. 83–89 (Cited on pages 100, 113).

Pascarella, Luca, Franz-Xaver Geiger, Fabio Palomba, Dario Di Nucci, Ivano Malavolta, and Alberto Bacchelli (2018). “Self-Reported Activities of Android Developers”. In: *5th IEEE/ACM International Conference on Mobile Software Engineering and Systems, New York, NY* (Cited on pages 49, 142, 167).

Pathak, Abhinav, Y Charlie Hu, and Ming Zhang (2011a). “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*. ACM, p. 5 (Cited on page 119).

- (2012a). “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*. ACM, pp. 29–42 (Cited on pages 11, 12, 95, 113, 119, 120).

Pathak, Abhinav, Y Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min Wang (2011b). “Fine-grained power modeling for smartphones using system call tracing”. In: *Proceedings of the sixth conference on Computer systems*. ACM, pp. 153–168 (Cited on pages 9, 11, 95, 113, 119).

Pathak, Abhinav, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff (2012b). “What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps”. In: *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, pp. 267–280 (Cited on pages 127, 129).

Pereira, Rui Alexandre Afonso (2018). “Energyware engineering: techniques and tools for green software development”. PhD thesis. Universidade do Minho (Cited on page 4).

Pereira, Rui, Tiago Carção, Marco Couto, Jácome Cunha, João Paulo Fernandes, and João Saraiva (2017a). “Helping programmers improve the energy efficiency of source code”. In: *Proceedings of the 39th International Conference on Software Engineering Companion*. IEEE Press, pp. 238–240 (Cited on page 114).

Pereira, Rui, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva (2017b). “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*. ACM, pp. 256–267 (Cited on page 163).

Pham, Raphael, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider (2013). “Creating a shared understanding of testing culture on a social coding site”. In: *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*. IEEE Press, pp. 112–121 (Cited on page 117).

- Pham, Raphael, Yauheni Stolar, and Kurt Schneider (2015). "Automatically recommending test code examples to inexperienced developers". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (FSE)*. ACM, pp. 890–893 (Cited on page 117).
- Picco, Gian Pietro, Christine Julien, Amy L Murphy, Mirco Musolesi, and Gruia-Catalin Roman (2014). "Software engineering for mobility: reflecting on the past, peering into the future". In: *Proceedings of the on Future of Software Engineering*. ACM, pp. 13–28 (Cited on page 46).
- Pinto, Gustavo, Fernando Castor, and Yu David Liu (2014). "Mining questions about software energy consumption". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, pp. 22–31 (Cited on pages 77, 120).
- Qian, Hao and Daniel Andresen (2015). "Reducing mobile device energy consumption with computation offloading". In: *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD), 2015 16th IEEE/ACIS International Conference on*. IEEE, pp. 1–8 (Cited on page 95).
- Rasmussen, Kent, Alex Wilson, and Abram Hindle (2014). "Green mining: energy consumption of advertisement blocking methods". In: *Proceedings of the 3rd International Workshop on Green and Sustainable Software*. ACM, pp. 38–45 (Cited on pages 13, 19).
- Ray, Baishakhi, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu (2014). "A large scale study of programming languages and code quality in github". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 155–165 (Cited on page 168).
- Reimann, Jan and Uwe Aßmann (2013). "Quality-Aware Refactoring For Early Detection And Resolution Of Energy Deficiencies". In: *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE Computer Society, pp. 321–326 (Cited on pages 114, 174).
- Reimann, Jan, Martin Brylski, and Uwe Aßmann (2014). "A tool-supported quality smell catalogue for Android developers". In: *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung—MMSM*. Vol. 2014 (Cited on pages 114, 120, 127, 169, 174).
- Robillard, Martin P and Nenad Medvidovic (2016). "Disseminating Architectural Knowledge on Open-Source Projects: A Case Study of the Book" Architecture of Open-Source Applications"". In: *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, pp. 476–487 (Cited on page 3).
- Romansky, Stephen, Neil C Borle, Shaiful Chowdhury, Abram Hindle, and Russ Greiner (2017). "Deep Green: modelling time-series of software energy consumption". In: *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, pp. 273–283 (Cited on page 119).

