

Supporting Evolution and Maintenance of Android Apps

Mario Linares Vásquez

Bogotá, Colombia

Bachelor of Engineering, Universidad Nacional de Colombia, 2005
Master of Science, Universidad Nacional de Colombia, 2009

A Dissertation presented to the Graduate Faculty
of the College of William and Mary in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary
August 2016

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

Mario Linares Vásquez

Approved by the Committee, May 2016

Committee Chair

Associate Professor Denys Poshyvanyk, Computer Science
The College of William and Mary

Associate Professor Peter Kemper, Computer Science
The College of William and Mary

Professor Andreas Stathopoulos, Computer Science
The College of William and Mary

Assistant Professor Xu Liu, Computer Science
The College of William and Mary

Associate Professor Massimiliano Di Penta, Computer Science
University of Sannio, Italy

ABSTRACT

Mobile developers and testers face a number of emerging challenges. These include rapid platform evolution and API instability; issues in bug reporting and reproduction involving complex multitouch gestures; platform fragmentation; the impact of reviews and ratings on the success of their apps; management of crowd-sourced requirements; continuous pressure from the market for frequent releases; lack of effective and usable testing tools; and limited computational resources for handheld devices. Traditional and contemporary methods in software evolution and maintenance were not designed for these types of challenges; therefore, a set of studies and a new toolbox of techniques for mobile development are required to analyze current challenges and propose new solutions.

This dissertation presents a set of empirical studies, as well as solutions for some of the key challenges when **evolving and maintaining Android apps**. In particular, we analyzed key challenges experienced by practitioners and open issues in the mobile development community such as (i) Android API instability, (ii) performance optimizations, (iii) automatic GUI testing, and (iv) energy consumption. When carrying out the studies, we relied on qualitative and quantitative analyses to understand the phenomena on a large scale by considering evidence extracted from software repositories and the opinions of open-source mobile developers.

From the empirical studies, we identified that dynamic analysis is a relevant method for several evolution and maintenance tasks, in particular, because of the need of practitioners to execute/validate the apps on a diverse set of platforms (*i.e.*, device and OS) and under pressure for continuous delivery. Therefore, we designed and implemented an extensible infrastructure that enables large-scale automatic execution of Android apps to support different evolution and maintenance tasks (*e.g.*, testing and energy optimization). In addition to the infrastructure we present a taxonomy of issues, single solutions to the issues, and guidelines to enable large execution of Android apps.

Finally, we devised novel approaches aimed at supporting testing and energy optimization of mobile apps (two key challenges in evolution and maintenance of Android apps). First, we propose a novel hybrid approach for automatic GUI-based testing of apps that is able to generate (un)natural test sequences by mining real applications usages and learning statistical models that represent the GUI interactions. In addition, we propose a multi-objective approach for optimizing the energy consumption of GUIs in Android apps that is able to generate visually appealing color compositions, while reducing the energy consumption and keeping a design concept close to the original.

TABLE OF CONTENTS

Acknowledgments	vii
List of Tables	viii
List of Figures	x
1 Introduction	2
2 Empirical Studies	9
2.1 Android API instability [163, 69]	10
2.1.1 RQ ₁ : Does the fault-proneness of APIs affect the user ratings of Android Apps?	12
2.1.2 RQ ₂ : Does the change-proneness of APIs affect the user ratings of Android Apps?	13
2.1.3 RQ ₃ : To what extent Android developers experience problems when using APIs?	15
2.1.4 RQ ₄ : To what extent Android developers consider problematic APIs to be the cause of negative user rating/comments?	16
2.2 Usage of Crowdsourced Requirements [201]	17
2.2.1 RQ ₁ :To what extent do developers fulfill reviews when working on a new app release?	19
2.2.2 RQ ₂ :What is the effect of a crowd review mechanism (for planning and implementing future changes) on the app success?	19
2.3 How Android Developers Detect and Fix Performance Bottlenecks [161]	20

2.3.1	RQ ₁ : What practices are used by Android developers to detect performance bottlenecks?	21
2.3.2	RQ ₂ : What tools are used by Android developers to detect performance bottlenecks?	21
2.3.3	RQ ₃ : What practices are used by Android developers to fix performance bottlenecks?	22
2.4	How Developers Micro-Optimize Android Apps	22
2.4.1	RQ ₁ : What is the distribution of micro-optimization opportunities across Android Apps at GitHub?	24
2.4.2	RQ ₂ : To what extent are micro-optimizations introduced or avoided during the evolution and maintenance of Android apps?	25
2.4.3	RQ ₃ : How do micro-optimizations impact CPU and memory usage of Android apps?	26
2.4.4	RQ ₄ : What practices are used by developers for detecting optimization opportunities in Android apps?	27
2.4.5	RQ ₅ : What micro-optimizations are recognized (and used) by developers as useful for improving the performance of apps?	28
2.5	Energy Greedy APIs [165]	28
2.5.1	RQ ₁ : Which are the most energy-greedy Android API methods?	30
2.5.2	RQ ₂ : Which sequences of Android API calls are the most energy-greedy?	31
2.6	How Developers Document and Design Test Cases for Android Apps	33
2.6.1	RQ ₁ : What are the strategies used by mobile developers to design test cases?	33
2.6.2	RQ ₂ : What are the preferences of mobile developers for test cases generated automatically?	34

2.6.3	RQ ₃ : What tools are used by mobile developers for automated testing?	34
2.6.4	RQ ₄ : Do mobile developers consider code coverage as a useful metric for evaluating test cases effectiveness?	35
2.7	Discussion	35
2.8	Bibliographical Notes	36
3	Enabling Large-Scale Execution and Testing of Android Apps	38
3.1	A Taxonomy of Essential and Accidental Issues	41
3.1.1	Essential Challenges for Automated GUI-based Execution . . .	42
3.1.1.1	Test Oracles	42
3.1.1.2	Event Coordination	43
3.1.1.3	External Features	46
3.1.1.4	External Dependencies	47
3.1.2	Accidental Challenges for Automated GUI-based Execution . .	47
3.1.2.1	Application Data and Cold Starts	47
3.1.2.2	Bugs in the Framework Utilities	48
3.1.2.3	Bugs and Limitations in the SDK Tools	50
3.1.2.4	Challenges Scaling Concurrent Virtual Devices	51
3.2	Proposed Solutions	53
3.2.1	Non-Image Based Generation of Oracles	54
3.2.2	Avoiding Hard-Coded Inter-Delay Times	55
3.2.3	Cleaning Data for Cold-Starts	57
3.2.4	Extraction of GUI Hierarchies for Dynamic Screens	58
3.2.5	Running Concurrent ADB Connections	59
3.2.6	Enabling Parallel Execution of AVDs	59
3.3	An Infrastructure for Large Scale Execution and Testing	61

3.4	Discussion	67
3.5	Bibliographical Notes	68
4	Combining Usage and GUI Models to Support Automatic Test Cases Generation	69
4.1	Background and Related Work	72
4.2	Mining and Generating Actionable Execution Scenarios with MONKEYLAB	74
4.2.1	<i>Record</i> : Collecting Event Logs from the Crowd	76
4.2.2	<i>Mine</i> : Extracting Event Sequences from Logs and the App	77
4.2.2.1	Mining GUI events statically from APKs (APK Analyzer)	78
4.2.2.2	Mining GUI events from event logs (Data Collector)	79
4.2.3	<i>Generate</i> : Event Sequences with Language Models	80
4.2.3.1	Language Models	81
4.2.3.2	Language Model Flavors	82
4.2.3.3	Generating Event Sequences	83
4.2.4	<i>Validate</i> : Filtering Actionable Scenarios	84
4.3	Empirical Study Design	87
4.3.1	Data Collection	87
4.3.2	Design Space	89
4.4	Results	89
4.4.1	RQ₁ : Language Models and Flavors	90
4.4.2	RQ₂ : MONKEYLAB vs. Android UI <code>monkey</code>	92
4.4.3	RQ₃ : MONKEYLAB vs. DFS	93
4.4.4	RQ₄ : MONKEYLAB vs. Manual execution	94
4.4.5	Limitations	95

4.5	Discussion	97
4.6	Bibliographics Notes	98
5	Visual Aesthetics Matter: Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps	100
5.1	Optimizing Energy Consumption of GUIs with GEMMA	104
5.1.1	Estimating the Power Consumption	105
5.1.2	Extracting Color Composition from GUIs	106
5.1.3	Multi-objective Optimization Model	112
5.2	GEMMA's Architecture	118
5.2.1	The GEMMA Web Client	119
5.2.2	The GEMMA Collector	120
5.2.3	The GEMMA Execution Engine	121
5.2.4	GEMMA in Action	123
5.3	Empirical Study Design	125
5.3.1	Research Questions	126
5.3.2	Choice of Multi-Objective Optimization Techniques for GEMMA's Evaluation	128
5.3.3	Context Selection	130
5.3.4	Data Collection	132
5.3.5	Data Analysis	137
5.4	Study Results	139
5.4.1	RQ₀ : Which multi-objective optimization approach is suitable for GEMMA?	140
5.4.2	RQ₁ : To what extent is GEMMA able to optimize the GUI energy consumption, contrast, and design objectives?	143

5.4.3	RQ₂ : Are the color compositions generated by GEMMA visually attractive as perceived by Android users?	147
5.4.4	RQ₄ : Would actual developers of mobile applications consider changing colors in an app as recommended by GEMMA ?	152
5.5	Threats to Validity	157
5.6	Related Work	158
5.6.1	Improving Energy Consumption of GUIs	158
5.6.2	Detecting Energy Bugs in Mobile Apps	159
5.7	Discussion	160
5.8	Bibliographical Notes	161
6	Conclusion	163
	Bibliography	195

ACKNOWLEDGMENTS

This dissertation is the result of four years of research at the College of William and Mary. During that time, I had the honor to work with and to be mentored by amazing people from different countries, in particular the USA and Italy. I would like to thank them for all their support, help, and guidance. In particular, I would like to thank Denys Poshyvanyk, who was not only my advisor but also my coach who helped me to improve and empower several academic and personal aspects. My thanks go to Gabriele Bavota, a research collaborator and a friend, who has been a role model to follow because of his excellence as a researcher, a professor, and a human being; Massimiliano Di Penta and Rocco Oliveto, research collaborators, who provided me with outstanding advice and helped me to improve my skills as a researcher; my American brother Christopher Vendome, and the Android SEMERU team (Carlos Bernal-Cárdenas and Kevin Moran), who are unconditional friends and collaborators; Bogdan Dit and Collin McMillan, former members of the SEMERU group, who helped and supported me during my initial steps at William and Mary. Finally, I am grateful to my family and Clau, who have been my emotional and spiritual pillars.

LIST OF TABLES

3.1	Overview of the Essential and Accidental Challenges Discussed (Relevant section numbers are indicated next to each challenge or solution, as well as corresponding links to detailed solutions in our online appendix [20] are given as citations. Text in blue represent partial solutions, green fully or mostly solved solutions, and red challenges that are currently unsolved.)	44
3.2	Proposed approaches for parallel and large-scale execution of Android apps. The columns are as follows: 1) Purpose of the tool [S ecurity E valuation, T esting C loud]; 2) Device Type [P hysical D evice, E mulator, V irtual M achine]; 3) Tool used for injecting input events [M anual, R obotium, M onkey R unner, U iAutomator]; 4) Type of exploration strategy [M anual, S ystematic, M ultiple, R uled-based, R andom]; 5) Requires Instrumentation [Y es, N o] ; 6) Type of Analysis [S tatic A nalysis, D ynamic A nalysis] ;7) Number of instances or clients executed in the experimentation phase	66
4.1	Android Apps Used in our Study. The stats include the number of activities, methods, and GUI components. Last two columns list the number of raw events (#RE) in the event logs (i.e., lines in the files collected with the getevent command) and GUI level events mined from the raw logs (#GE)	89
4.2	Accumulated Statement Coverage of the LMs	90
5.1	Android apps considered in our study.	129
5.2	Hypervolume: Wilcoxon test (adjusted <i>p</i> -values) and Cliff’s delta (<i>d</i>).	139

5.3	ECF: Wilcoxon test (adjusted p -values) and Cliff's delta (d).	142
5.4	CF: Wilcoxon test (adjusted p -values) and Cliff's delta (d).	142
5.5	DF: Wilcoxon test (adjusted p -values) and Cliff's delta (d).	142
5.6	ECF and CF: Wilcoxon test (adjusted p -values) and Cliff's delta (d). . .	144
5.7	RQ ₂ : Wilcoxon test (p -value) and Cliff's delta (d).	149
5.8	RQ ₃ : Wilcoxon test (p -value) and Cliff's delta (d) for pairwise comparisons of energy measurements (current in mA): original design vs lowest ECF	152
5.9	Battery life (in hours) when using the original version of a mobile application, and app modified with the GEMMA solution with the lowest energy consumption (lowest ECF). The column <i>Diff.</i> lists the relative difference in percentage, i.e., $\frac{\text{lowestECF}-\text{original}}{\text{original}}$	152
5.10	Energy Optimization of Android Apps	159

LIST OF FIGURES

3.1	Examples of GUI components in Android apps that are problematic for automated test cases.	45
3.2	Example of splash/welcome screens, dialogs, and semi-transparent overlays that appear in Android apps during cold-starts or the first time a feature is used.	49
3.3	A proposed architecture for enabling large-scale execution and testing of Android apps.	63
4.1	MonkeyLab architecture and the <i>Record</i> \rightarrow <i>Mine</i> \rightarrow <i>Generate</i> \rightarrow <i>Validate</i> framework	77
4.2	Accumulated Coverage for GnuCash	90
4.3	Accumulated Coverage for CarReport	91
4.4	Accumulated Coverage for Tasks	91
4.5	Accumulated Coverage for MyExpenses	92
4.6	Accumulated Coverage for Mileage	92
4.7	Total number of events executed by Strategy A that are not executed by Strategy B. The key color goes from white (zero) to red (highest value)	93
4.8	Total number of source code methods in which coverage is higher when comparing coverage of Strategy A versus Strategy B	94

5.1	Original design <i>vs</i> GEMMA’s solutions for the Learn Music Notes app. Each solution shows the color composition for three GUIs in the app.	101
5.2	The GEMMA approach and components	104
5.3	Current (mA) consumption models for primary colors of the Galaxy S4 SUPER AMOLED screen in standard RGB color space.	107
5.4	Current (mA) consumption models for primary colors of the Galaxy S4 SUPER AMOLED screen in linear RGB color space.	107
5.5	Example of palettes (20 colors each) with equidistant harmony (top), equidistant harmony with random saturation and brightness (middle), and monochromatic scale (bottom).	115
5.6	Architecture of the GEMMA Web Client and the Execution Engine. .	119
5.7	List of user requests (<i>i.e.</i> , GEMMA tasks) in the GEMMA web client.	122
5.8	Dashboard of GEMMA’s solutions for the <i>Privacy Friendly Dicer</i> app.	124
5.9	Pareto Front (top) and gauge style visualizations (bottom) in GEMMA .	125
5.10	Screenshots of the GEMMA Collector Android app when running on a Nexus 7 tablet and collecting GUIs information for the Game Master Dice app. The screenshots in this Figure are: initial activity (left), collection view (center), pre-submission activity (right).	126
5.11	Hypervolume for the three algorithms.	140
5.12	Boxplots of ECF, CF, and DF for different solutions generated by the three algorithms.	141
5.13	Original design <i>vs</i> GEMMA’s solutions for the Tasks app.	145
5.14	Original design <i>vs</i> GEMMA’s solutions for the Play Music app. . . .	145
5.15	Original design <i>vs</i> GEMMA’s solutions for the Keep app.	146

5.16 RQ ₂ : Boxplots of answers provided by participants. OD=Original Design, LE=Lowest ECF, ME=Median ECF.	149
5.17 Current (mA) drawn by the analyzed apps. Each plot depicts the range (<i>i.e.</i> , min to max) of the average current for each step across the 30 executions for (i) original app (in red), and (ii) modified version with GEMMA GUI composition for low energy-consumption (in cyan). The circles depict the averages in each range.	150
5.18 Percentage of battery consumed by the analyzed apps. Each plot depicts the average consumption per each step across the 30 executions for (i) original app (in red), and (ii) modified version with GEMMA GUI composition for low energy-consumption (in cyan).	151
5.19 Example of GUI without significant improvement in the energy consumption for the Simple Deadlines app.	153
5.20 Example of GUI without significant improvement in the energy consumption for the Tasks: Astrid To-Do List app.	154
5.21 IdeaSoftware app: Original design <i>vs</i> GEMMA's solution (excerpts).	155
6.1 A general infrastructure for generating test sequences with different purposes	165

Supporting Evolution and Maintenance of Android Apps

Chapter 1

Introduction

The last decade has seen the tremendous proliferation of mobile computing in our society. Billions of users have access to millions of mobile applications that can be installed directly to their mobile devices and electrical appliances such as TV handsets without any complicated setup. Moreover, the current offering of mobile applications is not only limited to games or apps for entertainment but also represents several categories of common computing tasks that users more frequently choose to carry out in the intuitive Graphical User Interface (GUI) of a smart phone. When considering only the top-2 platforms in terms of market share, the app offerings are represented by more than 3M apps available for download: Google Play (the official store for Android apps) offers about 1.9M apps organized in 23 different categories (data for 2016 Q1), and Apple's App store contains 1.4M+ apps also in 23 domain categories (data for 2015 Q2).

From developers' perspective, there are several reasons why mobile economy is attractive to them; and these reasons are not limited only to cultural popularity of mobile devices and apps. Factors such as new monetization/revenue models, programming models, and distribution infrastructures contribute to an "attractive" movement that captivates new and traditional developers, as well as a crowd of other professionals that explore, design, and implement mobile apps [232]. Also, the need for "enterprise apps" that support start-ups or serve as a new front-end for traditional companies is pushing software-related

professionals to embrace the mobile technologies [232]. However, the nature of the economy (including devices, apps, markets) imposes new challenges to the way how mobile apps are envisioned, designed, implemented, tested, released, and maintained. The mobile development cycle is built around the usage of online app markets that (i) allow developers to publish apps and crowdsource requirements (as user ratings and reviews¹), and (ii) allow users to access a large offering of apps and to quickly discard/uninstall the apps when expectations are not accomplished by the apps. The rating/reviews mechanism in online app markets is a cheap way to collect thousands of real requirements from the users, which also imposes a significant overhead on the developers because of the effort required to analyze the user reviews, distinguish informative reviews from non-informative ones, and prioritize the feature requests/bugs for the next release.

The mobile development cycle is mostly represented by the iterative/incremental sequence: development \rightarrow publishing \rightarrow reviews analysis \rightarrow development \rightarrow publishing \rightarrow ... development, and there are very specific challenges that mobile developers currently face during the development cycle such as the constant pressure from the users for frequent releases (i.e., continuous delivery) [142, 133, 143]; rapid platform/library evolution and API instability [163, 168, 185, 69]; complex reporting and bug report reproduction involving complex multitouch gestures [100]; platform fragmentation [121]; impact of reviews and ratings on the success of the apps [125, 163, 134, 200, 97, 76, 190, 78]; cloning and reuse across different markets [192, 189, 165]; management of crowd-sourced requirements [78, 134, 201]; limited availability/adequacy of testing tools for mobile apps [143, 146]; limited computational resources in mobile devices (e.g., memory, battery); among the others. These challenges are not considered by traditional/contemporary software development methods and techniques; therefore, the mobile software development lifecycle and their disciplines (e.g., evolution and testing) require new models, methods, and techniques. The need for a new “software engineering” for mobile applications is

¹It is worth noting that reviews not only contain feature requests, but also bug reports, users’ sentiments, etc.

even more evident with respect to their **evolution and maintenance**, mainly due to market pressure demanding rapid release cycles and a mostly crowdsource-based nature of requirements management [201].

As a response to the challenges, tremendous amount of research has been conducted to propose a new software engineering toolbox for mobile development. However, some key challenges for **evolution and maintenance of mobile apps**, such as dynamic analysis, testing, and energy consumption, lack effective industrial-grade solutions to support developers and testers. This dissertation presents an in-depth analysis of these key issues and an initial set of solutions for *supporting evolution and maintenance of Android apps*, in particular native Android apps running on handheld devices. The proposed solutions are rooted into (i) empirical studies performed to understand the challenges of the mobile development ecosystem (Chapter 2), and (ii) an infrastructure designed and implemented to support large-scale dynamic analysis and testing of Android apps (Chapter 3). This dissertation also presents two automated approaches that support two key challenges for *evolution and maintenance of Android apps*: automatic test cases generation (Chapter 4) and automatic optimization of GUIs to reduce energy consumption (Chapter 5).

In summary, this dissertation makes the following research contributions:

- **Empirical Studies.** As part of the dissertation, a set of empirical studies were performed to analyze phenomena such as: Android API instability; developer practices for detecting/fixing performance bottlenecks and for micro-optimizing Android apps “in the wild”; usage of crowdsourced reviews to plan releases; developer practices for documenting and designing test cases; and energy-greedy APIs. The studies combined different data collection and analysis techniques, and for some of the studies new approaches were implemented to enable these analyses; for instance, as part of the analysis of developers’ reaction to Android API instability, we designed a traceability link recovery technique for linking StackOverflow questions to Android API changes. The studies helped improve current understanding of the nature of

the mobile development, practices performed by developers, and challenges specific to the Android ecosystem. Therefore, our studies and corresponding results have been widely discussed and cited by the research community. In fact, the studies on the Android API instability problem were the first of its kind in the software engineering community; moreover, we performed the first large-scale study aimed at analyzing real practices of open source developers for micro-optimizing Android apps, and for detecting/fixing performance issues in Android apps. Some of the key insights extracted from the studies are summarized as in the following:

- Change and fault proneness of the Android API is a threat to the success of Android apps, and developers lack automatic notification tools when changes to the API can introduce bugs in the apps [163, 168, 69];
- Developers strongly rely on user reviews when planning new releases of their apps and mainly rely on user reviews for detecting bugs experienced by the users as well as for suggestions of new features [201];
- Android developers prefer dynamic analysis and observation-based techniques to detect opportunities for improving the performance of their apps [161];
- Micro-optimizations are not widely used by open source developers, and the impact of micro-optimizations is noticeable only under certain “load” conditions that might appear only on specific types of apps;
- Manual testing is preferred over automated testing, and execution of use cases or sequences of test cases are the preferred criteria for designing test cases;
- Coverage analysis is not considered as useful to measure the quality of test cases by mobile developers;
- GUI and database related classes and methods in the official API are the most energy greedy [165].

The main findings from the empirical studies suggest that dynamic analysis is a crucial mechanism to support evolution and maintenance of Android apps; moreover, effective dynamic analysis requires scalable infrastructural support for automatic execution of mobile apps under different conditions (e.g., different devices).

- **An Infrastructure for Large-scale Analysis and Testing of Android apps.**

Fragmentation at OS and device level is one of the most pressing challenges for mobile development because during evolution and maintenance, new features and fixes should be tested under different contextual events on a multitude of different platforms. However, manual testing is still preferred over automated approaches; in fact, development teams lacking testing resources usually test their apps only on a subset of devices [143, 149]. One of the main reasons for the preference for manual testing is the expensive maintenance of automated test cases and the lack of tools supporting large-scale testing. Therefore, we designed T+, an infrastructure for automatic generation of actionable test cases that can target different purposes and combine different sources of information. Under T+ we also designed the Execution Engine, which is an extensible infrastructure for large-scale execution of Android apps on virtual devices. T+ was published at the ACM SIGSOFT Student Research Competition at the *37th IEEE International Conference on Software Engineering (ICSE 2015)* [162], and won the Gold Medal (First Place) in the Graduate Student Category. The Execution Engine was inspired by the taxonomy of accidental and essential issues (and corresponding solutions) for large-scale execution of Android apps; this taxonomy was devised as part of the industrial collaboration with a major telecommunication company (Huawei). Both, T+ and the Execution Engine, are the foundations of our proposed solutions for supporting automated GUI testing and improving energy consumption of Android apps.