- Rua, Rui, Marco Couto, and João Saraiva (2019). “GreenSource: a large-scale collection of Android code, tests and energy metrics”. In: *Proceedings of the 16th International Conference on Mining Software Repositories* (Cited on page 11).
- Ruiz, Israel Mojica, Meiyappan Nagappan, Bram Adams, Thorsten Berger, Steffen Dienst, and Ahmed Hassan (2017). “An examination of the current rating system used in mobile app stores”. In: *IEEE Software* (Cited on pages 67, 74).
- Saborido, Rubén, Venera Venera Arnaoudova, Giovanni Beltrame, Foutse Khomh, and Giuliano Antoniol (2015). *On the impact of sampling frequency on software energy measurements*. Tech. rep. PeerJ PrePrints (Cited on page 24).
- Sahin, Cagri, Furkan Cayci, Irene Lizeth Manotas Gutiérrez, James Clause, Fouad Kiamilev, Lori Pollock, and Kristina Winbladh (2012). “Initial explorations on design pattern energy usage”. In: *Green and Sustainable Software (GREENS), 2012 First International Workshop on*. IEEE, pp. 55–61 (Cited on page 120).
- Sahin, Cagri, Lori Pollock, and James Clause (2014). “How do code refactorings affect energy usage?” In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, p. 36 (Cited on pages 15, 96, 100, 113, 114, 145).
- (2016). “From benchmarks to real apps: Exploring the energy impacts of performance-directed changes”. In: *Journal of Systems and Software* 117, pp. 307–316 (Cited on pages 13, 15, 42, 93, 96, 114, 119, 120).
- Sampson, Adrian, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman (2011). “EnerJ: Approximate data types for safe and general low-power computation”. In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM, pp. 164–174 (Cited on page 165).
- Schulman, Aaron, Thomas Schmid, Prabal Dutta, and Neil Spring (2011). “Phone power monitoring with battor”. In: *17th ACM International Conference on Mobile Computing and Networking (MobiCom 2011)* (Cited on page 13).
- Segata, Michele, Bastian Bloessl, Christoph Sommer, and Falko Dressler (2014). “Towards energy efficient smart phone applications: Energy models for offloading tasks into the cloud”. In: *2014 IEEE International Conference on Communications (ICC)*. IEEE, pp. 2394–2399 (Cited on page 13).
- Shafer, Ilari and Mark L Chang (2010). “Movement detection for power-efficient smartphone WLAN localization”. In: *Proceedings of the 13th ACM international conference on Modeling, analysis, and simulation of wireless and mobile systems*. ACM, pp. 81–90 (Cited on page 127).
- Shin, Donghwa, Kitae Kim, Naehyuck Chang, Woojoo Lee, Yanzhi Wang, Qing Xie, and Massoud Pedram (2013). “Online estimation of the remaining energy capacity in mobile systems considering system-wide power consumption and battery characteristics”. In: *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, pp. 59–64 (Cited on page 13).

- Silva, Davi Bernardo, Andre Takeshi Endo, Marcelo Medeiros Eler, and Vinicius HS Durelli (2016). “An analysis of automated tests for mobile Android applications”. In: *Computing Conference (CLEI), 2016 XLII Latin American*. IEEE, pp. 1–9 (Cited on pages 49, 50).
- Sousa, Leonardo et al. (2018). “Identifying design problems in the source code: a grounded theory”. In: *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. ACM, pp. 921–931 (Cited on page 139).
- Stolberg, Sean (2009). “Enabling agile testing through continuous integration”. In: *Agile Conference, 2009. AGILE'09*. IEEE, pp. 369–374 (Cited on page 57).
- Tonini, Aline Rodrigues, Leonardo Matthis Fischer, Julio Carlos Balzano de Mattos, and Lisane B de Brisolara (2013). “Analysis and Evaluation of the Android Best Practices Impact on the Efficiency of Mobile Applications.” In: *SBESC*, pp. 157–158 (Cited on page 96).
- Van Deursen, Arie (2001). “Program comprehension risks and opportunities in extreme programming”. In: *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, pp. 176–185 (Cited on page 67).
- Vekris, Panagiotis, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal (2012). “Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs.” In: *HotPower* (Cited on page 119).
- Visser, Joost, Sylvan Rigal, Rob van der Leek, Pascal van der Eck, and Gijs Wijnholds (2016). *Building Maintainable Software, Java Edition: Ten Guidelines for Future-Proof Code.* " O'Reilly Media, Inc." (Cited on pages 46, 146, 149, 153, 161–163, 166).
- Vlissides, John, Richard Helm, Ralph Johnson, and Erich Gamma (1995). “Design Patterns: Elements of reusable object-oriented software”. In: *Addison-Wesley* (Cited on page 117).
- Wan, Mian, Yuchen Jin, Ding Li, Jiaping Gui, Sonal Mahajan, and William GJ Halfond (2017). “Detecting display energy hotspots in Android apps”. In: *Software Testing, Verification and Reliability* 27.6, e1635 (Cited on page 12).
- Wang, Yong and Yazan Alshboul (2015). “Mobile security testing approaches and challenges”. In: *Mobile and Secure Services (MOBISECSERV), 2015 First Conference on*. IEEE, pp. 1–5 (Cited on pages 46, 58).
- White, Jules, Siobhan Clarke, Christin Groba, Brian Dougherty, Chris Thompson, and Douglas C Schmidt (2010). “R&D challenges and solutions for mobile cyber-physical applications and supporting Internet services”. In: *Journal of internet services and applications* 1.1, pp. 45–56 (Cited on page 174).
- Wilke, Claas, Sebastian Götz, and Sebastian Richly (2013a). “JouleUnit: a generic framework for software energy profiling and testing”. In: *Proceedings of the 2013 workshop on Green in/by software engineering*. ACM, pp. 9–14 (Cited on page 119).