- **Supporting Automated GUI testing.** We propose a novel hybrid approach for combining GUI and usage models for automatic generation of actionable test cases.

Our approach (namely MONKEYLAB) derives feasible and fully replayable GUI-based event sequences for (un)natural app usage scenarios (i.e., actionable scenarios), which provides stakeholders with an automated approach for scenario generation that can be as powerful as manual testing (Chapter 4). MONKEYLAB includes a novel mechanism for generating actionable scenarios that is rooted in mining event traces, generating execution scenarios via statistical language modeling, static and dynamic analyses, and validating these resulting scenarios using interactive executions of the app on real devices (or emulators). In particular, we explore interpolated n-grams and back-off models, and we propose three different flavors (i.e., up, down, and strange) for generating (un)natural scenarios. MONKEYLAB was originally published in the *12th IEEE Working Conference on Mining Software Repositories (MSR 2015)* [171].

- **Improving Energy consumption.** Finally, we propose a multi-objective approach, called GEMMA, for generating color compositions that reduce the energy consumption of GUIs in Android apps and are visually attractive at the same time (Chapter 5). GEMMA combines power models, pixel-based engineering, color theory, dynamic analysis, and a multi-objective optimization technique to produce a Pareto-optimal set of design solutions (i.e., GUI color compositions) across three different objectives: (i) reducing energy consumption, (ii) increasing contrast, and (iii) improving the attractiveness of the chosen colors by keeping the palette close to the original one. GEMMA is the very first approach adopting multi-objective optimization to choose colors for mobile apps with the main goal of reducing energy consumption, that accounts for multiple screens and the duration of time they are displayed, and produces pleasant and consistent color combinations. GEMMA was originally published in the *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)* [167] and won ACM SIGSOFT Distinguished Paper

Award and the Department of Computer Science Graduate Park Award.

Chapter 2

Empirical Studies

The way the mobile apps are developed represents a paradigm shift (as compared to the traditional desktop and web applications) not only because of gesture-driven interaction of these apps, but also due to the differences in the development cycle. For instance, apps are distributed using online markets; release periods are shorter; features planning and development is highly impacted by crowdsourced requirements; and device/OS fragmentation introduces additional overhead in terms of testing efforts. However, by the time when this thesis was conceived, several of the mobile development challenges remained without solutions, and the research community lacked understanding of important facets of mobile apps maintenance and evolution. Proposing solutions and new approaches supporting evolution and maintenance of Android requires understanding of developers' practices and challenges. Therefore, in the first research thrust, several facets from the Android mobile ecosystem were analyzed by conducting empirical studies. The studies aimed at understanding current practices performed by practitioners in-the-wild, and the challenges they face when maintaining and evolving Android apps.

The decision to conduct empirical studies was motivated by the lack of reports, evidence, and empirical data describing (i) how practitioners evolve and maintain mobile apps, and (ii) the disciplines in the development cycle associated with the evolution and maintenance tasks. In that sense, we relied on several methods widely used in Empirical

Software Engineering for data collection and analysis, such as mining software repositories, grounded theory, surveys with practitioners, and controlled experiments.

In this chapter we summarize the studies and the main findings; we do not provide all the details concerning motivation, study design, results, and the related work. However, we invite the interested reader to check all the details of each study in the corresponding publications listed in Section 2.8.

2.1 Android API instability [163, 69]

Stability and fault-proneness in the Android API is a sensitive and timely topic, given the frequent releases and the number of applications that use these APIs. Therefore, we designed two case studies [163, 69] to provide solid empirical evidence and shed some light on the relationship between the success of apps (in terms of user ratings), and the change- and fault-proneness of the underlying APIs (i.e., Android API and third-party libraries). In the first study we analyzed to what extent the APIs fault- and change-proneness affect the user ratings of the Android apps using them, while in the second we investigated to what extent Android developers experience problems when using APIs and how much they feel these problems can be causes of unfavorable user ratings/comments.

The first study (in the following referred to as “API-Study I”) was conducted on a set of 5,848 Android free apps belonging to different domains. We estimated the success of an app based on the ratings posted by users in the app store (Google Play¹). Then, we identified the APIs used by those apps, and computed the number of bug fixes in those APIs. In addition to the bug fixes, we computed different kinds of changes occurring in such APIs, including changes in the interfaces, implementation, and exception handling. Finally, we analyzed how the user ratings of an app are related to APIs fault- and change-proneness, specifically to different kinds of changes occurring to APIs. This study was mainly aimed at providing possible quantitative evidence about relationship between APIs

¹<http://play.google.com> verified on January 2014.

fault- and change-proneness, and the apps' ratings. However, especially because we have no visibility over the source code of such apps and of their issue trackers, it is difficult to provide a strong rationale and, possibly, a cause-effect relationship for such findings.

In order to provide explanations to the findings of API-Study I, we conducted a second study (in the following referred as "API-Study II"). This study consisted of a survey, and it involved 45 professional Android developers. We asked such developers to fill-in a questionnaire composed by 21 questions organized into five categories: (i) developer's background, (ii) factors negatively impacting user ratings, (iii) frequent reasons causing bugs/crashes in Android apps, (iv) experiences with used APIs, and (v) impact of problematic APIs on the user ratings. Then, we quantitatively analyzed the answers to 19 questions by using descriptive statistics, and completed the analysis with qualitative data gathered from the other two questions.

It is important to point out that this work does not claim a cause-effect relationship between APIs fault- and change-proneness and the success of apps, which can be due to several other internal (e.g., app features and usability) and/or external (e.g., availability of alternative similar apps) factors. Instead, the purpose of the study was to investigate whether the change- and fault-proneness of APIs used by the app relates (or not) to the app success, measured by its ratings. That is, a heavy usage of fault-prone APIs can lead to repeated failures or even crashes of the apps, hence, encouraging users to give low ratings and possibly even abandoning the apps. Similarly, the use of unstable APIs that undergo numerous changes in their interfaces can cause backward compatibility problems or require frequent updates to the apps using those APIs. Such updates, in turn, can introduce defects into the applications using unstable APIs. In the following we summarize the research questions and corresponding results. Complete details of the studies are with the FSE and TSE publications [163, 69].

2.1.1 RQ₁: Does the fault-proneness of APIs affect the user ratings of Android Apps?

APIs used by apps having higher user ratings are, on average, significantly less fault-prone than APIs used by low rated apps. However, it is interesting to understand if the observed difference in terms of APIs fault-proneness between apps having different levels of rating is due to the used official Android APIs, third-party APIs, or to both of them. To this aim, we separately investigated the fault-proneness of the official Android APIs and of the third-party APIs used by the apps object of our study.

Concerning the official Android APIs, apps having a *high* rating use APIs that underwent, on average, 6.2 bug fixes, as compared to the 9.7 (+56%) of apps having a *medium* rating and the 13.0 (+109%) of apps having a *low* rating. This result is inline with what we observed when analyzing all the used APIs as a whole. Also the results of the Mann-Whitney test confirm that official Android APIs used by apps having a higher average user rating are, on average, significantly less fault-prone than APIs used by low rated apps. Indeed, as already observed when considering all APIs, apps having a higher rating always exhibit a statistically significant lower number of bug fixes in the used APIs than apps having a lower rating (p -value always < 0.0001).

When analyzing third-party APIs in isolation we only considered the 1,224 apps using at least one third-party API since not all the considered apps use third-party APIs. In this case we observed a slightly different trend:

- Apps having a *high* rating use third-party APIs subject, on average, to 1.3 bug-fixing activities.
- Apps having a *medium* rating use third-party APIs subject, on average, to 3.6 bug-fixing activities (+177%).
- Apps having a *low* rating use third-party APIs subject, on average, to 2.7 bug-fixing activities (+108%).

Thus, while it is confirmed that apps having a *high* rating use less fault-prone APIs than apps having a *medium* and a *low* rating, from the average values it seems that apps having a *medium* rating use APIs more fault prone than those used by apps having a *low* rating. However, by looking into the data we found that this result is mainly due to a set of 28 apps falling in the *medium* rating category and all using the same (fault-prone) third-party APIs. In particular, these 28 apps are developed by the same software house² and use APIs subject to a number of bug-fixes going from a minimum of 23 to a maximum of 46, clearly raising the average value of bug-fixes in the *medium* rating category. In fact, when comparing the fault-proneness of the three categories by using the Mann-Whitney test, we obtained that apps having higher ratings use APIs statistically significant less fault-prone than low rated apps, even when comparing apps having a *medium* rating with those having a *low* (p -value always <0.0001 , with a small effect size).

Summarizing, the results of our **RQ₁** show that the higher the ratings of the apps, the lower the fault-proneness of the APIs they use. This holds when considering all APIs, as well as the official Android APIs and third-party APIs in isolation.

2.1.2 **RQ₂: Does the change-proneness of APIs affect the user ratings of Android Apps?**

As done for the fault-proneness, we also analyzed the change-proneness of APIs used by the different categories of apps by isolating official Android APIs and third-party APIs. Concerning the official Android APIs, we observed that those used by apps having high user ratings are significantly less change prone than those used by low rated apps, as also confirmed by the results of the Mann-Whitney test (p -value always <0.0001 with a small effect size). In particular:

- In terms of overall method changes, apps having a *high* rating use APIs that underwent, on average, 25 changes, as compared to the 37 (+48%) of apps having a

²<http://www.androidpit.it/it/android/market/applicazioni/list/owner/LightCubeMagic> verified on January 2014

medium rating and the 48 (+92%) of apps having a *low* rating. This trend is also confirmed when just considering changes to public methods, with apps having *low* rating using APIs subject to 27 changes, on average, 80% more than the apps having *high* rating.

- When focusing on changes performed on method signatures, apps having a *high* rating use APIs object, on average, of 5 changes, 40% less than APIs used by apps having a *medium* rating and 80% less than APIs used by apps having a *low* rating. These results are also confirmed when just focusing on public methods.
- If restricting our analysis to the Android APIs only, we do not observe any statistically significant difference in terms of changes performed to the exceptions thrown by methods between the different categories of apps.

Turning to the third-party APIs, the results of the Mann-Whitney test showed that the change-proneness of APIs used by apps having high user ratings is lower in a statistically significant way. Moreover, when comparing apps having a *high* rating with those having a *low* rating, we obtain a large effect size for all type of changes. For instance, when considering all changes performed to the API methods, we go from the three changes, on average, of APIs used by apps having a *high* rating to seven changes (+133%) of APIs used by apps having a *low* rating. The same trend has been also observed when (i) just focusing on public methods, and (ii) just considering the changes occurred to (public) methods' signature.

Summarizing, the results of **RQ₂** show that the higher the average ratings of the apps, the lower the change-proneness of the APIs they use. This holds when considering all APIs, as well as when restricting our attention to official Android APIs or third-party APIs only. Instead, there is no significant difference when the changes are on the exceptions thrown by API methods. Again, this result holds for all APIs as well as for the official Android APIs and the third- party APIs considered in isolation.

2.1.3 RQ₃: To what extent Android developers experience problems when using APIs?

Among the 45 developers answering our questionnaire, 33 (73%) said they have experienced problems with the used APIs. Of these 33, 21 indicated Android APIs as the cause of the problems, and 12 indicated third-party APIs. Again, this is likely because most of the APIs used in the apps belong to the Android SDK, and only few of them are third-party ones³. Also, 64% of developers (29) declared to have observed new bugs in their apps introduced as a consequence of new releases of the Android platform. Summarizing, the study results for RQ₃ indicate that:

- A large percentage of the developers (between 44% and 71%) consider change- and fault-proneness of APIs as threats to the proper working of their apps. When focusing on problems related to the APIs (i.e., considering all the answers but the “*Java programming errors in the app*” one), developers perceive that bugs present in third-party APIs represent the most frequent cause of bug introduction in their apps.
- Developers are generally more concerned about the effect of bugs present in the used APIs than about changes performed in new releases of the used APIs; this is true for both third-party as well as official Android APIs.
- Developers believe that more bugs are present in third-party APIs than in the official Android APIs. However, they are more concerned about the change-proneness of the Android platform than of the change-proneness of third-party APIs. This result likely has a two-fold explanation. First, the Android APIs have been object of a very fast evolution⁴ leading to 18 major releases over just four years. It is very unlikely that also third-party APIs have evolved so fast. This is also confirmed by the average

³Note that in our first study, we found just 21% of the considered apps to use at least one open source third-party API.

⁴https://developer.android.com/reference/android/os/Build.VERSION_CODES.html verified on January 2014.

frequency of commits per month observed in Study I for the Android APIs (164 commits per month) as compared to the third-party APIs (14 commits per month). Thus, developers have more likely experienced bugs introduced by major changes in the Android APIs than by changes in the used third-party libraries. Second, Android API reuse by inheritance is widely implemented by developers [192, 189], and Android apps are highly dependent on the official Android APIs [222]. Almost 50% of classes in Android apps inherit from a base class as shown in a recent study by Mojica Ruiz *et al.* [192]. This, again, makes more likely for developers to experience bugs due to changes in the official APIs than in third-party APIs.

2.1.4 RQ₄: To what extent Android developers consider problematic APIs to be the cause of negative user rating/comments?

Of the 45 surveyed developers, 28 (62%) declared to have observed a relationship between problems experienced with the used APIs and bad user's ratings/comments. These 28 developers evaluated the severity of the observed impact, providing a score on a five point Likert scale between 1=very low and 5=very high. The median is 4 (i.e., *high impact*) indicating that the use of problematic APIs could strongly impact the rating of an app from the developers' point-of-view. Also, it is important to note that no one of the developers assessed the impact at a value lower than 3 (i.e., *medium impact*). This means that developers, in their experience, not only observed a decrease of the ratings assigned by users to their apps as consequence of problems in the used APIs, but also that this decrease was substantial.

The answers provided by developers to questions related to **RQ₄** indicate that 62% of developers perceived a direct relationship between problems experienced with the used APIs and bad users' ratings/comments, and the impact of such APIs on the apps' user ratings was considered as *medium-high*. Also, the discussed examples support the quantitative results obtained in our first study: the use of problematic APIs could represent a threat for the success of Android apps.

2.2 Usage of Crowdsourced Requirements [201]

The distribution of updates—related to the introduction of new features and to bug fixes—through app online stores is accompanied by a mechanism that allows users to rate releases using scores (i.e., star ratings) and text reviews. The former (i.e., the score) is usually expressed as a choice of one to five stars, and the latter (i.e., the review) is a free text description that does not have a predefined structure and is used to describe informally bugs and desired features. The review is also used to describe impressions, positions, comparisons, and attitudes toward the apps [134, 78, 203, 120, 200]. Therefore, online store reviews are free and fast crowd feedback mechanisms that can be used by developers as a backlog for the development process. Also, given this easy online access to app-store-review mechanisms, thousands of these informative reviews can describe various issues exhibited by the apps in certain combinations of devices, screen sizes, operating systems, and network conditions that may not necessarily be reproducible during regular development/testing activities.

By analyzing ratings and reviews, development teams are encouraged to improve their apps, for example, by fixing bugs or by adding commonly requested features. According to a recent Gartner report’s recommendation [142], given the complexity of mobile testing “*development teams should monitor app store reviews to identify issues that are difficult to catch during testing, and to clarify issues that cause problems on the users’ side*”. Moreover, useful app reviews reflect crowd-based needs and are a valuable source of comments, bug reports, feature requests, and informal user experience feedback [76, 78, 97, 134, 200, 147, 191]. However, because of this easy access to the app stores and lack of control over the reviews’ content, relying on crowdsourced reviews for planning releases may not be widely used by mobile developers because of (i) the large amount of reviews that need to be analyzed, and (ii) some of these reviews may not provide tangible benefits to developers [78].

In this section we summarize the results of an empirical study [201] aimed at investi-

gating to what extent app development teams can exploit crowdsourcing mechanisms for planning future changes, and how these changes impact user satisfaction as measured by follow-up ratings. We performed (i) a mining study conducted on 100 Android open source apps in which we linked user reviews to source code changes, and analyzed the impact of implementing those user reviews in terms of app success (i.e., ratings); and (ii) an online survey featuring 73 responses from mobile app developers, aimed at investigating whether they rely on user reviews, what kind of requirements developers try to elicit from user reviews, and whether they observe benefits in doing that. The study makes the following noteworthy contributions:

- A novel reviews-to-code-changes traceability recovery approach, called CRISTAL. The approach combines classic Information Retrieval (IR) based traceability recovery approaches (*e.g.*, [61, 184, 209, 208]) with some specific heuristics to deal with (i) diversity and noise in crowd reviews, and (ii) inherent abstraction mismatch between reviews and developers' source code lexicon.
- Results of a survey conducted with 73 Android developers and aimed at investigating the developers' perception of user reviews, to what extent they address them and whether they observe any tangible benefits from such an activity.
- Results of an empirical study conducted on 100 Android apps. The study exploits CRISTAL to provide quantitative evidence on (i) how development teams follow suggestions contained in informative reviews, and (ii) how users react to those changes.
- A comprehensive replication package [202]. The package includes all the materials used in our studies.

In the following we summarize the research questions and corresponding results. The full details of the studies can be found in our ICSME'15 publication [201].

2.2.1 RQ₁:To what extent do developers fulfill reviews when working on a new app release?

Results obtained by mining 100 Android apps show that, in most cases, developers carefully take into account user reviews when working on the new release of their app. Indeed, on average, 49% of the informative reviews are implemented by developers in the new app release. The survey respondents confirmed such findings, claiming that they often rely on the information in user reviews when planning a new release of their app (49% very often, 38% often). They mainly look in user reviews for bug reporting (75%) and suggestions for new features (68%). Also, they identify useful reviews by searching for the constructive and detailed ones (44%), containing more than just emotional expressions. The vast majority of the survey respondents (75%) indicate that just a small percentage of user reviews is informative (< 25%).

2.2.2 RQ₂:What is the effect of a crowd review mechanism (for planning and implementing future changes) on the app success?

Developers of Android apps implementing user reviews are rewarded in terms of ratings; in fact, 90% of the surveyed developers (i.e., 66 out of 73) observed an increase of their app ratings as a consequence of user requests they implement. This is confirmed by the observed positive correlation (0.59) between *review coverage* and *change in overall score* between the old and the new app releases. Also, our qualitative analysis supports, at least in part, our quantitative findings. Answers provided by the surveyed developers confirmed that the small set of informative reviews represents a very precious source of information, leading to the fixing of bugs difficult to catch during testing activities, to the implementation of new successful features, and to the improvement of the app's non-functional requirements. The vast majority of the respondents (90%) believes that implementing requests from user reviews has a positive effect on the app's success (as assessed by the user ratings).

2.3 How Android Developers Detect and Fix Performance Bottlenecks [161]

Despite the efforts of researchers [119, 174, 199] and mobile API designers in providing developers with guidelines and best practices [9, 62, 60] for improving the performance of mobile apps, there is still a significant gap between research and practice in terms of dealing with performance issues by developers in the wild [174, 147].

Our work is empirical in nature and is aimed at filling this important gap and at increasing the understanding of current performance related practices by developers of Android apps. While previous empirical studies [119, 174, 199] focused on understanding and categorizing performance bugs, the study described in this section [161], analyzed *real practices* that are followed and *actual tools* that are used by developers to fix performance related bugs. In the context of the study, 485 contributors of open source Android apps and libraries hosted on GitHub were surveyed, inquiring about their practices and tools for detecting and fixing performance bottlenecks; then, the repositories (i.e., bugs from the issue trackers and commits from the code change histories) of their apps were manually analyzed to study real performance bottlenecks and fix-inducing commits to investigate actual strategies followed by developers to deal with performance related issues. The study makes the following noteworthy contributions:

- To the best of our knowledge, this is the first study aimed at analyzing *real practices* of open source developers for detecting and fixing performance issues in Android apps;
- The study provides an overview and key insights into types of performance related issues faced by developers as well as prevalent practices and tools used to deal with performance bugs and bottlenecks. The results also reveal current performance related needs of developers that can be used to drive future efforts of researchers;
- The findings complement previous studies of performance bottlenecks in Android

apps by providing the viewpoint of real developers via analysis of their practices;

- An extensive online appendix that includes the anonymized answers collected from the survey, the list of tools reported by the participants, a taxonomy of practices for fixing performance bottlenecks, and examples of real performance bottlenecks in Android apps [25].

In the following we summarize the research questions and corresponding results. Complete details of the study can be found in our ICSME'15 publication [161].

2.3.1 RQ₁: What practices are used by Android developers to detect performance bottlenecks?

The results suggest that open source Android developers primarily rely on *manual testing* and *analysis of the reviews* for detecting performance bottlenecks. Manual execution of apps is also accompanied by tools that help measure and visualize performance-related measurements (e.g., execution time and memory). Despite the availability of tools for profiling performance-related measurements, a number of practitioners still rely on manually inserting statements in the code to measure and print execution time and memory consumption.

2.3.2 RQ₂: What tools are used by Android developers to detect performance bottlenecks?

Although there is a diverse set of available tools, Android developers mostly rely on tools for performance profiling and debugging of their apps; most of the tools are from Google, as expected. There is a preference towards tools supporting observation-based analysis, and most of the tools do not support automatic detection of bottlenecks. Only a few tools support automatic detection of a limited number of types of performance bottlenecks (Google's StrictMode Android API and PerfChecker tool [174]), yet these tools are not

widely used. Static analysis tools are used less frequently and only in a few cases for detecting performance optimizations.

2.3.3 RQ₃: What practices are used by Android developers to fix performance bottlenecks?

We obtained 72 codes (after the open coding on free-text answers) identifying individual practices for improving performance bottlenecks. These 72 codes were further categorized into 16 groups by considering the type of practice (e.g., threading) or the goal (e.g., memory management). Finally, these 16 groups were linked to the following types of high-level categories: *GUI lagging*, *Memory bloat*, *Energy leak*, *General-purpose*, *Unclear-benefit*. The first three categories represent performance bugs defined by Liu *et al.* [174]; thus, the groups linked to these categories represent practices aimed at solving corresponding bug types (e.g., GUI optimization aims at reducing “Application Not Responding-ANR” errors and GUI lagging). The last two categories are for general-purpose cases (e.g., profiling) and when the perceived benefit is unclear. The complete taxonomy (codes, groups, and categories) is in our appendix [25].

The most frequent practices used by the surveyed developers for fixing performance bottlenecks include the usage of multi-threading to avoid lengthy operations in the main thread, GUI optimizations for reducing the complexity of the UI, caching to avoid redundant or blocking/time consuming resource accesses, memory management to avoid GC events and OOM errors, and source code optimizations.

2.4 How Developers Micro-Optimize Android Apps

Micro-optimizations consist of changes to the source code that are applied mostly at statement level and are not intended to change the system design or architecture. Some of the tools for detecting potential micro-optimizations (e.g., *FindBugs*, *LINT*, *PMD*) have been widely adopted by software developers and companies [67] building Java and

C/C++ systems. However, the effectiveness of the suggestions (i.e., warnings about micro-optimization opportunities or bugs) provided by these tools has been previously questioned in the context of fault detection in C/C++ and Java systems [234, 247, 231, 230, 247, 66]. Moreover, little research has been done into understanding whether mobile developers perform micro-optimizations, given the fact that mobile apps are more prone to performance bugs [174].

In this section, we summarize the results of three empirical studies aimed at understanding the usage of micro-optimizations in the wild, the reasons, and their impact when optimizing Android apps⁵. First, we measured the persistence of micro-optimization opportunities⁶ in change histories of open source Android apps; in particular, we mined the change histories of 3,513 Android apps hosted on GitHub to identify the most frequent micro-optimization opportunities suggested by two static analysis tools (i.e., *PMD* and *LINT*) in 297K+ snapshots of these apps (at commit level) and to understand if (and when) developers implement these micro-optimization opportunities. Second, we analyze the effectiveness of micro-optimizations on the resource consumption of Android apps by means of an in-depth analysis into whether implementing suggested micro-optimizations can help reduce memory and CPU usage in a sample of eight Android apps; to this we implemented a tool that automatically refactors micro-optimization opportunities and builds APKs from refactored source code. Finally, we investigated current practices of Android developers for improving the performance of their apps by conducting a survey involving 389 open-source Android developers to understand the state-of-practice with respect to micro-optimizing apps and the reasons for applying (or not) micro-optimizations in practice. One limitation of our study is that Android-specific micro-optimizations for the GUI were not included, because they can not be automatically performed and their implemen-

⁵The publication supporting this Section is currently under review at the Journal: Empirical Software Engineering. This is a joint work with Christopher Vendome and Michele Tufano from the SEMERU group.

⁶We use the term micro-optimization opportunities because (i) these are suggestions (more than must-do changes) reported by static analysis tools, thus these tools might suggest false positives, and (ii) micro-optimizations effectiveness might depend on the context (e.g., the impact of string micro-optimizations is noticeable with certain volume of string operations).

tation is always app-specific. Therefore, this paper represents a “starting point” for future efforts in terms of validating the impact of existing Android-specific micro-optimizations and providing developers with micro-optimizations that can have real impact on the performance of their apps.

The studies summarized in this sections make the following noteworthy contributions:

- To the best of our knowledge, this is the first set of studies aimed at analyzing micro-optimization practices of open source Android developers.
- The studies contribute to understanding the nature of Android apps and the need for specific practices that consider the context of the apps (i.e., device, OS); the theoretical assumption that practices with certain success in non-mobile systems can be transferred to mobile apps with the same results is not entirely valid as in the case of micro-optimizations, which require specific conditions that are not available in mobile apps (e.g., large number of String operations, or a large number of instantiations of objects in loops).
- An extensive online appendix that includes the anonymized answers collected from the survey, a taxonomy of practices for detecting micro-optimization opportunities, detailed results from the measurements study, and results from the mining study [26].

In the following we summarize the research questions and corresponding results. The publication supporting the results is under review at the journal of Empirical Software Engineering.

2.4.1 RQ₁: What is the distribution of micro-optimization opportunities across Android Apps at GitHub?

We utilized *LINT* [32] and *PMD* [39] to identify performance-related warnings (i.e, micro-optimization opportunities) in 3,513 Android open source projects. In total, we detected 56 types of micro-optimization opportunities (24 from *LINT* and 32 from *PMD*). The results

suggest that Android open source developers mostly do not implement micro-optimizations to improve mobile apps performance. When analyzing the most recent snapshot of the 3,513 open source Android apps, we found a large number of micro-optimizations opportunities (detected by *PMD* and *LINT*). Importantly, we observed that the density of these micro-optimization (*i.e.*, normalized by the number of files and LOC) reflected the high prevalence of micro-optimizations, since the densities mirrored the results for the raw frequencies. The top three micro-optimization types (*i.e.*, `MethodArgumentCouldBeFinal`, `LocalVariableCouldBeFinal`, `UnusedResources`) amount to over 1.5 million instances where micro-optimizations might be implemented. The next two micro-optimization opportunities in the top-5 list are `UnusedIds` and `AvoidInstantiatingObjectsInLoops` accounting for 53K+ instances.

2.4.2 RQ₂: To what extent are micro-optimizations introduced or avoided during the evolution and maintenance of Android apps?

The analysis method for **RQ₂** consisted of identifying the trends, by finding the function that best describes (*i.e.*, best fit) the evolution of the number of performance-related warnings (*i.e.*, micro-optimization opportunities) for each project at commit level. The function fitting was implemented in MATLAB targeting a diverse set of functions to avoid bias towards widely analyzed models (*e.g.*, linear and power); in particular, for the fitting, we used the following functions available with MATLAB: linear, quadratic, cubic, exponential, power, logarithmic, Weibull, sine, and first order Fourier series model (`fourier1`).

We observed a dominance of a first order *fourier* model fitting the number of micro-optimization opportunities at each commit of the analyzed apps, when considering both global data (*i.e.*, all the micro-optimization categories) and data for each category. The global distributions only showed *linear-const* as the dominant fitting at the 5th ranking (*i.e.*, top-5) of functions (235 projects), but it was prevalent for certain history sizes and particular warning categories. Of these projects, only 12 projects never had a micro-

optimization opportunity introduced during their revision history. However, we observed a high prevalence of *linear-constant* as the top two dominant function, when we investigated the warnings categories. Additionally, we observe a decreasing function, in the top three fittings for the global data, but when normalized by LOC and only for projects with long histories. The results indicate that the most prevalent behaviors in terms of micro-optimization opportunities detected in the analyzed apps are (i) the opportunities increase during the app history, and (ii) the opportunities are constant, therefore, it suggests that developers mostly do not implement micro-optimizations in mobile apps.

2.4.3 RQ₃: How do micro-optimizations impact CPU and memory usage of Android apps?

To measure the impact of micro-optimizations, we focused on apps with a large number of micro-optimizations opportunities. Instead of measuring the impact of incrementally applying one, two, three, etc., micro-optimizations, we decided to measure the impact in a scenario when all the micro-optimizations opportunities are addressed. To this, we developed a tool to automatically refactor (i.e., apply micro-optimizations) Android projects (i.e., source code), based on the list of warnings reported by *PMD* and *LINT*. We utilized a combination of AST-based analysis and textual transformations depending on the type of violation and required fix. The implementation of the micro-optimizations is based on the practices and solutions suggested by *PMD* and *LINT*. The tool generates a refactored version of an app (source code and APK) implementing either all the micro-optimizations belonging to a target category, or all the micro-optimizations in all the categories. Our tool after modifying the source code, automatically builds Android APKs. In addition to our tool, we used the optimizations provided by *Proguard* [40], which is the state-of-the-practice tool for optimizing APKs. We used the specific configuration of *Proguard* for optimizing apps as suggested by the related work [218]. In particular, this configuration performs (i) peephole optimizations for arithmetic instructions and casting operations, (ii) removes write-only fields, (iii) marks fields as private, if possible, (iv) propagates the

values of fields across methods, and (v) merges classes vertically and horizontally in the class hierarchy, whenever possible. It is worth noting that the micro-optimizations implemented by our tool and the ones in *Proguard* are not the same, thus, each tool provides a different set of optimizations. Also, note that the micro-optimizations performed by our tool are done in the source code, meanwhile *Proguard* optimizations work directly at the bytecode level.

The results on eight open source apps (AnCal, AndroidBicycle, Hangeulider, HealthFoodConcepts, SaveApp, RaspberrybusMalaysia, Subsonic-Android, HeartsSky) with a large number of micro-optimization opportunities, demonstrate that the analyzed micro-optimizations do not significantly improve CPU or memory consumption (except for the UnusedResources case), even in those apps that contain a high number of warnings. However, we found evidence that developers can significantly reduce the memory consumption of the running app, by removing unused resources, which can be automatically detected and performed by the Android *LINT* tool.

2.4.4 RQ₄: What practices are used by developers for detecting optimization opportunities in Android apps?

Although the current practices and tools based on dynamic analysis do not support automatic detection of optimization opportunities, these techniques were preferred by the survey participants over static analysis tools for ad-hoc fixing of performance related issues or early detection of micro-optimization opportunities. In general, 371 informative answers from the survey revealed the existence of a diverse set of practices and tools for detecting optimization opportunities. However, the responses show a bias in preference towards dynamic-based practices and tools provided by Google. Only 82 participants included in their answers static analysis-based practices, and only 15 participants indicated use of static analysis tools (e.g., *PMD*, *FindBugs*, *LINT*) in order to detect optimization opportunities. Manual testing and profiling turned out to be the most preferred practices. Only two participants rely on automated unit testing tools. In the case of

profiling, the most used suites and tools belong to the Android ecosystem, such as Eclipse MAT[17], DDMS [114], Android Device Monitor [104], traceview [109], systrace [111], and dmtracedump [109]. Only 20 answers included third-party tools such as Valgrind [51], LittleEye [33], AT&T ARO [8], Intel Parallel Studio [29], Adreno profiler [34], among the others. A detailed taxonomy of practices is provided in the online appendix [26].

2.4.5 RQ₅: What micro-optimizations are recognized (and used) by developers as useful for improving the performance of apps?

The practice of detecting and implementing micro-optimizations is not prevalent among Android developers who responded to our survey. Most of the participants are unaware of this practice, claiming that their apps do not require micro-optimizations, or that they do not trust in “premature optimization”. However, some developers reported successful usage of some micro-optimizations, which suggests that, in general, the impact of micro-optimizations might depend on the application domain (e.g., games) and the operations workload involved in the micro-optimizations (e.g., large volume of String operations)

2.5 Energy Greedy APIs [165]

Programming errors, hardware interactions, and API misuses can cause high levels of energy consumption (also known as *energy bugs*) in mobile apps [204]. To identify such problems, effective strategies for measuring energy consumption in mobile devices are needed. In the literature, several different strategies have been proposed, based on real measurements [77, 96, 127, 144, 152, 225] and power modeling [122, 123, 205, 206, 241, 246]. While previous work attempted at characterizing energy bugs in mobile devices [77, 82, 122, 205, 225], most of these classifications have been done either by mining software repositories (e.g., bug reports, forums, commit logs) [204, 229, 244] or by using dynamic tainting [172, 245]. Thus, there is a clear gap in the research literature on how and where the uses and misuses of APIs can lead to energy bugs based on large-scale

empirical data. Up-to-date, only the *wakelock* and *GPS* related APIs and their misuses have been studied and linked to energy bugs [206, 207, 229, 244].

Based on these considerations, we conducted a quantitative and, above all, qualitative exploration into how different API usage patterns can influence energy consumption in mobile apps [165]. We mined and analyzed thousands of instances of energy-greedy method calls and API usage patterns by measuring their energy consumption in 55 free Android apps belonging to different domain categories. In order to collect energy consumption values for each API call, we used a hardware-based approach for collecting actual energy measurements and aligned those values with execution traces generated from real usage scenarios on mobile apps. Once energy consumption data were collected for all execution scenarios, we traced all API calls back to the source code where they appear using an approach that we specifically implemented in this paper. Note that, since previous studies had already shown that 3G/GSM and GPS were energy-greedy hardware components [77, 122, 205, 206], in this study we were not interested in measuring the energy consumed by APIs related to those components.

Using the proposed measurement framework, we analyzed the consumption of individual APIs as well as their usage patterns. Firstly, we quantitatively identified the APIs and the patterns exhibiting high energy consumption in 55 free Android apps. Then, several evaluators inspected—following a categorization approach inspired from the grounded theory [98]—thousands of code fragments where such APIs/patterns occurred. In some cases, we found evidence of energy bugs due to API misuses or suboptimal API choices. Overall, the work is in the trend of mining energy consumption patterns for software systems, originally pioneered in this community by the work of Hindle [127]. The main contributions of this study are:

- An approach, based on power measurement and mining source code, for identifying API calls and usage patterns exhibiting high energy consumption;
- Quantitative ranking of high energy-demanding APIs and usage patterns, which

have been traced back to original source code in the apps;

- Results of a qualitative analysis aimed at understanding energy hotspots in API usage patterns and discussing whether alternative solutions are available;
- Actionable knowledge and recipes for developers on how to reduce energy consumption while using certain categories of Android APIs and patterns.

In the following we summarize the research questions and corresponding results. Complete details of the study are with the MSR publication [165].

2.5.1 RQ₁: Which are the most energy-greedy Android API methods?

Out of all the analyzed APIs, 131 represent negative outliers in terms of energy consumption (i.e., energy greedy APIs). While the average energy consumption for the remaining 676 (807-131) methods is $4e-5$ J on average, for the top-131 methods it is $5e-3$ J on average (125 times higher) with peaks of 0.15 Joule, i.e., more than 3,000 times than the average of all the methods. Note also that these API invocations are scattered across all the 55 analyzed apps, with a total of 9,609 instances (mean 73, median 23).

APIs related to *GUI & Image Manipulation* and *Database* represent, all together, 60% of the energy-greedy APIs. Concerning the *GUI & Image Manipulation* category, an interesting case is represented by the method `notifyDataSetChanged` of class `ArrayAdapter`. In the Android documentation, this method is described as the one in charge of notifying the attached observers that the underlying data has been changed and any `View` reflecting the data set should refresh itself. This method is energy-greedy due to the necessity of refreshing all views when changes happen to the data represented in them. This method appears to be a well-known energy bottleneck and it has been spotted and discussed by Android developers on Stack Overflow [219]. However, this example represents one of those cases where developers have no choice to implement automatic updating of view basic blocks in response to data changes.

Several methods from the *Database* category are needed in the management of an SQLite database on Android. We go from the database opening `SQLiteDatabase.openDatabase`, to its querying `SQLiteDatabase.query`, until its deletion `ContextWrapper.deleteDatabase`, for a total of 30 energy-greedy APIs. It should be noted that energy-greedy APIs responsible for XML files manipulation are, instead, very rare, with only the method `XML.newSerializer` belonging from the *File Manipulation* category. Of course, this does not mean that the use of an SQLite database represents an energy bug, but just that developers should seriously think if their application really needs to use a relational database as a storage layer or, instead, could also store persistent data in XML files or using `SharedPreferences` [103] for storing key-value pairs of primitive data types.

2.5.2 RQ₂: Which sequences of Android API calls are the most energy-greedy?

We found 642 and 319 different patterns with length two and three respectively in the considered set of apps. For the former (length = 2) there were 15 negative outliers, while for the latter (length = 3) we found eight outliers. As already observed when analyzing single API methods, also in this case energy-greedy patterns mainly fall into two categories: *GUI Manipulation* and *Database*. Note that this holds for both patterns of length two and three. The complete list of patterns and energy measurements can be found in the online appendix ⁷.

The pattern `<Activity.setContentView(int); Activity.findViewById(int); View.setVisibility(int)>` is the most energy-greedy sequence we found, with an average consumption of 0.20 Joules. It is used for setting the content view of an `Activity` and make it visible to the user. Most of the energy consumption for this pattern is due to the `Activity.findViewById(int)` method (also present among the 131 greedy-energy methods identified in **RQ₁**). This method is in charge of finding a `View` basic block identified by the *id* passed as a parameter. The problem is that the views structure for an

⁷<http://bit.ly/1fCsjwz>

Android app is stored in XML files (known as layout files) that can easily reach very large size. Thus, in order to find a specific view given its *id*, all the layout files must be iterated through, which is a computationally expensive operation resulting in high-energy consumption as well. Although the consumed energy depends on the amount of views declared in the layout files, some developers recommend to save a global private instance of every visual component that will be used instead of calling `Activity.findViewById` often, as described by an Android Framework Engineer in the Google Forum [27]:

How expensive findViewById? [...] Actually it's not nearly so smart – it is just a traversal through the view hierarchy until it finds a matching id [...] As with all things, you should avoid doing this repeatedly if you don't need to (keep the thing you find in a variable so you don't have to look it up again).

Curiously, this is exactly the opposite of what we found in several apps, like a birthday reminder app showing an app method with more than 50 calls to the `Activity.findViewById(int)` API. We looked for the energy consumed by those calls and we found that the 57 executions of `findViewById` can consume up to 0.22 J, which represents $8 \cdot 10^{-3}\%$ of the battery of a Nexus 4 smart phone. Note that this is the consumption caused by the execution of just one method, and that is given the fact that multicore CPUs implemented in modern smart phones are able to execute millions of methods in less than a second time.

Among the energy-greedy patterns related to database operations, interesting ones are those represented by sequences of calls to the `SQLiteDatabase.execSQL(String)` method, especially long when the statements are used to create/drop database elements. We found some examples, such as one method in an *Education* app in which 26 `execSQL(String)` invocations are performed, leading to battery consumption up to $3 \cdot 10^{-3}\%$ for a single execution of a method.

2.6 How Developers Document and Design Test Cases for Android Apps

While common wisdom and best practices suggest that test cases should be derived from requirements artifacts, to the best of our knowledge, previous studies (about mobile testing) have focused on the challenges and tools used for testing but without analyzing the details of the strategies used by mobile developers for designing manual and automated test cases [143, 149]⁸. In this section we summarize the findings of a survey with contributors of open source Android apps hosted on GitHub. The goal of the study was to understand testing practices of mobile developers from the perspective of how test cases are designed, what are the testing strategies, why manual testing is preferred over automated (as suggested by previous work [143, 149]), and what information should be included in an automatically generated test case. In addition, the survey asked mobile developers about the usage of widely used techniques in the research community such as random testing and coverage analysis.

In the following we summarize the research questions and corresponding results. The publication supporting the results is under submission to the journal IEEE Transactions on Software Engineering.

2.6.1 RQ₁: What are the strategies used by mobile developers to design test cases?

Mobile developers heavily rely on usage models to document and design test cases. First, requirements are documented using different artifacts such as feature lists, informal natural language descriptions, user stories and use cases; only 6 out of 102 participants reported

⁸Kochhar *et al.* [149] investigated also with a survey with 83 open source Android developers the tools they use and challenges they face while testing Android apps. It is worth noting that our survey also includes a question concerning the tools used for automated testing. Kochhar *et al.* [149] list a reduced set of tools (i.e., 10), however, we complement their findings with a list of 55 tools used by our surveyed participants. We also complement their findings with a specific question designed to understand experiences and issues of mobile developers when using random testing tools

they do not use any artifacts to document requirements. Second, the surveyed participants mostly rely on manual testing and unit testing for their testing strategies. The rationale provided by the participants for their preference and effort dedication to manual testing is supported by several reasons such as (i) changing requirements, (ii) lack of time for testing and process decisions, (iii) size of the apps, (iv) lack of knowledge of the tools and techniques, (v) usability and learning curve of available tools, and (vi) the cost of maintaining automated testing artifacts. Finally, the surveyed developers mostly focus on the usage model (i.e., use cases and features) to design test cases, therefore, they use single model or a combination of use cases/features as the target for their test cases.

2.6.2 RQ₂: What are the preferences of mobile developers for test cases generated automatically?

Automatically generated test cases in natural language or expressed using automated testing APIs (e.g., Robotium or Espresso) are preferred by the participants. This suggests a preference for high-level languages instead of low level events as the ones that can be expressed using ADB input commands. In addition, the surveyed developers prefer to have test cases that include expected outputs and reproduction steps organized/grouped by use cases or features.

2.6.3 RQ₃: What tools are used by mobile developers for automated testing?

The surveyed participants rely on a diverse set of tools for supporting automated testing of mobile apps. In particular, 55 different apps were reported showing a preference for APIs such as JUnit, Robolectric and Robotium. Compared to the study by Kochhar *et al.* [149], we report a larger set of apps answered by a larger set of participants. However, both studies agree on listing JUnit, Robolectric and Robotium as part of the top-4 used tools. Record & replay, and random testing tools are used only by few participants. In the case of random testing tools, few participants claimed some benefits such as stress testing,

execution/discovery of corner cases, and execution of events that are hard for humans. However, some of the reasons that are an impediment for a wide usage of random testing tools (e.g., Monkey) is the lack of expressiveness of the generated event streams.

2.6.4 RQ₄: Do mobile developers consider code coverage as a useful metric for evaluating test cases effectiveness?

Code coverage is neither used nor considered as useful for measuring the quality of test cases by 63.73% (i.e., 65 out of 102) of the surveyed participants. Some of the reasons explaining the lack of confidence in the metric is that they prefer fault-detection or feature-coverage capabilities of test cases as a measure of quality, or they prefer to measure test cases quality by performing code reviews. On the other side, some participants consider that code measure is a good metric because it helps to identify parts of the code that are not tested.

2.7 Discussion

The empirical studies provided insights into the key challenges of mobile developers, preferred practices, and tools required by practitioners to support evolution and maintenance. A main cross-cutting concern, over most of the studies, is related to the high dependency on dynamic analysis for evolution/maintenance tasks: execution of Android apps on a diverse set of platforms (i.e., combination of device and OS) is a mandatory task to (i) assure the quality of the apps, (ii) validate/acknowledge crowdsourced requirements and bugs, and (iii) test non-functional requirements such as performance and energy consumption. Moreover, dynamic analysis of Android apps has to follow the constraints and nature imposed by the mobile development lifecycle; in that sense, tasks relying on dynamic analysis (e.g., testing) should be efficient and effective (i) under the pressure for continuous delivery, (ii) with an unstable API that is a threat to the success of the apps, (iii) with a constantly changing set of target devices, and (iv) with a contextual model (i.e., events

and adversarial conditions) that imposes a complex set of inputs. Therefore, there is a clear need for a versatile infrastructure supporting dynamic analysis of mobile apps, and a need for approaches that support evolution and maintenance of Android apps that rely on such an infrastructure. The next Chapter describes an infrastructure designed to support large-scale execution of mobile apps, which follows design drivers such as the open-closed principle, vertical/horizontal scalability, and loose coupling between an Execution Engine and diverse clients supporting different tasks.