- Wilke, Claas, Sebastian Richly, Sebastian Gotz, Christian Piechnick, and Uwe Aßmann (2013b). “Energy consumption and efficiency in mobile applications: A user feedback study”. In: *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCoM), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*. IEEE, pp. 134–141 (Cited on page 100).
- Xu, Fengyuan, Yunxin Liu, Qun Li, and Yongguang Zhang (2013). “V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics”. In: *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pp. 43–55 (Cited on page 12).
- Yıldırım, Kasım Sinan, Amjad Yousef Majid, Dimitris Patoukas, Koen Schaper, Przemysław Pawelczak, and Josiah Hester (2018). “Ink: Reactive kernel for tiny battery-less sensors”. In: *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*. ACM, pp. 41–53 (Cited on page 165).
- Yoon, Chanmin, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha (2012). “AppScope: Application Energy Metering Framework for Android Smartphone Using Kernel Activity Monitoring.” In: *USENIX Annual Technical Conference*. Vol. 12, pp. 1–14 (Cited on page 12).
- Yskout, Koen, Riccardo Scandariato, and Wouter Joosen (2015). “Do security patterns really help designers?” In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*. Vol. 1. IEEE, pp. 292–302 (Cited on page 117).
- Yu, Kisoo, Donghee Han, Changhwan Youn, Seungkon Hwang, and Jaechul Lee (2013). “Power-aware task scheduling for big. LITTLE mobile processor”. In: *SoC Design Conference (ISOCC), 2013 International*. IEEE, pp. 208–212 (Cited on page 9).
- Zaeem, Razieh Nokhbeh, Mukul R Prasad, and Sarfraz Khurshid (2014). “Automated generation of oracles for testing user-interaction features of mobile apps”. In: *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, pp. 183–192 (Cited on page 46).
- Zeng, Xia, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie (2016). “Automated test input generation for Android: Are we really there yet in an industrial case?” In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, pp. 987–992 (Cited on pages 50, 51).
- Zhang, Lide, Mark S Gordon, Robert P Dick, Z Morley Mao, Peter Dinda, and Lei Yang (2012). “Adel: An automatic detector of energy leaks for smartphone applications”. In: *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, pp. 363–372 (Cited on page 11).

Zhang, Lide, Birjodh Tiwana, Zhiyun Qian, Zhaoguang Wang, Robert P Dick, Zhuoqing Morley Mao, and Lei Yang (2010). “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*. ACM, pp. 105–114 (Cited on pages 11, 12, 113, 119).

Zhao, Yangyang, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu (2017). “The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study”. In: *32nd IEEE/ACM International Conference on Automated Software Engineering* (Cited on pages 48, 51, 70, 72, 166).

Zhou, Bowen, Amir Vahid Dastjerdi, Rodrigo N Calheiros, Satish Narayana Srirama, and Rajkumar Buyya (2015). “A context sensitive offloading scheme for mobile cloud computing service”. In: *2015 IEEE 8th International Conference on Cloud Computing*. IEEE, pp. 869–876 (Cited on page 11).

– (2017). “mCloud: A context-aware offloading framework for heterogeneous mobile cloud”. In: *IEEE Transactions on Services Computing* 10.5, pp. 797–810 (Cited on page 11).

Zhu, Haitao Steve, Chaoren Lin, and Yu David Liu (2015). “A programming model for sustainable software”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE, pp. 767–777 (Cited on page 165).