2.8 Bibliographical Notes

The empirical studies summarized in this Chapter were performed in collaboration with other members of the SEMERU group at William and Mary and researchers from the University of Sannio, the University of Bozen-Bolzano, and the University of Molise:

- Palomba, F., **Linares-Vásquez, M.**, Bavota, G., Oliveto, R., Di Penta, M., Poshyvanyk, D., and De Lucia, A., “User Reviews Matter! Tracking Crowdsourced Reviews to Support Evolution of Successful Apps”, in Proceedings of 31st IEEE International Conference on Software Maintenance and Evolution (ICSME’15), Bremen, Germany, Sept. 29 - Oct. 1, 2015, pp. 291-300 (22% acceptance ratio)
- **Linares-Vásquez, M.**, Vendome, C., Luo, Q., and Poshyvanyk, D., “How Developers Detect and Fix Performance Bottlenecks in Android Apps”, in Proceedings of 31st IEEE International Conference on Software Maintenance and Evolution (ICSME’15), Industry Track, Bremen, Germany, Sept. 29 - Oct. 1, 2015, pp. 352-361 (39% acceptance ratio)
- **Linares-Vásquez, M.**, Vendome, C., Tufano, M., and Poshyvanyk, D., “How Developers Micro-Optimize Android Apps”, currently under review.
- Bavota, G., **Linares-Vásquez, M.**, Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D., “The Impact of API Change- and Fault-Proneness on the User

Ratings of Android Apps ”, IEEE Transactions on Software Engineering (TSE), vol 41, no 4, April 2015, pp. 384-407

- **Linares-Vásquez, M.**, Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D., “Mining Energy-Greedy API Usage Patterns in Android Apps: an Empirical Study”, in Proceedings of 11th IEEE Working Conference on Mining Software Repositories (MSR’14), Hyderabad, India, May 31- June 1, 2014, pp. 2-11 (34% acceptance ratio)
- **Linares-Vásquez, M.**, Bavota, G., Di Penta, M., Oliveto, R., and Poshyvanyk, D., “How do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK”, in Proceedings of 22nd IEEE International Conference on Program Comprehension (ICPC’14), Hyderabad, India, June 2-3, 2014, pp. 83-94 (48% acceptance ratio)
- **Linares-Vásquez, M.**, Holtzhauer, A., Bernal-Cárdenas, C., and Poshyvanyk, D., “Revisiting Android Reuse Studies in the Context of Code Obfuscation and Library Usages”, in Proceedings of 11th IEEE Working Conference on Mining Software Repositories (MSR’14), Hyderabad, India, May 31- June 1, 2014, pp. 242-251 (34% acceptance ratio)
- **Linares-Vásquez, M.**, Bavota, G., Bernal-Cárdenas, C., Di Penta, M., Oliveto, R., and Poshyvanyk, D., “API Change and Fault Proneness: A Threat to the Success of Android Apps”, in Proceedings of 9th Joint Meeting of the European Software Engineering Conference and the 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE’13), Saint Petersburg, Russia, August 18-26, 2013, pp. 477-487 (20% acceptance ratio)

Chapter 3

Enabling Large-Scale Execution and Testing of Android Apps

In the last few years, the industrial community has experienced a tremendous gravitation toward a new economy driven by mobile devices, apps, and other corresponding services/products. As a response to this phenomena, researchers in software engineering have contributed various tools and approaches aimed at supporting essential tasks in the development process of mobile apps including ensuring security [115, 180, 65, 213], testing [55, 124, 178, 175, 81, 171, 195], improving energy consumption [123, 152, 166, 235, 154, 167], and crowdsourcing requirements [120, 203, 200, 201], among others. However, despite ongoing efforts from both academic and industrial communities, there are still outstanding challenges related to dynamic analysis and execution of Android apps that highly impact mobile development and maintenance.

Industrial mobile developers and testers face the following challenges: (i) continuous pressure from the market for frequent releases [133, 142], (ii) platform fragmentation at device and OS levels [143, 146], and (iii) rapid platform/library evolution and API instability [160, 168, 185, 69]. Therefore, continuous validation of mobile apps on a large set of device configurations is a “must-have” in the development process to ensure the quality of mobile apps. However, this must be enabled within the constraints of tight

release schedules and limited developer and hardware resources. In academic research, the large number of apps available in markets such as Google Play [24] and F-droid [21], and the considerable volume of recently published papers that utilize dynamic analysis or perform execution of Android apps with large datasets, are pushing the field to increase the number and complexity of apps used in the studies. When taking into consideration the challenges in both communities (industry and academia), an infrastructure for large-scale dynamic analysis of Android apps is highly desirable.

Large-scale execution of mobile apps in the context of this paper can be defined as the concurrent execution of a high number of varying configurations of the following attributes: (i) devices (either as a physical or virtual device farm, further defined by hardware constraints such as screen size, amount of memory, etc.), (ii) platform/OS versions, (iii) apps (in terms of both the diversity of supported apps, and the ability to run multiple versions of the same application), and (iv) execution scenarios or test cases (either manually written or automatically generated; utilizing one of several scripting tools/languages). The most typical use case for such a framework is conducting mobile testing by running a suite of manual or automated GUI-based scenarios on given versions of an application on a multitude of device configurations for verification purposes, typically before a release. However, this type of large-scale dynamic analysis can also support other tasks such as energy optimization, performance analysis, and security analysis.

In order to be practical such an infrastructure must meet two *key* requirements (i) it should be easily adoptable and highly configurable by both researchers and developers via relying on open-source technologies, and (ii) it should not require a large investment of expensive hardware/software components. Unsurprisingly, there are several challenges and issues that arise when enabling such a framework within these requirements. For instance, currently the tools provided by Google (e.g., emulators) and the Android SDK/Framework utilities exhibit several issues that inhibit automated, parallel execution tasks on a large number of concurrently running devices/emulators. Furthermore, emulator instability and inappropriate management of inter-event delays are exacerbated when a

large set of emulators are executed in parallel. Additionally, there are also open problems specific to automated mobile testing, such as the generation of test oracles.

Previous studies have proposed approaches involving large-scale, parallel analysis of Android apps [74, 179, 177, 220, 70, 238, 186]; however, the papers neither report technological challenges when enabling their frameworks nor offer clear solutions or guidelines that developers and researchers should follow when implementing a large parallel execution framework to meet their needs. Thus, there is currently a glaring lack of best practices and guidelines for large-scale execution and testing of Android apps. Recently, as part of a research effort and industrial collaboration with a large telecommunication company (Huawei), we have developed and assembled an infrastructure for enabling large-scale execution and testing of Android apps that is easily adoptable and relies only on open source components. In the course of this development, several technical challenges were met and overcome. It is likely that many of these challenges will continue to go unaddressed by emerging tools or documentation, and while commercial tools may solve some issues, the research community will lack a critical roadmap for implementing a framework that will enable further work related to dynamic execution of applications. Thus, a clear discussion of the specific challenges encountered, the developed solutions, and current open questions is required in order to provide researchers and practitioners with a clear understanding of the architecture and guidelines we present.

The purpose with this Chapter is two-fold: (i) provide researchers and developers with a list of guidelines and best practices for enabling large-scale execution and testing of Android apps, and (ii) present a infrastructure that can be instantiated/extended to allow large execution of Android apps with multiple purposes. Therefore, the contributions of this Chapter are as the following:

- A taxonomy of issues that developers, testers, and researchers experience when enabling large-scale execution and testing of Android apps (Sec. 3.1);
- Specific solutions for some of the issues presented in the taxonomy (Sec. 3.2);

- Guidelines for designing an infrastructure that will support large-scale execution and testing of Android apps (Sec. 3.3);
- A comprehensive online appendix with recipes and pertinent guidelines [20].

3.1 A Taxonomy of Essential and Accidental Issues

Software difficulties, in the strict sense of the language used by Brooks in his famous “No Silver Bullet” and [72] and “The Mythical Man-Month” collection of essays[73], are categorized into “*essence* and *accidents*”¹. The former, i.e., *essential* difficulties, represent challenges that stem from the inherent nature of software such as complexity, changeability, and invisibility; the latter, i.e., *accidental* difficulties, refer to issues that are an artifact of the process or the tools utilized for some software development goal. Typically, *accidental* challenges can be solved, however, only partial solutions exist (if any) for *essential* challenges.

In this chapter, we follow the same dichotomy to describe the set of issues we have experienced when enabling large-scale execution and testing of Android apps. Consequently, we have found *essential issues/challenges* that are inherent to large-scale execution and testing of Android apps, and *accidental issues/challenges* that are introduced by the tools currently available for Android development. An overview and at-a-glance information for these issues and their corresponding solutions (presented in detail in Section 3.1) can be found in Table 3.1. When developing the infrastructure proposed in this paper we had testing as our main goal, and thus we center our discussion of issues and challenges around this application. However, several of the challenges discussed are not specific to testing and can impact other activities that require dynamic analysis.

¹The distinction between essential and accidental properties is not an exclusive concept of Computer Science; the concepts were first introduced by Aristotle in “Metaphysics”, and later applied by Brooks in the context of SE.

3.1.1 Essential Challenges for Automated GUI-based Execution

Dynamic analysis and testing of Android apps generally follows GUI-based strategies because of the GUI-centric design and event-driven nature of mobile apps. In this sense, essential issues in execution and testing of apps are mostly related to (i) efficient and effective verification of the expected results for the GUI and app states, (ii) execution strategies that account for proper timing and coordination of event execution, and (iii) the impact of external dependencies. We highlight these essential issues in detail.

3.1.1.1 Test Oracles

Current “in-the-wild” practices for mobile testing heavily rely on the developer/tester to manually verify expected results [143, 149, 169], or require the programmatic definition of oracles via assertions and test automation APIs such as JUnit [30], Espresso [112], or Robotium [43]. In the context of automated approaches proposed by researchers, the most common execution strategies are GUI ripping and systematic exploration with the intention of discovering crashes and errors that are reported as exceptions [132, 81, 195, 133, 179, 177]; other approaches focus on automatic generation of test cases as sequences of GUI events but without the corresponding oracle [171, 243, 159], which requires manual definition or verification. Additionally, expected transitions between windows and GUI screenshots have been proposed as potential solutions for automatic detection of GUI errors in Android apps [159].

Using APIs for automated testing (e.g., JUnit, Espresso, Robotium) is very convenient in terms of automatic execution but is also expensive in the sense that they require a developer/tester to write and maintain the tests and corresponding oracle(s). State-of-the-art rippers are very useful for detecting crashes and unexpected errors, but lack capabilities for identifying errors occurring in the GUI; e.g., a GUI component expected to be in a window was not activated due to a bug in presentation or business logic where no exception was thrown (or the exception was encapsulated or handled with an empty catch).

Approaches that rely on screenshot comparison (i.e., expected GUIs vs. resulting GUIs) are closer to the goal of detecting such GUI-level errors, however, require image similarity thresholds, which could vary widely for different apps and analyzed windows. The GUI comparison and similarity threshold definition can be impacted by several conditions exhibited under different devices and settings [159]. In our experience we have found that the accuracy of matching GUI oracles (i.e., using images as test oracles) can be negatively affected when comparing images (i.e., a GUI oracle against collected GUIs during tests) of different screen sizes and when the test is performed on a different device or orientation (e.g., horizontal vs. vertical). Additionally, GUI hierarchies are typically rendered differently (e.g., hiding/showing some components) on smaller displays because of responsive design capabilities provided by the Android framework or defined programmatically by the developer, which can cause further issues in matching screenshot pairs on devices with diverse configurations. Color palettes can be modified drastically in a target device according to the theme defined by the user or the language setting, which can cause false positives when using images as test oracles. Moreover, GUI components for dynamic content, such as date/time pickers (see Fig. 3.1-a) or map visualizations, can differ depending on the variables linked to the component (e.g., current date, location); for instance, collecting a GUI oracle on a `CalendarView` in an app on February 29th, and then running the test cases ten days later would result in a failing test. We discuss a custom solution to the problem of test oracles in Section 3.2.1.

3.1.1.2 Event Coordination

Automated tests for Android apps can be described as sequences of events that exercise the application under test by means of GUI events, services or intent invocations, and contextual events that trigger execution of specific listeners. An important factor to be considered in automated tests is the delay between events (a.k.a., inter-arrival delay) to avoid phenomena such as early dropping of events [159, 158] and test flakiness. However, response time for GUI output between events can not be totally controlled (even in the

Table 3.1: Overview of the Essential and Accidental Challenges Discussed (Relevant section numbers are indicated next to each challenge or solution, as well as corresponding links to detailed solutions in our online appendix [20] are given as citations. Text in blue represent partial solutions, green fully or mostly solved solutions, and red challenges that are currently unsolved.)

Essential Challenges	Solutions	Accidental Challenges	Solutions
Test Oracles: (3.1.1.1) How does one design an extensible GUI-based oracle that can be used to verify expected application behavior across different device configurations?	(3.2.1) Use a hash of the xml representation of an application’s GUI, which is more extensible between devices compared to image based heuristics.[49]	Application Data and Cold Starts: (3.1.2.1) Android applications can persist data using several different options. How can one ensure a sterile or desired testing environment within the context of large scale testing?	(3.2.3) Utilize the built-in adb commands for clearing application data for apps that store it internally, and identify and delete external data that might be saved to the filesystem. [48]
Event Coordination: (3.1.1.2) In automated GUI-based testing, deciding the proper timing of firing events on a device is challenging.	(3.2.2) Query the Android window server to determine when application animations are occurring in order to infer when the state of the GUI is idle, and thus it is safe to send new commands. [46]	Bugs in Framework Utilities: (3.1.2.2) There exist certain limitations with core Android Frameworks popular with GUI-testing, such as the inability to obtain a snapshot of GUI-information for dynamically updating screens.	(3.2.4) We modified the source code of the Android uiautomator framework in order to enable the efficient collection of the GUI-hierarchy from apps with dynamically changing screens. [49]
External Features: (3.1.1.3) How can external features such as notifications be properly incorporated into large-scale testing of mobile apps?	Researchers should consider using actual usage data from the field in order to inform the type and frequency of external features such as notifications.	Bugs and Limitations in the SDK Tools: (3.1.2.3) There are limitations inherent to Android developer tools, such as adb, that make large-scale parallel execution and testing difficult.	(3.2.5) Use multiple adb servers, with no more than 15 devices per server, in order to properly interface with a large number of connected physical or virtual devices. [47]
External Dependencies: (3.1.1.4) Many mobile apps use external servers or databases in order to send, store, and receive information. How can these dependencies be appropriately incorporated into large-scale testing?	While this issue is currently unsolved, future work could focus on modeling the behavior of these external dependencies, based on usage data.	Challenges Scaling Concurrent Virtual Devices: (3.1.2.4) Android emulators were not designed to scale well on modest hardware, thus, other types of virtual devices are required to enable large scale execution and testing.	(3.2.6) Use Android Virtual Machines based on the android-x86 project with a hypervisor such as VirtualBox in order to enable a configurable and scalable virtual device farm. [47]

case of end-to-end tests with hermetic servers) due to the asynchronous nature of some events in Android and processing delegated to external services. Consequently, “waiting” times have to be hard-coded in scripts/code for automated execution/testing of apps , when the tool/API used for automatic execution is not idle-time aware. For instance, if after executing an event (e.g., click on an OK button), the response from the app might take more time than expected, in particular, if the response depends on an external service or a lengthy operation, the next scheduled event in a script (or test case) can be executed, but the app is may not be ready yet to process it. Therefore, the scheduled event is not executed, which can lead to unexpected test case results, false declaration of the test case as failed [159, 158], or execution of testing sequences that are not designed by developers/testers.

This issue is often aggravated when running several test cases on multiple virtual de-

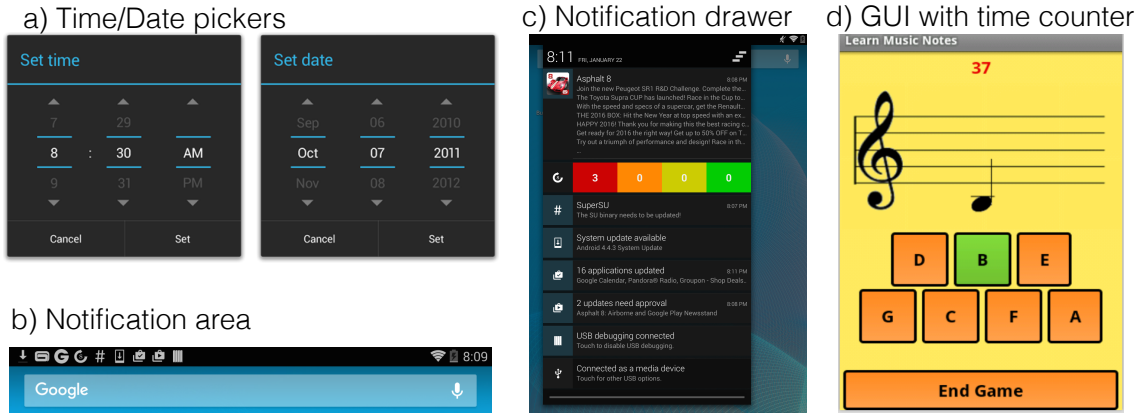


Figure 3.1: Examples of GUI components in Android apps that are problematic for automated test cases.

vices (i.e., emulators) on the same machine, even with GUIs that do not involve lengthy operations or calls to external services. Depending on the machine specifications, the processing (and corresponding time) of an event in a virtual device can depend on the scheduling mechanism in the host machine, or the general GUI responsiveness of the virtual devices. For example, on a moderately capable desktop workstation, with a 2.8GHz Intel Xeon E-5-1603 v3 CPU with 4 cores, 128 GB of RAM, and an Nvidia Quadro K420 GPU, we have experienced longer than expected wait times during automated test-case execution with multiple emulators. Specifically, we experienced issues with a hardcoded delay-time of 15 seconds between events (e.g., some event transition animations took longer than 15 seconds, so the following event in the test case was executed during the transition causing unexpected behavior). When considering existing APIs for automated testing (i.e., JUnit, Robotium, Robolectric, Espresso), only Espresso is idle-time aware. The execution of a GUI event with Espresso waits for the app to be idle after the event; the source code of the action methods in Espresso includes a call to `UiControllerImpl.loopMainThreadUntilIdle()` [19], which internally waits for the following conditions with a *do-while* loop: (i) there is no task running, (ii) there is no `AsyncTask` running, or (iii) the resources are in an *idle-state*. However, this solution is Espresso-dependent, and can not be used with other APIs. We discuss a general custom solution to hard-coded times in Section 3.2.2, which is not coupled to any specific API.

3.1.1.3 External Features

Android apps can provide “external features” (i.e., outside of apps’ GUIs) via notifications, timers, and widgets. For instance, notifications are messages that can be displayed outside of an app at any moment (even when the app is not in foreground) [108]. A notification appears initially as a banner (or icon) in the notification area at the top of the screen, and the details can be displayed by opening the notification drawer. The banner/icon, detail, and conditions for the notification depend on the application logic, while the location of the icon (Fig. 3.1-b) and the items in the notification area/drawer (Fig. 3.1-c) depend on the existence of other notifications in the device. Therefore, testing the opening and correct displaying of a notification automatically requires approaches that are able to identify the location of the icons/details in the notification area/drawer.

Moreover, automatically testing notifications that are time-related is a difficult task; for example, assume a test case was recorded or programmed to show a notification at 4:05pm; the test case was collected at 4:00 pm, thus, a test case will wait for at least 5 minutes before opening the notification. When using a Record & Replay tool like Monkey Runner [2] or RERAN [100], the timing between the sequences of actions to set a date picker will be the same, however, the value chosen may differ depending on the specific time the test is run and the corresponding default value in the date picker. This specific scenario can be solved by relying on the `PickerActions` [105] provided by the Espresso API and automatic collection of current time using the Java API. However, if the notification time can not be estimated automatically because the notification is generated by a service running in the device or by an external service, the test case might include an infinite loop, or time-out-bounded loop that continuously checks if the alert is in the notification area. These types of external features are better candidates for manual testing, however, there is still effort required in assuring that the feature would work well on different devices, OS versions, and under varying contextual conditions. This is an open issue for which no solutions exist yet.

3.1.1.4 External Dependencies

Complex mobile apps delegate tasks to back-end servers (e.g., authentication servers) or rely on mobile cloud architectures [53]. These external dependencies pose a threat to the quality of mobile apps, and a challenge for automated testing. Several strategies have been proposed by the industry to test apps that require back-end servers [106]. For instance, replacing real back-end servers or services with mock/fake servers or mock/fake network implementations running on the same machine that host the virtual device (with an app under test) reduce the risk of test flakiness and testing time, when comparing to end-to-end tests [106]. However, this type of strategy imposes overhead on virtual devices, or machines hosting the “fake” servers (when it is the same as running the virtual devices), which can be an impediment to large-scale execution and testing when the developers/researchers do not have access to elastic on-demand cloud-based infrastructures.

3.1.2 Accidental Challenges for Automated GUI-based Execution

3.1.2.1 Application Data and Cold Starts

Android apps can persist data using different options, such as shared preferences, internal/external storage, and SQLite databases [110]. Depending on the option selected by the developer, application data can be persistent even after killing or uninstalling the app. Application data in shared preferences, internal storage, cache, and SQLite databases are private and this data is removed when the application is uninstalled; however, data persisted using these options is persistent across different sessions. Whereas data saved in external storage (e.g., sdcard), is available in the device even after uninstalling the app. In order to ensure a consistent and “sterile” execution environment, particularly for testing-related activities with repetitive executions, it is imperative that approaches for automated execution are aware of internal and external application data. Therefore, it is possible that some apps, which are reinstalled (e.g., after running a test case), will use the data in external storage. In some cases, the behavior of an app depends on the

application data and whether it is the first time the app is launched after installation on a device (i.e., a cold start); splash/welcome screens, dialogs for downloading data from a back-end server, or text bubbles and semi-transparent overlays with help/demo purposes, are displayed by some apps when users interact with certain features for the first time (Fig. 3.2).

Imagine a scenario in which automated test cases are recorded/programmed without assuming cold-starts for an app *A*, but each test case is executed with cold-start; also, assume that app *A* has semi-transparent overlays on each activity that appear only the first time an Activity is started (see Fig. 3.2-f). In this scenario the recorded/programmed test cases will not consider the semi-transparent overlays that disappear by clicking on it, therefore, some GUI events in the test cases expected to be executed on GUI components will be “wasted” when clicking on the overlays. Another example scenario is one in which the test cases consider cold-starts, however, each test case is executed without removing application data or uninstalling the app. Consequently, cold starts, and app data can drive unexpected executions of test cases and thus false positives/negatives (in terms of passed/failed test cases)². We describe a technique for cleaning app data in Section 3.2.3, which can be used with automated executions scenarios.

3.1.2.2 Bugs in the Framework Utilities

The Android OS running on devices includes a layer known as the Application Framework (AF), which contains a set of system services (a.k.a., managers) that allow Android apps to access basic properties of a device, such as activities, resources, and window management [92]. The AF also includes a set of utilities that can be invoked remotely (i.e., via Android Debug Bridge [102]) for debugging and testing purposes. These utilities are located in the `/system/` folder of the Android OS and provide the core functionality to SDK tools, such as the `hierarchyviewer` and `uiautomatorviewer`. Both tools (and the

²Fuzz testing tools like Monkey can get stuck on welcome screens/dialogs. However, GUI rippers do not experience problems because the exploration is based on the current snapshots, therefore, GUI rippers are aware of the GUI components in the welcome screens/dialogs

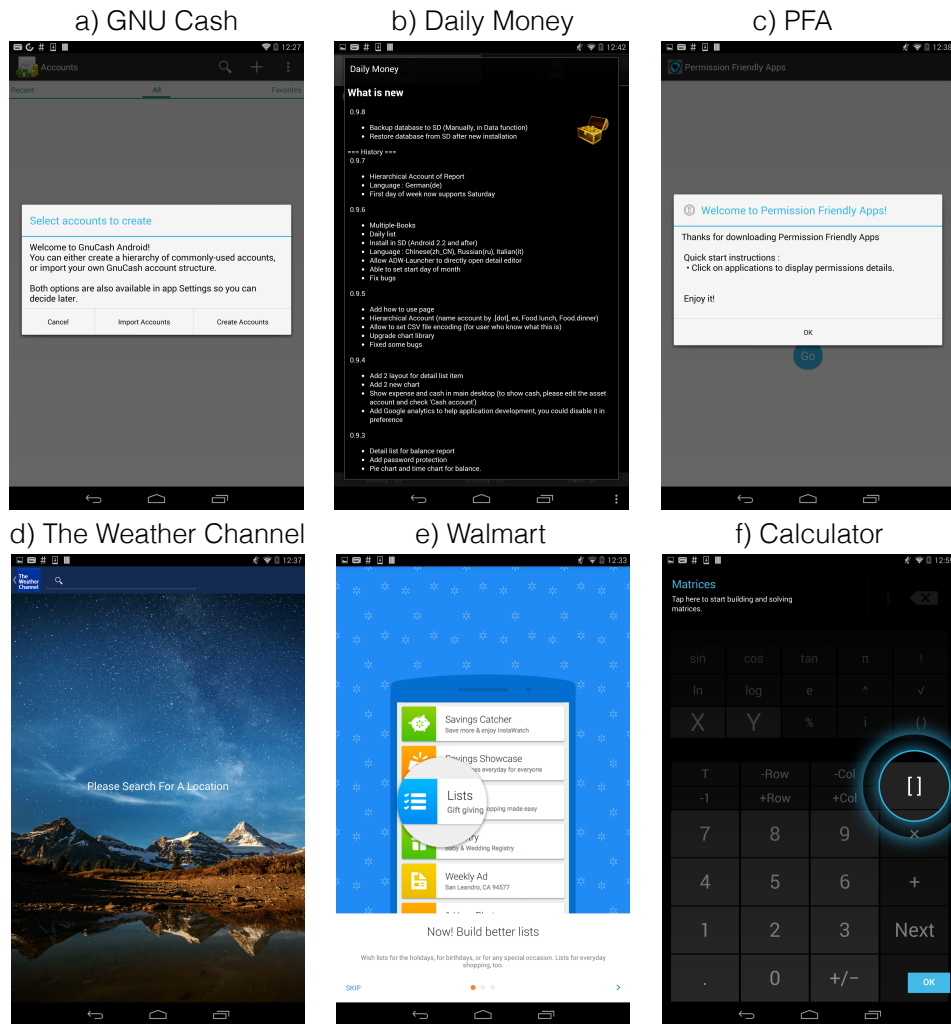


Figure 3.2: Example of splash/welcome screens, dialogs, and semi-transparent overlays that appear in Android apps during cold-starts or the first time a feature is used.

corresponding AF utilities) have been widely used in the research community (including our own projects) to extract snapshots (i.e., components, locations, properties, and layout) of the GUIs rendered in devices. As a matter of fact, utilizing such tools is the only way to collect accurate GUI information dynamically without instrumenting apps or the OS. Consequently, because the AF is the only layer that can be accessed remotely for dynamic analysis purposes, bugs in the AF utilities directly impact developers/researchers infrastructures for execution and testing of Android apps. For example, we initially used the `hierarchyviewer` utility from the AF, however, we found the utility to be too slow to

be practical, incurring a large overhead when snapshots are collected after each event is executed in a large test suite (automatically executed). After further research, we found that the AF `uiautomator` utility operates much faster, thus reducing the time for GUI snapshot collection. However, there are two issues that should be considered when using `uiautomator`: (i) in the case of dialogs, the snapshot does contain information of the GUI behind the dialog (conversely to `hierarchyviewer` that includes all the information); and (ii) GUIs with changing components or animations generate an "ERROR: could not get idle state"³ message and the snapshot is not generated (this bug does not show up with `hierarchyviewer`). The latter issue shows up in cases when there are components that are constantly changing (e.g., a stopwatch) like in the “One minute test” window of the Learn Music Notes app that has a seconds (red) countdown counter at the top (Fig. 3.1-d). We describe our fix for the issue in Section 3.2.4.

3.1.2.3 Bugs and Limitations in the SDK Tools

IDEs for Android development, such as Android Studio or Eclipse ADT, rely on the Android SDK, which contains the APIs, emulators, and tools for compiling, packaging, and debugging Android apps. Also, the SDK can be used as a stand-alone suite by invoking the tools via command line. One of the most used tools by developers and researchers, included in the SDK is the Android Debug Bridge (`adb`) [102]. It is a command-line tool that allows for remote control and communication to a device/emulator from the machine hosting the SDK. The `adb` operates in a client-server style in the host machine and talks to the `adb` daemon running on each device/emulator. An `adb` server running on a machine with the SDK is able to handle up to 15 different real or virtual devices [102] when using default ports. In our testing experience, `adb` has a tendency to become unstable (e.g., dropped commands, devices disconnecting without warning) when the maximum number of devices for a particular server is exceeded. In Section 3.2.5 we describe a procedure to

³This error is also apparent in the `uiautomatorviewer` tool in the SDK, which displays the following error: “Error obtaining UI hierarchy. Reason:Error while obtaining UI hierarchy XML file: com.android.ddmlib.SyncException: Remote object doesn't exist!”.

overcome this adb limitation.

3.1.2.4 Challenges Scaling Concurrent Virtual Devices

With the continuing fragmentation of the Android Platform and the continued release of devices from different manufactures, it is clear that developers need automated methods to support testing of their apps on several device configurations. An economically feasible way to accomplish this goal is to use Android Virtual devices (e.g., either Android emulators or virtual machines), as it does not require the upfront cost and maintenance of a physical device farm (i.e., a large collection of physical Android devices). Furthermore, from a time perspective, it is desirable to enable concurrent automated testing using a “cloud” of emulators running simultaneously to reduce testing overhead. However, practical solutions for achieving this on in-house hardware with modest specifications have not been effectively discussed in the context of automated testing research, and open source solutions are not immediately available.

During the course of developing a custom large-scale automated framework for the parallelized execution and testing of Android apps, we have encountered several issues with Android Virtual Devices (AVDs). We define an AVD as any instantiation of a physical Android device in virtualized manner, e.g., either using emulated hardware, or fully virtualized. There are many different potential solutions for running a virtual device on a personal workstation or sever, however, there are several issues that could serve as roadblocks to developing a large-scale testing infrastructure of which researchers and practitioners should be made aware. Currently, the most popular current solutions for running virtual Android devices are the following: (i) the standard Android emulator officially supported by Google [36], (ii) the Genymotion [23] emulator, and (iii) Virtual Machines based on the `androidx86` project [3]. In this section we further discuss problems encountered with the Google and Genymotion emulators in terms of scalability.

When designing a scalable framework for supporting multiple concurrent instances of AVDs, it may seem that starting with the default Android emulator is the best course of

action, however, this is not the case, as several problems arise that limit this option. First, in order to properly utilize Google’s emulator at screen sizes and pixel densities representative of modern physical devices, the host machine must be equipped with a moderately powerful GPU at a bare minimum. This severely limits the potential usefulness of these tools on headless servers which do not typically contain the graphics cards required for running these AVDs; even major IaaS providers like Linode [31] do not provide hardware nodes with GPUs. We found that the maximum supported resolution of Google’s emulators on hosts without graphics cards is that of 1280x786 (or that of the Nexus 4 phone). Additionally, these types of emulators require graphics acceleration in order to function at an acceptable speed in terms of boot times and GUI-responsiveness. While it is clear that a graphics card is important for the proper operation of these machines, many development workstations typically include at least a moderate GPU, and several emulators could conceivably be instantiated on such a workstation.

However, three additional limiting factors emerge when attempting to run more than a few emulators on such a machine: (i) the amount of RAM available to the host machine, (ii) whether the host CPU supports hardware acceleration, and (iii) the specifications and computing potential of the available GPU. System RAM is perhaps the most important limiting factor for such emulators, as large amounts of memory are still relatively expensive, and this hardware specification puts a hard cap on the number of devices that can be instantiated on given host machine. This is because at a minimum each emulator should be assigned no less than 500 Mb of RAM; on a machine where 8GB of memory are available, this means that a *maximum* of 16 emulators could be established, assuming all this memory would be *free* from the system (an optimistic assumption at best). As we explain later, this is the most important system component in our experience and thus should be the focus of building a machine to support parallel execution of AVDs.

Both the Google and Genymotion emulators rely on CPU hardware virtualization (through either qemu, kvm, Intel HAXM, or VirtualBox) in order to emulate the hardware at acceptable speeds. Therefore, this is a system requirement that must be taken

into consideration. This is of particular importance for those wishing to instantiate a parallel AVD framework on a cloud provider such as Linode[31] or Amazon EC2 [1], as these cloud providers typically offer virtualized environments, meaning that unless the provider enables paravirtualization, it is impossible to run these types of emulators on these providers without using a commercial solution such as the one by Ravello systems [42].

Finally, the computational ability of the GPU can also limit the number of parallel emulators running on a given host machine. When the GPU is maxed out by concurrent instances of emulators, new emulators launched will typically throw one of the following errors to the command line window that launched the emulator: “ERROR: Could not initialize OpenGL ES emulation” or “eglMakeCurrent failed”. Additionally, it should be noted that the standard Google emulator can not be executed concurrently with either Genymotion or an `androidx86` VM instantiated with VirtualBox. This is due to the fact that VirtualBox, which is also used by Genymotion, locks the CPU virtualization interface (e.g., `qemu`, `kvm`, `HAXM`) required by the Google emulators. We discuss the solutions that we have developed for concurrent execution of AVDs in Sections 3.2.6 and 3.3.

3.2 Proposed Solutions

In this section we outline custom solutions formulated in course of enabling large-scale, concurrent execution and testing of Android apps in a custom framework developed as part of our research efforts and collaborations with Huawei. Specifically, we aim to systematically provide either complete or partial solutions to the issues outlined in Section 3.1. We envision these solutions as constituting a set of guidelines, learned lessons, and best practices for academic research teams, small teams of industrial practitioners, and independent mobile developers and testers looking to enable a concurrent testing infrastructure with relatively modest hardware requirements. Table 3.1 gives an overview of these solutions complete with direct anchored links to tutorials and examples in our on-

line appendix [20]. The discussion of these solutions culminates in Section 3.3 where they are used to drive the design of our large-scale execution framework. However, larger industrial teams could also benefit from adopting and applying some of the technologies and techniques that we outline. Because of space limitations, we outline our solutions and refer readers to our online appendix for detailed technical discussions and tutorials for implementing these solutions [20].

3.2.1 Non-Image Based Generation of Oracles

The automated generation of test oracles is particularly challenging for automated mobile testing approaches. While approaches have been proposed for detecting and reporting crashes [133, 195], only a few approaches attempted to provide a solution for this oracle problem. Typically, these solutions rely on specific system circumstances [243] or expensive image-based analysis of GUI screenshots [159]. We consider that an ideal automatically generated test oracle for mobile testing should be (i) easy and fast to collect from a device or emulator, (ii) abstract enough to focus on the GUI behavior and state instead of specific elements of the GUI appearance (e.g., colors), and (iii) should not be coupled to a specific technology for the oracle generation (e.g., the approach by Lin *et al.* [159] requires an external camera, which can introduce a lot of noise). In the course of our work we found *xml dumps of GUI-hierarchies* to be suitable test oracles under proper testing conditions, and scalable when the comparison is done between hashes of the hierarchies instead of the entire hierarchy itself.

A GUI-hierarchy dump is an XML generated with the `hierarchyviewer` or `uiautomator` utilities from the AF (i.e., the utility runs on the device). The XML contains a hierarchy of nodes, in which each node is a GUI component or a container (e.g., `LinearLayout`). Each node contains information, such as the component type, text, location, id assigned by the developer (i.e., resource id), and boolean flags reporting whether the component is focused, clickable, selected, etc. This XML representation allows for easy manipulation and parsing that can be used to generate test oracles. For example, by hashing the GUI-

hierarchy dump (or selected information from the dump) of the GUI output (i.e., screens) as a response to each event in a test case, the concatenation of the test-hashes can serve as an oracle for that test case. Then, during the execution of the same test case, the GUI state can be extracted in the same way.

Formally, we define a test case tc_i , as a sequence of m events $e_0e_1e_3...e_m$. After executing an event e_j , a corresponding dump d_j is collected from the device/emulator. If a hash H_j is generated for each d_j , then, a test oracle O_i representing the GUI state when executing tc_i is the concatenation of $H_j, \forall j \in [1, m]$. Therefore, a test suite is a set of couples $\langle tc_i, O_i \rangle$. The O_i oracles can be collected automatically when replaying the test cases on (i) an app version A_0 used as a reference for the testing process, and (ii) the version of the app under testing A_t ; the O_i oracles collected with A_0 and A_t should be the same (assuming the GUI has not changed drastically from A_0 to A_t).

It is worth noting that the complete dump can be used to generate the hashes, however, the dump contains locations and the text of each component. Therefore, developers/testers should be careful when deciding what information from the dump is used to generate the hashes. For example, if the testing goal is to verify that each GUI has the expected components and hierarchy, then the hashes should be generated from the dumps but considering only the resource-id and component types attributes in the nodes (and following the hierarchy in the dump). If the purpose is to verify all the “texts” are rendered as expected, then the hashes should be generated from the dump considering the resource-id, component types, and texts. Additionally, generally speaking, such an oracle will only function correctly for deterministic applications, further work is required to develop a robust oracle for apps with varying degrees of non-determinism.

3.2.2 Avoiding Hard-Coded Inter-Delay Times

When automatically executing GUI or system events in Android apps, a critical property must be met: *after sending an event to the device, the next event to be executed should not be sent until the previous event has been received and processed.* This is usually

achieved using either a manually hardcoded or automatically inferred *wait-time* between GUI events. Some approaches rely on instrumentation of an app or the OS [133, 211] in order to hook directly into event-handlers to confirm when certain events have been “completed”. However, instrumentation has been shown to be fault-prone [195], and these approaches may not function properly (due to instrumentation overhead) when scaling to several virtual devices concurrently running on a single machine. An ideal solution to this problem for GUI-based automated testing should allow an Android device/emulator property to be queried, without the need for instrumentation, to check current *animation/idleness* status.

In our work we found a solution by utilizing a property in the `adb shell dumpsys -a` command to query the device about the current state of the animation transitions on the screen. Since we are interested in the foreground app’s transition state at the time of the running the command, we read the value of the `mAppTransitionState` property returned by the command. This property has one of the four following states [12, 15]: (i) *App State Ready*, (ii) *App State Idle*, (iii) *App State Timeout*, and (iv) *App State Running*. Further information on these states, as well as a detailed technical discussion of this solution can be found in our appendix [20].

The Window Manager can be queried directly after a GUI-input event has been sent to the device. This will typically result in either 1) *App State Ready* or 2) *App State Idle* being returned. If State 1 is returned, it means the app is not ready yet to process another event, therefore, the device should be polled until State 2 is observed. Once State 2 has been observed, the Window Manager has indicated that the currently displayed GUI is idle and that it is safe to continue with the next GUI input command. While this methodology offers a robust solution to the problem of non-deterministic animation delay in automated mobile testing, it should be noted that this is applied in the context of the GUI only, and does not guarantee contextual events (e.g., network activity) have completed on the current screen. However, this solution offers an effective method of determining the state of app animation transitions without the need for expensive instrumentation or a specific

API because the `dumpsys` command can be invoked from any code that includes a method for invoking console commands.

3.2.3 Cleaning Data for Cold-Starts

One often overlooked problem regarding the testing of Android apps is ensuring the same execution environment at the beginning of each test run. As described in Section 3.1.2, this can be a more difficult practice than researchers and testers expect. “Stale” app data can cause problems with automated testing approaches; when performing automated GUI-testing of Android apps, a desired feature of an automated execution/testing framework is *to ensure that the initial application state is the same (e.g., clean) before each test-case is executed*. This guarantees that sequences of GUI commands sent to the device will have the same outcome during each run (for deterministic apps/tests), which is crucial when evaluating or verifying test-cases.

In case of internal app data the solution to this problem is straightforward. Simply uninstalling and reinstalling the app before each test case is executed will clear such data from the previously executed scenario. The `adb` command `adb shell pm clear <package-name>` also clears the app data similarly to the “Clear data” button, in the device settings module. However, files that are saved externally are more difficult to handle properly. First, a tester must determine where such files are saved on a particular device and whether these files affect app execution in a derogatory manner with respect to the testing goals. If it is determined by the tester that the persistence of an external file is causing unexpected or unwanted behavior between executions of subsequent test cases, then this data must be cleared by an `adb` shell command. This will ensure that each test case is executed in the same environment, leading to the typically desired goal of deterministic test cases.

3.2.4 Extraction of GUI Hierarchies for Dynamic Screens

In GUI-based testing of mobile apps, the extraction of the currently displayed GUI-hierarchy on a device is a fundamental operation for data collection and future event generation, depending on the automated technique being used. In practice we relied on `uiautomator` to extract GUI-related information from devices under test, as it fast and flexible. However, as described in Section 3.1 we encountered two major problems including (i) an inability to collect GUI-information for components “behind” dialogs, and (ii) a bug in the `uiautomator` that does not allow capture information from dynamically changing GUI components (e.g., a countdown timer is running on the screen). We are able to solve the second issue by modifying the `DumpCommand.java` file from the AOSP v4.4.2 [5] (although the fix should apply to any recent OS version in which `uiautomator` is available).

There is a potential problem with the base implementation of `uiautomator`’s dump command in the context of GUI-testing, namely, if an xml GUI dump is attempted when a screen is transitioning/updating, no dump will be returned as opposed to waiting until the screen is idle before attempting to grab a dump. As a consequence of this, if a screen has a single component, such as a timer or dynamically updating widget, `uiautomator` will never view the screen as idle and a GUI dump for that screen will not be possible. While the original behavior may have been desired as a safe method to extract xml GUI dumps from a device, it can dramatically hinder GUI-based testing that relies on such information, particularly for apps with dynamically updating screens. To remedy this, we modified the `DumpCommand.run()` method as explained in detail in our online appendix [20]. Our modified version will make ten attempts (at most) at obtaining the ui-hierarchy from a static screen, however if no idle state is reached, the command will still provide a dump.

Once the file is modified, the AOSP source code needs to be recompiled to generate the `uiautomator` executable and libraries that should be replaced in the Android

device/emulator. When recompiling the AOSP, it is best to follow the recommended practices [4]. A full description of this solution, including pre-modified downloadable files, a tutorial on how to modify and recompile the AOSP, and how to push the modified files to a device or emulator are available at our online appendix [20]

3.2.5 Running Concurrent ADB Connections

`adb` clients connected to different instances of Android devices or emulators are limited to ports in the range of 5555 to 5585. Each Android device or emulator uses *two ports* within this range, one for the `adb` and another for console connections, therefore, the first device instantiated on a particular `adb` server would occupy ports 5554 and 5555. Therefore, an `adb` server can handle up to 15 devices/emulators. With a default setup using the `adb` server, automation of Android apps execution/testing has an upper bound in terms of devices/emulators that can be used (more details in [102]). One potential solution is to enable multiple (n) `adb` servers on the same machine to have $n \times 15$ devices/emulators, by starting different `adb` servers on different ports [11]. However, we tried to enable multiple `adb` servers in that way and we found that while this methodology works well on specific SDK platform versions, the newest versions of the Android SDK platform led to inconsistent behavior. We found that revision 23.0.1 and earlier of the platform tools allow for the stable creation and use of multiple devices on multiple `adb` servers. However, newer versions of the SDK platform tools led to several issues. Namely, when trying to start an Android emulator on an `adb` port other than the default port, rather than the device connecting to the expected port, it would connect to the default port and show a status of `inactive`, making the device inaccessible via `adb`.

3.2.6 Enabling Parallel Execution of AVDs

The most economically feasible method of enabling testing approaches capable of executing apps concurrently on varying device configurations this is through the parallel execution of concurrent AVDs. Additionally, as research progresses with regard to automated testing

approaches based on dynamic analysis, and researchers aim to provide practical, usable open source tools to developers, the concurrent execution of several virtual devices will most likely become a requirement. Thus, we present a solution that we utilized in a prototype for large-scale automated execution/testing after encountering the issues described in Section 3.1.2. Our proposed solution had several *key features*: (i) it has the ability to run many instances of AVDs in parallel on hardware with modest specifications; (ii) it utilizes only open-source tools to reduce cost and encourage potential dissemination; and (iii) it is highly configurable so as to accommodate varying hardware configurations such as screen-size, memory, and graphics components.

Our solution is built around the `androidx86` project [3], available under the Apache Public License 2.0. The goal of this project is to port the AOSP to Intel’s x86 architecture and instruction set so that it can run natively on such hardware. Note that unlike the emulator-based AVDs from Google or Genymotion, where hardware components are simulated in software (and often hardware accelerated depending on the platform), `androidx86` can be run directly on x86 hardware, or virtualized and managed by a hypervisor. This fact helps us overcome the *accidental* issues discussed in Section 3.1 that inhibit the other AVDs in terms of parallel execution. Specifically, this helps to overcome the issues of (i) GPU limitations, as the minimum video memory required for each VM is relatively small, and even with a modest GPU, devices of nearly any screen size, resolution and pixel density can be instantiated; and (ii) certain hardware acceleration limitations, because if the host machine is able to run a virtualization software like VirtualBox, then it is able to run these Android virtual machines. To summarize, using `androidx86` images through VirtualBox, allows for highly scalable parallel execution of AVDs, limited only by the amount of RAM and processing power of the CPU. For example, this enables the execution of ten AVDs on a modern laptop, or over 50 AVDs on our server configuration described in the online appendix [20].

In the context of our solution, we use the images provided by the `androidx86` project and instantiate them in VirtualBox [52] so that the hardware can be customized to the

desired specifications for testing on devices of differing hardware configurations. The setup process for booting an `androidx86` image from VirtualBox is very straightforward, and there are several freely available guides online, therefore, we forgo a detailed discussion of such a set-up in this paper, but host an overview and links to such guides in our online appendix [20]. Additionally, once the image is set up in VirtualBox, it is trivial to adjust many hardware settings such as the amount of RAM and video memory allocated. Minimum hardware settings for the Virtual Machine to boot and run satisfactorily are also provided in our appendix. Simulating different screen sizes and allowing the virtual devices to concurrently connect to the host machines adb server and network are more difficult to configure, however, we provide a detailed tutorial and explanation online [20]

3.3 An Infrastructure for Large Scale Execution and Testing

In Section 3.2 we described a set of solutions that we have designed and implemented for some of the issues described in Section 3.1 related to the large-scale dynamic analysis of Android apps with a focus on testing. In addition to those individual solutions we designed a flexible and scalable architecture that enabled us to perform concurrent execution of Android apps for the purpose of dynamic analysis (Fig. 3.3). The proposed infrastructure can be instantiated with commodity machines and open-source software. Moreover, it is built on top of the solutions presented in Section 3.2, and considers the main design decisions from previous work. In the course of our work, this architecture has supported an approach (with research and industrial purposes) for energy optimization as well as large-scale systematic exploration and testing including over 4,500 test cases (100 event sequences each) via a maximum of 100 parallel Android virtual devices with varying configurations across two machines (50 concurrent instances per machine)⁴. In particular, our proposed architecture was designed with a client-server style, in which multiple client systems can submit requests for asynchronous execution of different tasks

⁴Details and references are not provided because of the double-blind requirement in the ASE CFP.

in a virtual Execution Engine (EE) (i.e., composed of one or more EE-nodes). We defined the architecture with the following design drivers:

- Processing and execution tasks should be done on an Execution Engine that is invoked by remote clients;
- The EE should be open for extension but closed for modification in the sense that several tasks (e.g., random testing, or GUI ripping) should be enabled easily. Therefore, workers in charge of executing the tasks can be implemented by following a predefined interface in the EE;
- Asynchronous and decoupled communication between the clients and the EE. This is particularly important because some execution and testing tasks might be lengthy and computationally intensive;
- Hybrid storage strategy: the EE should not store any information of the tasks or the proposed solution after completing the tasks, but clients should have local DBs (if required). Images and files generated by the EE, that are rendered/painted in the client afterwards, are stored in an external web folder;
- Potential horizontal-scalability by adding more EE nodes;
- Vertical scalability controlled by pools of background and `android-x86` VM threads, and a queue of tasks.

To achieve loose coupling and asynchronous calling between clients and the EE while enabling of horizontal scalability, we decided to use a “Message Bus” architectural style, which can be supported by technologies such as OpenESB [37], a message broker like RabbitMQ [41], or a NoSQL engine like CouchDB [6]⁵. The message bus is the communication bridge between the clients and EE nodes. The tasks from clients are pushed to the bus in

⁵CouchDB provides storage of JSON documents and supports sharding to enable horizontal scalability. It provides a REST interface which reduces the time required for programming the connection between the clients and CouchDB.

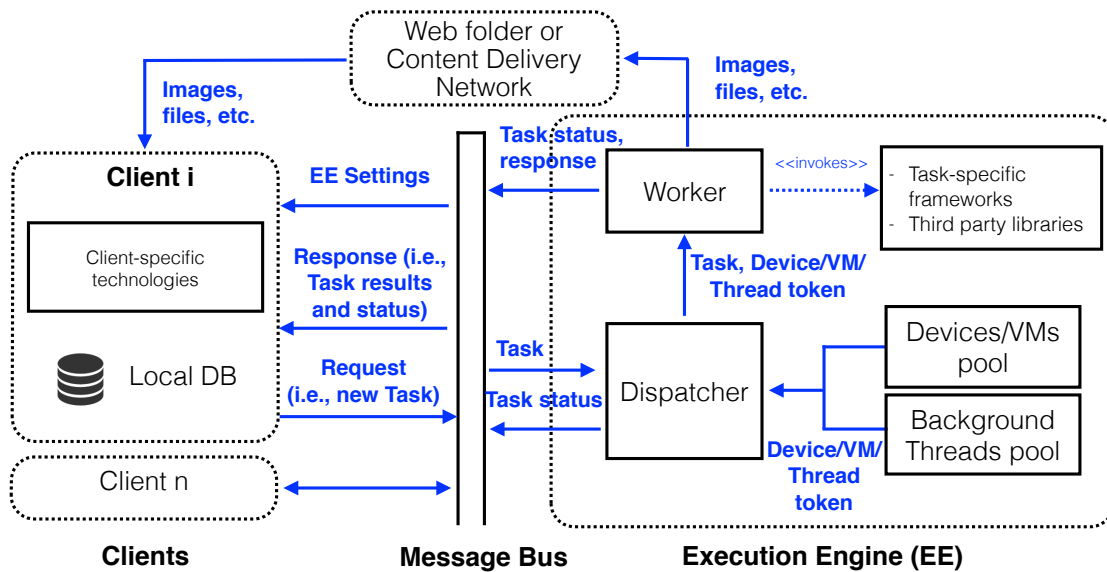


Figure 3.3: A proposed architecture for enabling large-scale execution and testing of Android apps.

a JSON-like format, which is not coupled to the EE implementation⁶. If more EE nodes are required to scale-out, more EE nodes should be registered in the bus. An EE node queries the requests in the “bus”, and then dispatches the requests to workers (i.e., the units in charge of running tasks) following a FIFO policy. During the dispatching process, the EE verifies the availability of background threads and free devices or VMs in the host machine. If no device/VM or threads are available in the EE, the dispatcher keeps the tasks in the queue. Otherwise, the task is assigned to a worker.

Once a worker is dispatched with a task, an device/VM, and a thread, the worker starts to run (asynchronously) the requested task. During the execution, the workers update the status of the assigned tasks directly to the “bus”, and after completion, the generated artifacts are updated in the task document (in the bus) or copied into a public web folder or a content delivery network; also, the tokens for the thread and device/VMs are released back to the pools after completion. Note that, because there are no synchronous messages between the workers and the clients notifying when a task is finished, the clients should query the “bus” and synchronize their local databases to reflect the responses/results

⁶The JSON document should have metadata for the EE, which is used for the dispatching process. However, the format for task-specific data (e.g., the APK to execute and the test case to run) is free given that this data is only used and understood by the corresponding worker.

generated with each task. In our design we include the possibility of having real devices connected to the EE-nodes, because dynamic analysis tasks like performance analysis require execution on real devices instead of virtual devices.

During the implementation of the EE, we found `android-x86` VMs particularly useful for duplication and deployment (on different physical hosts) of pre-packaged environments (i.e., Android OS with application data and required services) required for dynamic analysis or testing. Also, the VMs allow for easy roll back or process restoration based on VirtualBox’s capabilities for saving the state of the VM memory and disk image to a host-based file.

There is a substantial amount of work in the related literature supporting research approaches and technological solutions to enable parallel execution of, or to support scalable dynamic analysis of Android apps. Yet, none of those papers provides the details of the issues, solutions, or guidelines when enabling large-scale analysis of Android apps. We consider these papers as relevant and related to this experience report. Our contribution is complementary to this body of work to further support developers and researchers in designing, implementing and setting-up in-house infrastructures for scalable execution and analysis of Android apps according to their specific needs. In this section we describe this relevant work, which is summarized in Table 3.2.

Several papers reported the usage of cloud-based architectures to support app testing. For instance, Mahmood *et al.* [177] proposed a cloud-based approach for generating a large number of test cases to assess security. The authors use an infrastructure that instantiates a virtual machine image (with a predefined environment) on several virtual machines running in parallel using Amazon EC2. The instances are used to execute test cases on the apps under analysis. Starov *et al.* [220] present a Testing-as-a-Service (TaaS) solution for cloud-based testing. The cloud environment is composed of multiple *Application* nodes that handle the communication with connected physical devices. The TaaS solution also has a *Master Application* node that coordinates the work flow and outputs of the *Application* nodes. The communication between all the components is

made through REST services.

Wen *et al.* [238] proposed an approach called *PATS* to improve the execution time of test cases by using a master-slave model; *PATS* has a module called *Testing coordinator* that schedules jobs in slave nodes (i.e., virtual machines) to execute unexplored UI-states. Another example of master-slave style, is *ATT* [186], which uses an *AgentManager* to control worker instances; each worker is in charge of handling one device/emulator. Experiments with *ATT* were performed by varying the number of emulators from 1 to 32, showing that *ATT* is able to conduct parallel test executions. However, the paper does not provide any implementation details or report issues related to large-scale execution.

Besides testing, malware detection has been also supported on infrastructures for large-scale execution. Malek *et al.* [179] report the usage of a cloud-based framework to find vulnerabilities on Android apps. The framework has a component called *Test Execution Environment* that executes multiple instances of an app in parallel on emulators. Maggi *et al.* [176] developed a framework called *AndroTotal* for detecting malware by using existing malware detectors. *AndroTotal* requires an APK and a JSON with metadata information required for the execution. Both APK and JSON messages are pushed to an execution queue. Multiple workers running in parallel (each one handling one device) are in charge of taking the requests from the queue and executing the apps in a device/emulator. *DroidDolphin* [239] combines dynamic and static analyses to detect malware. *DroidDolphin* installs an APK to collect execution logs, and then executes the app automatically in an emulator from a pre-instantiated pool; this pool enables parallel execution of apps. *Andlantis* [70] identifies malicious apps by relying on dynamic analysis; *Andlantis* uses `androidx86` images (similarly to our proposed solution) to create virtual machines that execute the apps.

Other approaches propose mechanisms for reducing execution time of dynamic analysis or large-scale data collection. Mutti *et al.* [197] proposed a platform, *BareDroid*, that can work on top of any Android execution engine to speed up the testing process. *BareDroid* has a supervisor component orchestrating the communication between a cloud of physical

Table 3.2: Proposed approaches for parallel and large-scale execution of Android apps. The columns are as follows: 1) Purpose of the tool [Security Evaluation, Testing Cloud]; 2) Device Type [Physical Device, Emulator, Virtual Machine]; 3) Tool used for injecting input events [Manual, Robotium, MonkeyRunner, UiAutomator]; 4) Type of exploration strategy [Manual, Systematic, Multiple, Ruled-based, Random]; 5) Requires Instrumentation [Yes, No] ; 6) Type of Analysis [Static Analysis, Dynamic Analysis] ;7) Number of instances or clients executed in the experimentation phase

Paper	Purp.	Device	Events	Exp.	Instr.	Ana.	I/C
[74]	SE	PD	MA	MA	N	D	20
[179]	SE	E	—	S	N	S/D	—
[177]	TC	E	R	S	N	S	100
[220]	TC	PD	MR	MU	N	S	—
[176]	SE	PD/E	MR	RU	N	S	6
[239]	SE	E	—	S	Y	S/D	32
[70]	SE	VM	MR	S	N	D	188
[238]	TC	VM	U	S	N	D	2
[186]	TC	PD/E	R	RA	Y	D	32

devices and an analysis framework. Burguera *et al.* [74] uses crowdsourced-based collection of execution traces on real devices; a central server collects and parses the traces, and finally performs a behavior-based malware detection.

In summary, these related works provide value in outlining some useful design decisions that can be used by researchers and developers, such as the usage of (i) cloud-based architectures and (ii) architectural styles in which a central manager that dispatches tasks to workers/slaves on different “execution” nodes. Physical devices and emulators are the preferred choice in the analyzed papers, despite of the cost of keeping a physical devices farm and the emulator issues (reported in Section 3.1.2); and only two papers [177, 220] report the usage of Virtual Machines (our preferred choice). However, while these publications provide value in terms of high level-infrastructure they do not provide implementation details or the issues and technical challenges they faced in enabling concurrent testing of Android apps. Lack of such details makes replication of these frameworks arduous at best, which can in turn stifle research progress in scalable automated dynamic analysis. This experience report is positioned to remedy this and offer researchers and developers a set of practical solutions and guidelines for developing a framework supporting the large-scale execution and testing of Android applications.

3.4 Discussion

The solutions described in this Chapter are a technological contribution for practitioners and researchers. Moreover, the taxonomy of issues is an starting point for other solutions that can provided by the research community. However, the most important contribution of the work described in this Chapter is the infrastructure that facilitated the implementation of novel approaches for evolution and maintenance tasks at a large scale. Building the infrastructure required the implementation of a set of libraries for dynamic analysis of Android apps, called the SEMERU Framework, which is currently supporting all the research efforts of the SEMERU group related to maintaining and evolving mobile apps. An example of an approach using the SEMERU Framework is the one described in Chapter 4 for automatic generation of test cases. In addition, the infrastructure allowed the implementation of the approach for optimizing energy consumption (GEMMA) described in Chapter 5, and other products of the SEMERU group such as a novel approach (CRASHSCOPE) for automatic discovery, reporting and reproduction of crashes in Android apps [195]. Both products are currently prototyped for industrial usage with the help of the Huawei European Research Center.

In summary, the empirical studies described in Chapter 2 highlighted (i) the importance of dynamic analysis for supporting evolution and maintenance of Android apps, and (ii) the need for solutions and versatile infrastructures that support dynamic analysis under the constraints imposed by the mobile development cycle. Then, in this Chapter, we presented a set of solutions for enabling automatic and large-scale execution and testing of mobile apps, which are foundational tasks for dynamic analysis. The solutions and infrastructure (including the SEMERU framework) were designed to tailor multiple dynamic-analysis based tasks, and have been proved to support several research efforts of the SEMERU group [162, 165, 194, 167, 195]. In the following two chapters, we describe two of those implementations, in particular for two tasks that are highly important in the mobile development cycle: testing and energy optimization.

3.5 Bibliographical Notes

The papers supporting the content described in this Chapter were written in collaboration with the members of the SEMERU group at William and Mary and researchers from the Huawei European Research Center:

- **Linares-Vásquez, M.**, “Enabling Testing of Android Apps”, in Proceedings 37th IEEE/ACM International Conference on Software Engineering (ICSE’15), ACM Student Research Competition, Florence, Italy, May 16-24, 2015, pp. 763-765. **First Place Graduate Winner ACM Student Research Competition.**
- **Linares-Vásquez, M.**, Moran, K., Bernal-Cárdenas, C., Poshyvanyk, D., Farahbod, R., Kilian-Kehr, R., “Experience Report: Enabling Large-scale Execution and Testing of Android Apps”, under review.

Chapter 4

Combining Usage and GUI

Models to Support Automatic

Test Cases Generation

Although several tools are available to support automated execution of Android apps for validation purposes, in practice, testing is still performed mostly manually as recent survey study results show [143, 149] and confirmed with the survey described in Section 2.6. Limited testing time during the development process tends to prohibit the use of data from these manually generated scenarios for different devices [143] in a continuous development fashion, especially taking into account that some apps may have users coming from as many as 132 unique devices [146]. A significant amount of work is required to generate replay scripts that are coupled to screen dimensions for a single device (i.e., one script is required for each target device). Consequently, these scripts become quickly outdated when significant changes are made to the app’s GUI [143, 116, 149].

Existing research tackled some of these issues by deriving models that represent the GUI and behavior of apps. These models are abstract representations that can be decoupled from device dimensions and event locations and can still remain valid when small changes are done to the app (e.g., button location change). For instance, some representa-

tive approaches for deriving such models use either dynamic [224, 211, 55, 68, 175, 80, 198, 227] or static analyses [212, 187, 141, 75, 59, 95]. However, current approaches fall short in (i) generating scenarios that are representative of natural (i.e., typical end-user) application usages [227], (ii) taking into account the context which exists in an app’s execution history [89, 227], (iii) generating sequences with previously unseen (i.e., not available in artifacts used to derive the model) but feasible events. Moreover, utilizing model-based testing techniques for GUI-based testing in industrial contexts may be particularly challenging because creating such models requires specialized expertise and using these models (for manual and automated testing) assumes a logical mapping between the model and the actual system that was modeled [54].

Another issue of current approaches for automated testing of mobile apps is in the lack of consideration of event sequences as part of use cases of features. Test cases composed of events that are not exercised having the usage model in mind, are not representative of real application usages. The empirical study described in Section 2.6 showed that surveyed participants prefer to design test cases targeting individual or combinations of use cases/features. Therefore, to take into consideration the way as users and developers exercise use cases/features in Android apps, we designed a method for generation of execution scenarios (a.k.a., test sequences) that combines GUI and usage models.

In practice, developers constantly test their apps *manually* by exercising apps on target devices. In fact, manual testing is usually preferred over automated approaches for testing mobile apps [143]. However, oftentimes, this execution data is simply thrown on the ground and never used when an app needs to be retested. *Our key hypothesis is that all this data that is produced from regular app usages by developers, testers, or even end-users can be effectively recorded and mined to generate representative app usage scenarios (as well as the corner cases) that can be useful for automated validation purposes.* Furthermore, our intuition is that the models mined from execution (event) traces can be augmented with static analysis information to include unseen but feasible events.

In this chapter, we propose a novel hybrid approach for mining GUI and usage models

from event logs collected during routine executions of Android apps. Our approach (MONKEYLAB) derives feasible and fully replayable GUI-based event sequences for (un)natural app usage scenarios (i.e., actionable scenarios) based on the novel *Record* \rightarrow *Mine* \rightarrow *Generate* \rightarrow *Validate* framework. MONKEYLAB provides stakeholders with an automated approach for scenario generation that can be as powerful as manual testing. MONKEYLAB mines event traces and generates execution scenarios using statistical language modeling, static analysis, and dynamic analysis. To generate (un)natural event sequences, our novel approach (i) augments the vocabulary of events mined from app usages with feasible events extracted statically from the app’s source code and (ii) exploits a space of possible events and transitions with different flavors of language models capable of modeling and generating combinations of events representing natural scenarios (i.e., those observed relatively frequently in app usages) and corner cases (i.e., those sequences that were observed relatively infrequently or not observed at all).

We evaluated MONKEYLAB on several medium-to-large Android apps from Google Play and compared MONKEYLAB to other commonly used approaches for generating GUI-based event scenarios. The results demonstrate that MONKEYLAB is able to generate effective and fully replayable scenarios. Moreover, MONKEYLAB is able to generate scenarios that differ from observed executions enabling it to explore other paths that could trigger unexpected app crashes.

In summary, this Chapter provides the following contributions:

- A *Record* \rightarrow *Mine* \rightarrow *Generate* \rightarrow *Validate* framework for generating actionable scenarios for Android apps. We designed MONKEYLAB to be independent from specific Android devices or API platform versions.
- A novel mechanism for generating actionable scenarios that is rooted in mining event traces, generating execution scenarios via statistical language modeling, static and dynamic analyses, and validating these resulting scenarios using interactive executions of the app on a real device (or emulator). In particular, we explore interpolated

n -grams and back-off models, and we propose three different flavors (i.e., up, down, and strange) for generating (un)natural scenarios.

- A thorough empirical evaluation and comparison of MONKEYLAB to competitive approaches on Android devices (Google Nexus 7 tablets). Experimental data, videos of the generated scenarios, and other accompanying tools are available in our online appendix [170].

4.1 Background and Related Work

Automatic generation of GUI-based scenarios (event sequences) has applications not only in automated testing but also in creating app usage documentation [236]. In general, event sequences can be generated automatically by relying on models built (i) statically from app source code, (ii) dynamically from interactive app executions (e.g., GUI ripping) or from execution traces, (iii) manually defined by programmers, or (iv) approaches using random chains of events without any knowledge of the app such as the widely used Android UI `monkey` [113]. Once the model is defined, it can be used to generate sequences of feasible (in theory) events.

Models derived from the app source code rely on static analysis [212], symbolic [141, 75] and concolic execution [59] techniques. Rountev *et al.* [212] proposed a method for statically extracting GUI components, flows of GUI object references and their interactions. The concolic-based testing model proposed by Anand *et al.* [59] generates single events and event sequences by tracking input events from the origin to the point where they are handled.

Models derived using dynamic analysis are mostly based on interactive execution of the app for systematically identifying the GUI components and transitions (i.e., GUI ripping). The execution is usually done heuristically by using some strategy (e.g., depth-first search (DFS) or a non-uniform distribution). Examples of approaches and tools relying on systematic exploration are the work by Takala *et al.* [224], the tools *VanarSena* [211],

AndroidRipper [55], A³E [68], *Dynodroid* [175], *SwiftHand* [80], *OME** [198], and *Mobi-GUITAR* [56] that extracts a state-based model of the GUI, which is used to generate JUnit test cases.

Dynamic analysis has also been used for collecting real execution traces and inferring some grammar-based rules or statistical models describing observed events. Representative approaches of models extracted from execution traces collected apriori include *SwiftHand* [80], which uses statistical models extracted from execution traces, and the approach by Elbaum *et al.* [95] which generates test cases from user-session data. TAU-TOKO [86] mines typestate models from test suite executions and then mutates the test cases to cover previously unobserved behavior. Recent work by Tonella *et al.* [227] derives language models from execution traces.

Hybrid models, such as *ORBIT* [242], have also been proposed, which combine static analysis and GUI ripping. Moreover, *Collider* [141] combines GUI models and concolic execution to generate test cases that end at a target location. *EvoDroid* [178] extracts interface and call graph models automatically, and then generates test cases using search-based techniques. Yeh *et al.* [75] track the execution paths of randomly generated input events by using symbolic execution. Afterward, the paths are used for detecting faults and vulnerabilities in the app.

Event sequences can also be generated by relying on random distributions. For instance, the Android UI *monkey* [113] is widely used for generating random input events without prior knowledge of the app’s GUI. Although a tool like *monkey* helps automate testing, it is not robust since it is prone to providing a high ratio of invalid inputs. *Dynodroid* [175] also includes a strategy for generating GUI and system level events via a uniform distribution where available GUI events are discovered interactively using GUI ripping.

Regardless of the underlying analyses to build the model, model-based GUI testing is inherently difficult, because the current state of an app (available components and possible actions) can change continually as new GUI events are executed. For example, consider

a state machine-based GUI model in which each node (state) is a screen (i.e., activity) of an Android app, and the transitions are actions on specific GUI components (e.g., click the OK button). Some of the real transitions are feasible under certain conditions (e.g., delete a task in a non-empty list of a TODO app), and some states that are reachable only after executing natural (and unnatural) sequences of GUI events (e.g., customized scroll behavior on an empty list) are hard to model using state machines. In addition, although systematic-based exploration techniques could execute events on all of the GUI components of the app, the execution follows a predefined heuristic on the GUI (e.g., DFS) that neither represents natural execution scenarios nor considers execution history. Existing approaches also assume a high rate of generated event sequences that are simply infeasible [89, 227]. For instance, some events in the sequence can be infeasible in the context of a given test case (e.g., an invalid action on a component or an action on a component that is not displayed in the current GUI state). Another issue, which is pertinent to the models extracted from execution traces, is their inability to generate sequences with unseen (i.e., not observed in the traces) but feasible events.

4.2 Mining and Generating Actionable Execution Scenarios with MONKEYLAB

Given that manual execution of Android apps for testing purposes is still preferred and relied upon over automated methods [143], we set out to build a system for generating inputs (event sequences) to mobile apps on Android that would simulate the convenience, naturalness, and power of manual testing (and utilize all the data that is produced during these app usages) while requiring little effort from developers. Also, we identified a set of four key goals that we felt such a system must satisfy in order to be useful:

1. The solution for generating executable scenarios needs to resemble manual testing, where the process of collecting test scripts allows developers/testers to interact with the app naturally by using gestures and/or GUI-based actions. One solution to this

issue is represented by the event log collection capability used by *RERAN* [100]. However, *RERAN* replays only what it records and does not allow an arbitrary recombination of events for replay; moreover, scripts recorded with *RERAN* cannot be easily deciphered by testers since they are low-level, hardware-specific events that are coupled to screen locations (e.g., events collected on a Nexus 4 will not work on a Galaxy S4).

2. The solution should have context awareness of an app’s execution history, where current possible actions are generated and executed as event streams to facilitate the highest possible testing coverage. In this regard, our key insight is to rely on statistical language models (n -grams) to generate event sequences by using program structure and previous executions of an app. Our motivation behind using language models (LMs) is in their capacity to represent event streams while maintaining contextual awareness of the event history. *MONKEYLAB* uses LMs for representing sequences of events as tokens (more specifically, *Object:Action* lexemes) that can be extracted statically from the app and also mined from the app’s execution. Extracting the tokens only from execution traces carries the risk of missing possible feasible events that are not observed in traces. Therefore, combining static and dynamic information may further augment the vocabulary used by LMs in such a way that some unnatural (those that did not appear in traces) but feasible tokens are included in the model, effectively taking into account both the execution history and unobserved events.
3. The solution should be able to generate *actionable scenarios* as streams of events that can be reproduced automatically on a device running the app. These scenarios should not be low-level event streams but rather human readable streams that developers can easily comprehend and are not coupled to locations in a screen. This goal can be met by translating high-level readable tokens (e.g., click OK button) to low-level event streams that can be executed automatically on a target device.

These high-level readable tokens are derived from language models formulated by mining from app usages and source code.

4. The solution should not impose significant instrumentation overhead on a device or an app in order to be able to reproduce automatically generated scenarios. In order to meet this goal, we rely on the replay capability provided by the input commands for remote execution in Android [44]. However, any framework for remote execution of GUI events can be used such as Robotium [43].

Having these goals in mind, the proposed approach for generating actionable scenarios is described by the following framework: (i) developers/testers use the app naturally; the event logs representing scenarios executed by the developers/testers are *Recorded*; (ii) the logs are *Mined* to obtain event sequences described at GUI level instead of low-level events; (iii) the source code of the app and the event sequences are *Mined* to build a vocabulary of feasible events; (iv) language models are derived using the vocabulary of feasible events; (v) the models are used to *Generate* event sequences; (vi) the sequences are *Validated* on the target device where infeasible events are removed for generating *actionable scenarios* (i.e., feasible and fully reproducible). The framework and the architecture are depicted in Figure 4.1 as well as real examples of event logs collected in the *Record* phase, event sequences derived after the *Mine* and *Generate* phases, and actionable scenarios generated with the *Validate* phase, for the Android app (Mileage). In the following subsections, we describe each step of the framework and the components required for each step.

4.2.1 *Record*: Collecting Event Logs from the Crowd

Collecting event logs from developers/testers should be no different from manual testing (in terms of experience and overhead). Therefore, developers/testers using MONKEYLAB rely on the `getevent` command for collecting event sequences, similar to *RERAN* [100]. The `getevent` command produces logs that include low-level events (see `getevent` log example in Fig. 4.1), representing click-based actions as well as simple (e.g., swipe) and complex

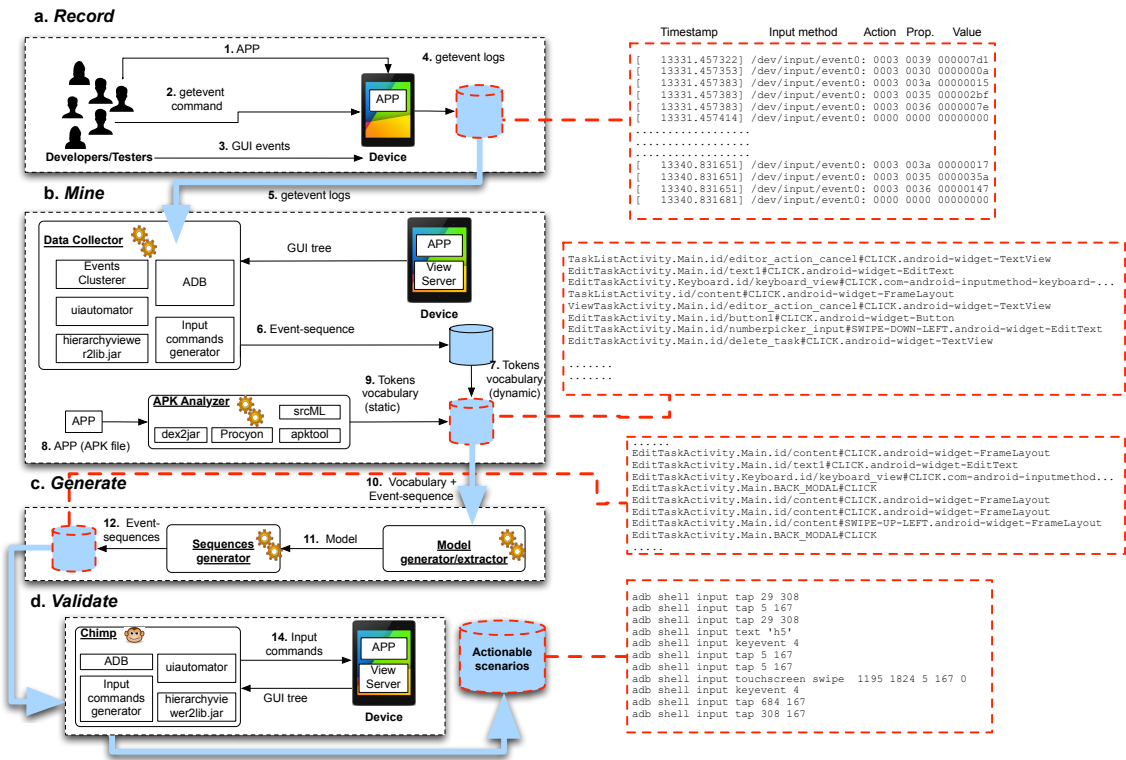


Figure 4.1: MonkeyLab architecture and the *Record*→*Mine* →*Generate*→*Validate* framework

gestures; those logs are collected during an app execution. After MONKEYLAB is enabled, developers/testers exercise/test the app as in manual testing. After having executed the app, the logs are generated with the `getevent` command and copied to the logs repository. Since our log collection approach poses no overhead on developers/testers (they just use apps as usual), this setup permits collecting logs on a large scale. In fact, *this log collection approach can be easily crowd-sourced, where logs are collected from daily app usages by ordinary users.*

4.2.2 Mine: Extracting Event Sequences from Logs and the App

The goal of the *Mine* phase is to extract the vocabulary of events (i.e., feasible events) and translate the `getevent` logs required to build the language models. The vocabulary of

events is extracted from the source code of the app and also from the event logs. However, the low-level events are coupled to specific locations on the screen and do not describe the GUI components in the app; in addition, a single GUI event is represented by multiple lines in a log (see Fig. 4.1). A line in a `getevent` log has a timestamp, input method (e.g., screen or physical keyboard), an action (e.g., `003 = click`), a property related to the action (e.g., `0035 = x-axis position`), and the property value. Therefore, to eliminate the dependency of the actions on the screen coordinates in specific devices, we translated the `getevent` logs to a GUI-based level representation: we model an event e_i —represented by multiple lines in a `getevent` log—as the tuple $e_i := \langle Activity_i, Window_i, GUI-Component_i, Action_i, Component-Class_i \rangle$ (see examples of event sequences in Fig. 4.1). We included the *Window* element to distinguish actions on the activity (i.e., screen) and actions on the displayable Android keyboard. For the *Action* element, we included `click`, `long-click`, and `swipe`. This representation considers the fact that Android apps are composed mostly of Activities representing screens of the application. Each Activity is composed of GUI components that are rendered dynamically according to the app’s state. Thus, the number of GUI components that are visible in an activity can be different across time. Additionally, the set of feasible actions for each component depends on the component type. For instance, `click` and `long-click` event handlers can be attached to a button but not to a `swipe` gesture handler. The *Component-Class* field is necessary to validate GUI-components in the app at runtime. This representation of e_i tuples is used for both generating the vocabulary of events from app code and event sequences from `getevent` logs.

4.2.2.1 Mining GUI events statically from APKs (APK Analyzer)

The *APK-Analyzer* component (Fig. 4.1-b) uses static analysis to extract from the app source code a list of feasible GUI events. The list is used to augment the dynamically built vocabulary from user event streams. To achieve this, GUI components are extracted from decompiled APKs before links between these components to activities, windows,

and actions/gestures are constructed. To extract this information, the *APK-Analyzer* (i) uses the *dex2jar*[16] and *Procyon*[221] tools for decompilation and (ii) converts the source files to an XML-based representation using *srcML* [50]. We also rely on *apktool* [7] to extract the resource files from the app’s APK. The `ids`, types, and hierarchy of the GUI components were extracted from the XML files of the APK resources.

The next major step in building the static vocabulary is linking each GUI component to its respective actions/gestures. Rather than parsing the source code to determine which gestures are instantiated, the *APK-Analyzer* assigns inherent gestures to the types of GUI components. For standard Android component types, the linking is done with expected gestures, e.g., `Button` would be linked with `click` and `long-click`. For custom components, the *APK-Analyzer* parses source code to determine gesture handlers and event listeners, which are attached to the custom components. After linking actions/gestures and types to the components is complete, the *APK-Analyzer* links the components to the Activities in which they appear. The list of events extracted statically from the app is represented by a set of tuples e_i . It should be noted that the APK Analyzer cannot generate a static vocabulary from obfuscated code nor code where components are instantiated dynamically.

4.2.2.2 Mining GUI events from event logs (Data Collector)

The e_i tuples require high-level information (i.e., activity, GUI component, window) that is not provided by the `getevent` logs. Thus, we implemented a component (*Data collector* in Fig. 4.1-b), which is able to translate `getevent` logs into sequences of e_i tokens. The *Data collector* replays the logs in a ripping mode in order to dynamically collect the GUI information related to the event. The sentences (i.e., lines) in a log L_k are grouped to identify individual GUI events. Then each group of sentences (c) representing a GUI event is translated into a natural language description (e.g., `<click, {x = 10, y = 200}>`). For identifying the corresponding component in the GUI, we queried the `Android View Server` running on the device. The `View Server` provides a tree representation of the GUI

components with some attributes such as the top-left location of a component and the dimensions. Given the event location, we traversed the GUI tree looking for the component area (top-left corner plus dimensions) that contains the event’s location. For identifying the current activity we used the `adb shell dumpsys window windows` command, and then we looked for the property `mCurrentFocus|mFocusedApp`. For identifying whether the keyboard is displayed, we queried the list of current windows in the `View Server`. With the e_i tuple, we built an input event command[44] that can be remotely executed using the `Android Debugger Bridge` (e.g., `adb shell input tap x y`). Finally, we executed the input command to update the app state and continued with the next group c . The output of the procedure is the list T_k of e_i tuples extracted from the L_k `getevent` log. The T_k sequences and unique e_i tuples are copied to the sequence and token repositories, respectively.

4.2.3 *Generate*: Event Sequences with Language Models

The vocabulary of event tokens extracted from the app code and the event logs, both represented as GUI-level events, are the universe of feasible events in the app and are natural use cases respectively. They represent the GUI model of the app including activities, components, feasible actions, and natural transitions. This data can be used to build statistical models for generating streams of events as execution scenarios of the app. In fact, the *Generate* phase of our framework uses LMs [140, 145, 83] to generate (un)natural sequences of events for that purpose. These LMs are trained with the vocabulary and event sequences extracted in the *Mine* phase.

Our motivation behind using LMs is their capacity to represent event streams while maintaining a contextual awareness of the event history. Hindle *et al.* [128] demonstrated the usefulness of applying LMs to software corpora because real programs written by real people can be characterized as *natural* in the sense that they are even more repetitive than natural languages (e.g., English). Tonella *et al.* [227] exploited this interpretation of naturalness by casting it to streaming events as test cases. *Our hypothesis is that*

event-sequences extracted from getevent logs emit a natural set of event streams, which map to core functionality, as well as an unnatural set of event streams. Language models inherently capture natural behavior, but we also consider the complement of this natural space, thereby improving the interpretive power of these statistical models. Our novel interpretation of software LMs systematically segments the domain of a LM over dynamic traces and imputes testing semantics to each segment. For example, a sampling engine may select tokens from the natural space to improve coverage, yet it may sample from the unnatural space to crash the app. In the following subsections we provide formal definitions behind LMs, LM flavors that we defined in the context of our work as well as details on using LMs for generating event sequences.

4.2.3.1 Language Models

A statistical language model is a probability distribution over units of written/spoken language, which measures the probability of a sentence $s = w_1^m = w_1 w_2 \dots w_m$ based on the words (a.k.a., tokens) probabilities:

$$p(s) = p(w_1^m) = \prod_{i=1}^m p(w_i | w_1^{i-1}) \approx \prod_{i=1}^m p(w_i | w_{i-n+1}^{i-1}) \quad (4.1)$$

In Eq. (4.1), the Markov assumption approximates the joint distribution by measuring the conditional probabilities of token subsequences known as n -grams. After “discounting” the model’s maximum likelihood estimates, there are generally two methods for distributing the probability mass gleaned from the observed n -grams: back-off and interpolation. The Katz back-off model [145] takes the form

$$p_{\mathcal{B}}(w_i | h) = \begin{cases} \alpha(w_i | h) & c(hw_i) > k \\ \beta(h) p_{\mathcal{B}}(w_i | w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases} \quad (4.2)$$

where $h = w_{i-n+1}^{i-1}$ is the history, $\alpha(w_i | h)$ is the discounted maximum likelihood estimate of word w_i given h , $\beta(h)$ is the back-off weight, $c(hw_i)$ is the number of times hw_i appears

in the training corpus, and k is a constant, which is typically zero. If the history was observed in training, then a back-off model says the conditional probability of a word given its history is equal to the discounted estimate of the n -gram, where $\alpha(w_i|h)$ is computed using a smoothing technique such as Good-Turing estimation [101]. Otherwise, the model truncates the history and recursively computes the probability. Back-off models only consider lower-order n -grams when $c(hw_i) = 0$ for every w_i in the vocabulary \mathcal{V} . On the other hand, interpolation considers lower-order n -grams whether or not $c(hw_i) > 0$. The general interpolated model takes the form

$$p_{\mathcal{I}}(w_i|h) = \alpha(w_i|h) + \beta(h)p_{\mathcal{I}}(w_i|w_{i-n+2}^{i-1}) \quad (4.3)$$

where $\alpha(w_i|h)$ is computed using a smoothing technique such as modified Kneser-Ney estimation [79].

4.2.3.2 Language Model Flavors

Back-off (B0) and interpolation (INTERP) are different approaches for computing probabilities, but each LM is a way to characterize the naturalness of a sentence. Moreover, we propose that each model can be used to generate unnatural sentences as well, and these unnatural sentences have clear software testing semantics. Parenthetically, both B0 and INTERP can be decomposed to different flavors—**up**, **down**, and **strange**—that are simple transformations for driving a sampling engine to specific segments of the language model’s domain.

up corresponds to the natural distribution over tokens, but there are subtle differences in the implementation depending on whether the language model uses B0 or INTERP. For example, suppose we are given the following conditional probabilities from a LM: $\theta = \{\alpha(w_a|h) : 0.20, \alpha(w_b|h) : 0.50, \alpha(w_c|h) : 0.10, \alpha(w_d|h) : 0.10\}$. For B0 models, **up** will sample a uniform variate using the cumulative sum of the smoothed estimates. If $c(hw) = 0$, then **up** will back-off to the next lowest order and repeat the procedure,

where unigram probabilities serve as the base case in the recursion. By construction, interpolation mixes input from every n -gram that can be sliced from hw , so **up** recurses on the order through the unigrams to compute $p(w_i|h)\forall w_i \in \mathcal{V}$ using Eq. (4.3). Naturally, while the probability mass is concentrated on expected transitions, **INTERP-up** models are able to reach any transition in the vocabulary of transitions at any point in time. This is not the case for **BO-up** models.

down corresponds to the unnatural distribution. For example, suppose we are given the same conditional estimates θ . For **BO** models, **down** will normalize the estimates, sort the tokens according to their probabilities and then reverse the probabilities over the tokens to produce the following distribution: $\{p(w_c|h) : 0.55, p(w_d|h) : 0.22, p(w_a|h) : 0.11, p(w_b|h) : 0.11\}$. In cases where two tokens have the same probability, **down** will randomly arrange the tokens, so $\{p(w_c|h) : 0.55, p(w_d|h) : 0.22\}$ and $\{p(w_c|h) : 0.22, p(w_d|h) : 0.55\}$ are equally likely. According to the training log, tokens w_c and w_d are unnatural given the context, but **down** seeks the unnatural transitions when it generates samples. For **INTERP** models, **down** will build a categorical distribution over \mathcal{V} using Eq. (4.3). Therefore, as with **up**, we sort the tokens in \mathcal{V} according to their probabilities and then reverse the probabilities over the sample space. **INTERP down** models marshal uncharacteristic transitions in \mathcal{V} given the context into the natural space, replacing frequently observed transitions.

strange linearly interpolates **up** and **down**, which likely has the effect of increasing the entropy of $p(w|h)$. When the entropy of the natural distribution over tokens is low and the mixture coefficient $\lambda \approx 0.5$, **strange** distributions can take an interesting form where the probability mass is simultaneously concentrated on very (un)natural events. This strange case describes a scenario where the generative model produces a stream of natural tokens, yet at any point in time is "equally" likely to choose a corner case in the execution.

4.2.3.3 Generating Event Sequences

The generation engine estimates two n -gram language models—one **BO** and one **INTERP**—using the vocabulary extracted in the *Mine* phase. The default order and smoother for

both models are three and modified Kneser-Ney [79], respectively. Tokens in the vocabulary that are not in the event-sequence repository are factored into the models' unigram estimates. Clients (i.e., instances of the *Chimp* component in Fig. 4.1) can communicate with the *Sequence Generator* by sending JSON requests and reading JSON responses. A request includes: model, flavor, history, and length. The model field can take one of two values: `BO` or `INTERP`. The flavor field can take one of three values: `up`, `down` or `strange`. The history field contains the initial prefix as an array of strings. If the client does not have a prefix, then the history field is the empty string. When the client does not have a history (or the length of the history is suboptimal), then the *Sequence Generator* will essentially bootstrap the history to length $n - 1$, where n is the order of the model(s). When the client gives a history that is longer than n , the *Sequence Generator* will apply the Markov assumption, so the history will be sliced according to the order before querying the language model. The length field governs the length of the sequence to query from the language model (i.e., number of events in the test case). This enables us to use the *Sequence Generator* in serial mode (e.g., requesting a sequence with 100 events) or in interactive mode (i.e., only one event is requested at a time).

4.2.4 *Validate*: Filtering Actionable Scenarios

The streams of events generated by the *Sequence Generator* are expressed at GUI level, which decouples the sequence from device-specific locations. Thus, the *Chimp* component (Fig. 4.1-d) understands the events and *Validates* them dynamically on a device. The validation can be performed in a serial (as in Tonella et al. [227]) or with MONKEYLAB interactive mode.

In the serial mode, the *Chimp* requests a sequence that is validated iteratively following the procedure in Alg. 1. For each event e in the event sequence S , the *Chimp* searches for the component (in the event tuple) in the `Hierarchy View` (i.e., GUI tree) of the current GUI displayed in the target device. If the component is not in the GUI or the current activity is not the same as in the tuple e (line 5), then the *Chimp* discards the event

Algorithm 1: Validating Event Sequences: Serial Mode

Input: S
Output: AS

```
1 begin
2    $i = 1, ATC = \emptyset;$ 
3    $startAppInDevice();$ 
4   foreach  $e \in S$  do
5      $feasible = queryVS(e.component, e.action);$ 
6     if  $!feasible$  then
7        $continue\_with\_next\_event;$ 
8      $\langle x, y \rangle = searchVS(e.component);$ 
9      $cmd_i = getInputCmd(e.action, \langle x, y \rangle);$ 
10     $addEvent(AS, \langle e, cmd_i \rangle);$ 
11     $executeInputCmd(cmd_i);$ 
12     $i = i + 1;$ 
```

(line 7); otherwise the *Chimp* queries the **View Server** (line 8) in the device to identify the location of the component. Afterward, it generates an **input** command by using the location $\langle x, y \rangle$ of the component and the action in the tuple e (line 9). Then the event e and its corresponding input command are added to the actionable scenario (line 10). Finally, the command cmd_i is executed on the GUI to update the app state. This serial model is similar to the one proposed in [227], however we are proposing three different flavors for the interpolated n -grams and the back-off model (Sec. 4.2.3.1 and Sec. 4.2.3.2) and we automatically validate the sequences on a target device.

In the interactive mode (Alg. 2), the *Chimp* requests single events until the target length k is exhausted. Our motivation for this mode is the possibility of having infeasible events in a sequence (i.e., in serial model) that can influence further events. We are augmenting the vocabulary used to train the models, with individual tokens extracted statically from the source code. So, it is possible that unseen—but also infeasible—events appear in a sequence. For example, in the sequence $e_1e_2\dots t_1e_ke_{k+1}\dots t_2t_3e_l e_{l+1}$, tokens t_1 , t_2 , and t_3 are infeasible, which leads to the following issues: (i) the inclusion of events e_k , e_{k+1} , e_l , e_{l+1} into the sequence is influenced by the infeasible tokens t_1 , t_2 , and t_3 ; and (ii)

Algorithm 2: Validating Sequences: Interactive Mode

```
Input:  $k$   
Output:  $AS$   
1 begin  
2    $ATC = \emptyset, history = \emptyset;$   
3    $startAppInDevice();$   
4   for  $i \in 1 : k$  do  
5      $e = queryEvent(history) \text{ feasible} = queryVS(e.component, e.action);$   
6     if  $!feasible$  then  
7        $e = getEvent(randComponent());$   
8        $\langle x, y \rangle = searchVS(e.component);$   
9        $cmd_i = getInputCmd(e.action, \langle x, y \rangle);$   
10       $addEvent(AS, \langle e, cmd_i \rangle);$   
11       $history = e;$   
12       $executeInputCmd(cmd_i);$   
13       $i = i + 1;$ 
```

once the first infeasible token is read by the *Chimp*, the further events in the sequence can also be infeasible in the GUI. This is why, in the serial mode, we opted to skip infeasible events and continue reading until the sequence is exhausted, which drives to sequences with less events than the target k .

Therefore, as a second option, we opted for requesting single events and executing only feasible events. This is how the interactive mode operates. The *Chimp* in interactive mode asks for a single token (similarly to code suggestion problem [128]) until a target number of feasible events (lines 4 and 5 in Alg. 2) is executed on the device. To avoid infeasible events and loops because of chains of infeasible events (i.e., the returned token is always infeasible), we execute a random feasible event—queried from the current GUI state— (line 7) when the component and action in event e are not feasible (line 6). This guarantees that the *Chimp* always executes a feasible event, and that event is the history for the next one (line 11). The interactive mode relies on the language model extracted from the event logs and the source code, and takes advantage of the current GUI state to reduce the rate of infeasible-events produced by the model. In summary, our interactive mode combines static and dynamic analyses as well as GUI ripping.

4.3 Empirical Study Design

Our main hypothesis is that models derived with MONKEYLAB are able to produce event sequences for natural scenarios and corner cases unseen in the scenarios executed by stakeholders. Therefore, event sequences generated with MONKEYLAB should not only include events from scenarios used to derive the models (i.e., scenarios collected from users) but also events not observed in those scenarios. To test our hypothesis, we performed a case study with five free Android apps and unlocked/rooted Nexus 7 [107] Axus tablets each having a 1.5GHz Qualcomm Snapdragon S4 Pro CPU and equipped with Android 4.4.2 (kernel version 3.4.0-gac9222c).

We measured statement coverage of sequences generated with (i) MONKEYLAB, (ii) a random-based approach (i.e., `Android GUI monkey`), (iii) GUI ripping using a DFS heuristic, and (iv) manual execution. We also compared the GUI events in the sequences to identify events that were executed only by one approach when compared to another (e.g., GUI events in sequences from MonkeyLab but not in the users' traces). In summary, the study aimed at investigating the following research questions (RQs):

RQ₁: *Which language model-based strategy is more suitable for generating effective (un)natural scenarios?*

RQ₂: *Do scenarios generated with MONKEYLAB achieve higher coverage as compared to `Android UI monkey`?*

RQ₃: *How do scenarios generated with MONKEYLAB compare to a DFS-based GUI ripper in terms of coverage?*

RQ₄: *Do MONKEYLAB scenarios achieve higher coverage than manual executions used to train the models?*

4.3.1 Data Collection

Tab. 4.1 lists the Android apps used in the study, lines of code (excluding third party libraries), number of Activities, methods, and GUI components. We selected the appli-

cations looking for a diverse set in terms of events supported by the apps (`click`, `long click`, `swipe`), number of GUI components, number of Activities, and application domain. However, the main selection criteria was the app’s size. We decided to use only medium-to-large applications that exhibit non-trivial use-cases and a large set of feasible events.

To answer the research questions, we simulated a system-level testing process where we asked graduate students to get familiarized with the apps for five minutes (exploratory testing), and then to test the app executing multiple scenarios for 15 minutes (functional testing). Five Ph.D. students at the College of William and Mary executed the *Record* phase of our MONKEYLAB framework. Afterward, we executed the *Mine* \rightarrow *Generate* \rightarrow *Validate* phases in MONKEYLAB to generate actionable sequences for each app. During the *Validate* phase, we collected coverage measurements for each of the generated event sequences by using a tailored version of Emma [18] for Android. Tab. 4.1 lists the total number of low-level and GUI-level events collected from the participants.

To represent a random approach, we generated sequences of touch events with `Android GUI monkey`, simulating test cases composed of random events. Regarding the GUI ripping approach, we implemented our own version of DFS-based exploration. In terms of GUI-model extraction/inference, it considers cases that were not captured in other tools [175, 68], such as pop-up windows in menus/internal windows, on-screen keyboard, and containers. In this case, we do not have a set of actionable sequences because the app is explored trying to visit as many clickable GUI components as possible. However, we measured the accumulated coverage for the whole systematic exploration. This measurement helped us to establish a baseline to validate whether our event sequence generation approach outperforms competitive approaches. More details of the DFS implementation are in our online appendix.

Table 4.1: Android Apps Used in our Study. The stats include the number of activities, methods, and GUI components. Last two columns list the number of raw events ($\#RE$) in the event logs (i.e., lines in the files collected with the `getevent` command) and GUI level events mined from the raw logs ($\#GE$)

App	Ver.	LOC	#Act.	#M.	#Comp.	#RE	#GE
<i>Car Report</i>	2.9.1	7K+	6	764	142	23.4K+	1.5K+
<i>GnuCash</i>	1.5.3	10K+	6	1,027	275	14.7K+	895
<i>Mileage</i>	3.1.1	10K+	51	1,139	99	9.8K+	783
<i>My Expenses</i>	2.4.0	24K+	17	1778	693	20.3K+	854
<i>Tasks</i>	1.0.12	10K+	4	561	200	70.6K+	1.7K+

4.3.2 Design Space

For training the language models, we used the traces collected from the participants for the functional testing scenario. Given two LMs (i.e., INTERP and B0) and three flavors (i.e., up, down, strange), we generated 100 scenarios for each combination $\langle Model, Flavor, APP \rangle$ with 3-gram LMs, then we generated actionable scenarios using the serial mode. For the interactive mode, we generated 33 scenarios for each of the three INTERP flavors. Each scenario (in serial and interactive modes) was composed of 100 events. We also executed Android UI monkey on each app 100 times with 100 touchable events and inter-arrival delay of 500ms. Finally, we executed our DFS-based GUI ripper on each app.

4.4 Results

Figures 4.2 to 4.6 show cumulative coverage of MONKEYLAB and Android UI Monkey for the event sequences, in addition to coverage achieved by logs collected from human participants and coverage of the DFS-based exploration for the analyzed apps. In the case of MONKEYLAB, the figures depict the accumulated coverage of the best LM in serial mode (in red color), and the combination of the 99 scenarios of INTER-up, INTER-down, and INTER-strange (33 scenarios for each strategy) in interactive mode (*I-LM*). In the following subsections, we present the results for the research questions defined in Section ??.

Table 4.2: Accumulated Statement Coverage of the LMs

App	INTERP			BO			<i>I-LM</i>
	up	down	str.	up	down	str.	
<i>Car Report</i>	12%	14%	13%	11%	11%	11%	30%
<i>GnuCash</i>	7%	6%	7%	7%	7%	7%	22%
<i>Mileage</i>	36%	10%	25%	32%	24%	24%	26%
<i>My Expenses</i>	10%	10%	10%	10%	10%	10%	26%
<i>Tasks</i>	40%	34%	35%	22%	22%	34%	42%

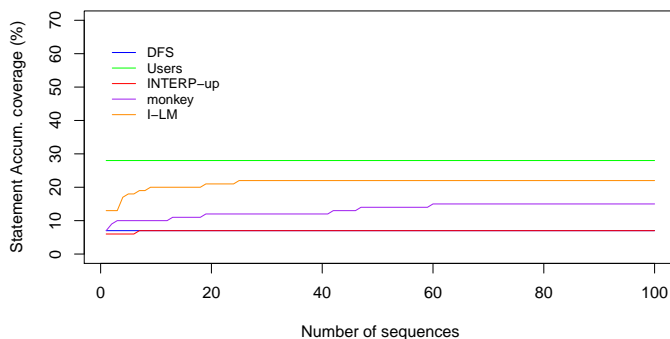


Figure 4.2: Accumulated Coverage for GnuCash

4.4.1 RQ₁: Language Models and Flavors

The accumulated coverage values for the language models in serial mode are listed in Table 4.2. In general, the INTERP-up flavor is the one with the highest accumulated coverage for the serial mode. Apps with splash-screens and modal dialogs in clean launches (i.e., the app is launched right after being installed) such as *GnuCash*, *Car Report*, and *My Expenses* posed a challenge for the language models (see Section 4.4.5 for more details). The interactive mode (*I-LM*), outperforms the language models in serial model in 4 out of the 5 apps. The *I-LM* mode is able to deal with the problem of splash-screens and modal dialogs in clean launches to reduce the notoriously high rate of infeasible events.

The accumulated coverage provides a high-level measure when comparing two different models, however, it does not reveal the entire story. It is possible that different strategies are generating different events and sequences that drive to different executions of the apps. In fact, our goal behind proposing six different flavors of LMs and the interactive mode is based on the possibility that each flavor can explore different regions in the GUI

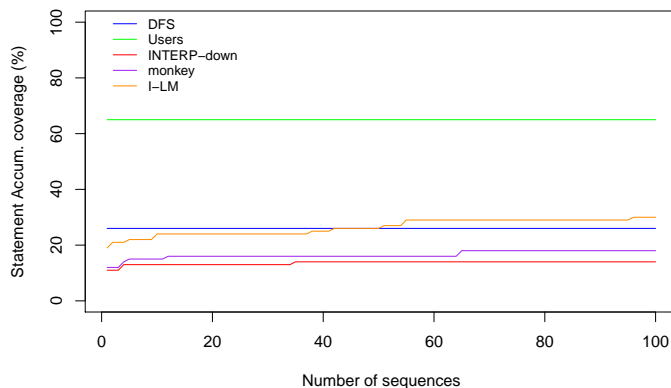


Figure 4.3: Accumulated Coverage for CarReport

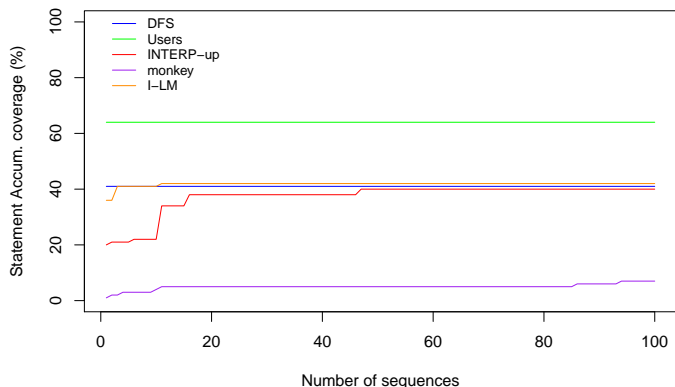


Figure 4.4: Accumulated Coverage for Tasks

event space. Therefore, we investigated the mutually exclusive events executed by each strategy in the five apps. We also measured the number of times the method coverage for each method in an app was higher in one strategy when doing pairwise comparisons (e.g., INTER-up versus INTER-down). The results are presented with heat-maps in Figure 4.7 and Figure 4.8. Both figures corroborate the fact that each LM strategy is able to generate different sets of events. For example, when comparing INTERP-strange to BO-down (Figure 4.7), the former strategy was able to generate 327 GUI events that were not generated by the latter. In addition, the *I-LM* mode is the strategy with the highest difference of executed events when compared to the other strategies.

Answer to RQ₁. The LMs strategies are able to generate diverse and orthogonal sets of GUI events, and the interactive mode is the most effective.

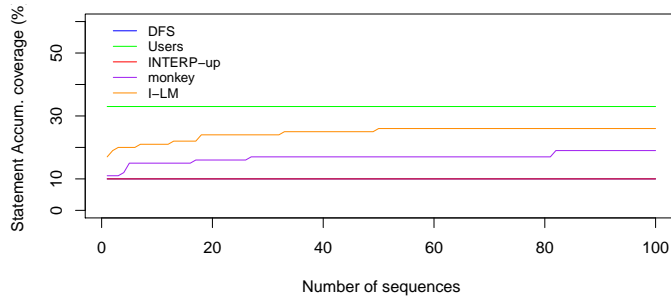


Figure 4.5: Accumulated Coverage for MyExpenses

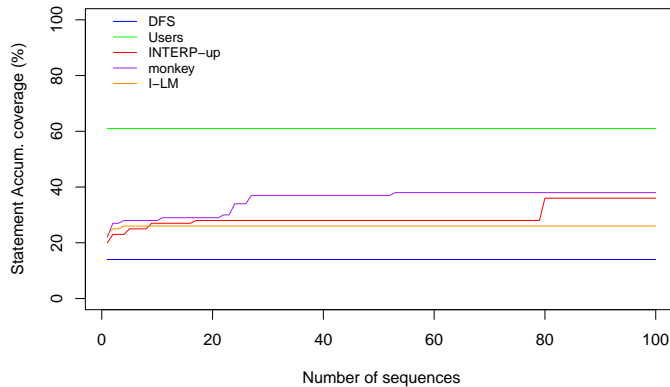


Figure 4.6: Accumulated Coverage for Mileage

4.4.2 RQ₂: MONKEYLAB vs. Android UI monkey

The coverage of UI monkey surprisingly outperformed the LMs in serial model for *Gnu Cash*, *Car Report*, and *MyExpenses*. LMs provided a better coverage only in *Tasks* and showed similar coverage in *Mileage*. The random nature of the events generated by the UI monkey allows this strategy to execute more GUI components without deeply exploring execution paths that are related to the use cases. For instance, UI Monkey is able to click on a diverse set of GUI components; however, it does so without context awareness of the execution history. The LMs are able to explore execution paths that lead to the activation of Activities and features that can not be reached easily by the UI Monkey. However, the *I-LM* mode outperforms the accumulated coverage of UI monkey in all the apps except for *Mileage*. When looking into the results in Figure 4.8, it is clear how UI monkey is able to achieve higher coverage in source code methods more times than the serial LM-strategies.

		109	143	243	327	297	1919	1323	BO-down	Strategy B
14			61	209	284	251	1912	1315	BO-strange	
21	34			187	264	235	1912	1316	BO-up	
150	211	216			151	140	1917	1330	Interp-down	
99	151	158	16			33	1886	1254	Interp-strange	
99	148	159	35	63			1892	1258	Interp-up	
185	273	300	276	380	356			1038	I-LM	
162	249	277	262	321	295		1611		Users	
BO.down	BO.strange	BO.up	Interp.down	Interp.strange	Interp.up	I.LM	Users			

Figure 4.7: Total number of events executed by Strategy A that are not executed by Strategy B. The key color goes from white (zero) to red (highest value)

However, the *I-LM* is able to achieve higher coverage in more source code methods when compared to `Android UI monkey`.

Answer to RQ₂. `Android UI monkey` is able to achieve higher coverage than `MONKEYLAB` in serial model. However, our interactive mode outperforms `UI monkey`. In terms of scenarios, all the `MONKEYLAB` strategies are able to generate execution paths that are not covered by `UI monkey`.

4.4.3 RQ₃: `MONKEYLAB` vs. `DFS`

The LMs in a serial mode provided similar statement coverage as compared to `DFS` in `GnuCash`, `Tasks`, and `MyExpenses`, better coverage in `Mileage`, and lower in `Car Report`. `DFS` follows a systematic strategy for executing click events on the GUIs. This systematic exploration is able to exercise deeper executions paths as compared to `UI monkey`, but `DFS` is not able to recognize data dependencies. We found that the LMs can actually model some of these dependencies (e.g., some fields are required to create a budget or task). In

		1046	1875	1041	836	959	882	843	944	1903	Users
3842		2407	1255	974	1143	1117	1011	1081	2561	DFS	
3357	1052		1058	958	1077	935	959	1024	2254	Monkey	
3850	1206	2314		505	421	337	371	360	2574	Interp-up	
3965	1281	2593	841		607	719	633	479	2783	Interp-down	
3914	1253	2459	582	412		450	290	194	2598	Interp-strange	
3965	1358	2471	634	684	450		185	540	2811	BO-up	
3997	1351	2601	756	687	540	262		376	2746	BO-down	
3926	1252	2486	548	345	257	421	174		2610	BO-strange	
3235	1035	2044	1066	900	969	1028	877	895		I-LM	
Users	DFS	Monkey	Interp.up	Interp.down	Interp.strange	BO.up	BO.down	BO.strange	I.LM		Strategy A
											Strategy B

Figure 4.8: Total number of source code methods in which coverage is higher when comparing coverage of Strategy A versus Strategy B

other cases, the LMs can get stuck in specific paths that are very common (frequent) in the observed traces. However, the *I-LM* mode outperformed the accumulated coverage of DFS in the 5 apps. Concerning, the source code methods with higher coverage (Figure 4.8), there is a notorious difference between DFS and the LMs in serial mode. However, the *I-LM* provides higher coverage on more methods when compared to DFS ($I-LM - DFS = 2,561$ and $DFS - I-LM = 1,035$ methods).

Summary for RQ₃: Although the LMs in serial mode are not able to achieve better coverage than DFS, *I-LM* is able to achieve higher coverage as compared to DFS.

4.4.4 RQ₄: MONKEYLAB vs. Manual execution

The coverage achieved by the users during the Record phase outperforms UI Monkey, DFS, and the LMs in serial and interactive modes, as originally expected. However, the

LMS were able to execute events that were not observed in the event traces collected by the users (see Figure 4.7). In general, the LMS were able to generate actionable scenarios for natural executions, i.e., there are sequences generated with the LMS including GUI events from the collected traces. However, the LMS were also able to generate sequences with unseen events, which means that the actionable scenarios cover natural cases but also corner-cases not considered by the users. And the benefit is noticeable when using the interactive mode, *which was able to generate 1,611 GUI events not considered by the participants in our study.*

While we can not claim that the LMS are better than the other approaches in terms of coverage, after looking into the details of the events executed by each method, it becomes clear that the combinations of manual testing and automated approaches can significantly improve the coverage. However, MONKEYLAB is able to learn a model and then generate actionable scenarios that can be executed on a target device. In addition, MONKEYLAB generates not only natural scenarios but also corner cases that are not generated by any of the competitive approaches.

Answer to RQ₄: Although the overall coverage achieved by MONKEYLAB is not as high as compared to manual execution, MONKEYLAB is able to generate actionable scenarios including not only natural GUI events, but also events that are not considered during manual testing. Therefore, the scenarios generated by MONKEYLAB can help increase the coverage achieved by manual testing without extra effort that is imposed by collecting test-scripts.

4.4.5 Limitations

Automated GUI-based testing approaches can benefit from the following information in this study, since some of these issues can pose similar problems for various testing *strategies*.

Encapsulated components. Encapsulated components (e.g., `KeyboardView`, `AutoCompleteTextView`, `CalendarView`, and `DatePicker`) cannot be analyzed during system-

atic exploration, because the sub-components (e.g., each key of the keyboard) are not available for ripping at execution time. Thus, the sub-components are not recognized by the `View Server` nor the `Hierarchy Viewer`. This type of component requires a predefined model of the feasible events with the locations and areas of these sub-components.

Training corpus size in serial mode. Another lesson gleaned from this study concerns an effective training set. Specifically, we diagnosed two critical issues that contributed to the high rate of infeasible events produced by the interpolated n -grams and back-off models. The principal issue was relatively small amount of training data for each app as compared to the size of the respective vocabularies. The need for substantial training data was observed in most of the sequences for the apps (especially *Keepscore*) for which the LM-based strategies were not able to produce a response. For example, an app with a vocabulary size of 250 events would require 250×249 and $250 \times 250 \times 249$ parameters to reliably estimate bigram and trigram models, respectively. Data sparsity is a key concern in statistical language modeling, and a number of techniques have been developed to manage the problem of deriving useful statistical estimates from relatively small corpora [79]. Naturally, one approach to controlling the number of parameters is to reduce the order of the model [181]; however, reducing the order of our models presented another issue. In our experiments, reducing the order degenerated the effectiveness of the `INTERP-up` models at generating feasible test cases. This degeneration can be attributed to allocating too much weight to the unigrams in the mixture model (Equation 4.3). In other words, after unfolding the recurrence in Equation 4.3, unigrams will generally have more influence on the predictions of interpolated n -gram models as compared to interpolated models whose order is greater than n . For example, unigrams will have more influence on the predictions of interpolated trigrams than interpolated 4-grams, because the unigrams are multiplied by $\beta(h)$ (Equation 4.3), a number less than one, an additional time in interpolated 4-grams. Our expectation was for `INTERP-up` models to yield favorable coverage results in line with previous empirical studies [227], yet the low-orders disabled the models in our experiments. However, we did not expect `INTERP-down` nor

INTERP-*strange* models to yield high rates of feasible events. Considering the nature of the interpolation smoothing technique, models at *all* orders are factored into the estimator. So, regardless of the order of the interpolated model and the history of events, every event will have a chance to be selected at each point in time. Moreover, for INTERP-*down* and INTERP-*strange*, the probability distributions are transformed such that *all* of the unlikely events (given the context) in the vocabulary are generally assigned more probability mass. Thus, we expected INTERP-*down* and INTERP-*strange* models to yield high rates of infeasible events. The sparse training set had a similar degenerative effect on the back-off models. With relatively few data points to inform high-order component models, the back-off model will successively probe low-order models until reaching unigrams—the base case. Probing low-order models is not conducive to generating feasible events. The purpose of smoothing techniques like interpolation and back-off is to use substantial contexts when possible before defaulting to lower-order models (e.g., unigrams and bigrams), but this is predicated on a training set that can support the estimation of higher-order models.

4.5 Discussion

The main contribution of our work is a novel *Record*→*Mine*→*Generate*→*Validate* framework for the automatic generation of actionable scenarios for Android apps. The framework relies on mining real app usages, static analysis of source code, and dynamic analysis to translate low level events and GUI components to a high-level representation that allows event sequences decoupled from specific locations and device sizes. The framework is supported on a modular architecture that allows easy extension (more details about the framework extensibility in the final Chapter 6) and multiple implementations depending on the purpose of the developers. One instance of the framework is MONKEY-LAB , which generates (un)natural event sequences. The intention with MONKEYLAB is to generate sequences that are not often exercised by developers/testers; to this, MON-

KEYLAB uses statistical properties of two language models and three proposed flavors of the language models to generate the sequences. For the evaluation, we used several medium-to-large Android apps. On one hand, the results suggest that the B0 model is more suitable for generating unnatural scenarios with small and large corpora. On the other hand, INTERP is not able to generate scenarios when the available corpora are small. If small corpora are available, INTERP is more suitable for natural scenarios. Finally, the interactive mode in MONKEYLAB outperformed the serial mode in terms of coverage. The evaluation demonstrated the ability of MONKEYLAB to generate (un)natural sequences not exercised by the developers. It is worth noting that the testing goal of MONKEYLAB is not coverage; therefore, MONKEYLAB can be used by development teams as an automatic and complementary approach to manual testing.

We also provide a set of learned lessons for GUI-based testing of Android apps using actionable scenarios. The lessons recommend designing models for encapsulated components and increasing training corpus size of LMs. The lessons can be used by researchers to implement automated approaches for GUI-based testing that are more attractive and useful to mobile developers and testers. Actionable scenarios can also be combined with automatic generation of testing oracles [159] for generating GUI-based test cases for mobile apps.

4.6 Bibliographics Notes

The papers supporting the content described in this Chapter were done in collaboration with other members of the SEMERU group at William and Mary:

- **Linares-Vásquez, M.**, “Enabling Testing of Android Apps”, in Proceedings 37th IEEE/ACM International Conference on Software Engineering (ICSE’15), ACM Student Research Competition, Florence, Italy, May 16-24, 2015, pp. 763-765. **First Place Graduate Winner ACM Student Research Competition.**

- **Linares-Vásquez, M.**, White, M., Bernal-Cárdenas, C., Moran, K., and Poshyvanyk, D., “Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios”, in Proceedings of the 12th IEEE Working Conference on Mining Software Repositories (MSR’15), Florence, Italy, May 16-17, 2015, pp. 111-122 (30% acceptance ratio)

Chapter 5

Visual Aesthetics Matter: Multi-Objective Optimization of Energy Consumption of GUIs in Android Apps

Nowadays, an impressive amount of our everyday activities is supported by apps running on mobile devices: A calendar app reminds us of that important meeting we have to attend, the GPS navigator recommends the best route to reach our destination, and we take and share meeting's notes by using our favourite note app. This impressive adoption of mobile devices and apps in daily activities has motivated the need for reducing their energy consumption. For instance, common energy bugs in mobile apps have been identified and catalogued [204, 207, 172, 244, 173], as well as typical hot spots [235] together with energy greedy APIs [205, 165]. In addition, several infrastructures and methods have been proposed to measure and estimate the energy consumption of mobile devices and apps [130, 122, 183, 152].

Some practices for avoiding and fixing energy hotspots (bugs) in mobile apps focus on how the apps should use energy-greedy hardware components in the device, such as

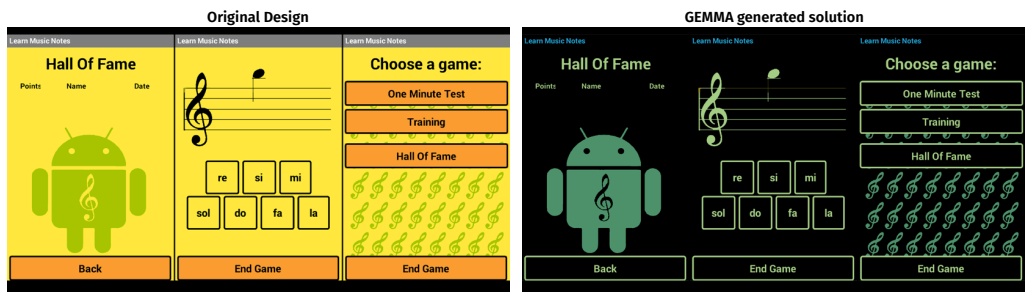


Figure 5.1: Original design *vs* GEMMA’s solutions for the Learn Music Notes app. Each solution shows the color composition for three GUIs in the app.

GPS, Wi-Fi, or the screen. However, despite using good programming practices targeting energy greedy components, programming practices have a limit in terms of energy saving, and the limit is imposed by the hardware components. For instance, in the case of LCD displays, the energy drawn from the battery is constant regardless of the colors displayed on the screen. This is not the case for OLED displays, for which the energy consumption depends on the combinations of colors at the sub-pixel level. This property of OLED displays motivated the adoption of power models for estimating the energy drawn by the graphical user interfaces (GUI) displayed on the screen. In fact, previous work have used power models to estimate and improve the energy consumption of web browsers [90], mobile web apps [154], and mobile apps in general [91, 235].

All these prior techniques on improving energy consumption of GUIs running on OLED devices have been mostly driven by a single objective to reduce energy consumption at all costs. Also, these approaches generally exploit only a small subset of possible color schemas that developers/designers can define to optimize the GUI. For example, using predefined themes or using color transformations starting only from the original scheme can drastically reduce the color palettes and compositions that can be explored as a potential solution [91]. This is also the case when the proposed solution uses by default dark colors in the background [154, 235]. Another issue with previous approaches is that the solutions are generated individually for each GUI in the app, which can lead to

inconsistencies in the whole design concept [154, 235].

In this Chapter we propose a multi-objective approach, namely GEMMA¹, for generating color compositions that reduce the energy consumption of GUIs in Android apps and are visually attractive at the same time. GEMMA combines power models, pixel-based engineering, color theory, dynamic analysis, and a multi-objective optimization technique—namely Non-dominated Sorting Genetic Algorithm (NSGA)-II [88]—to produce a Pareto-optimal set of design solutions (*i.e.*, GUI color compositions) across three different objectives: (i) reducing energy consumption, (ii) increasing contrast, and (iii) improving the attractiveness of the chosen colors by keeping the palette close to the original one. GEMMA is supported by a solution that relies on an infrastructure for large-scale execution of Android apps described in Chapter 3.

To the best of our knowledge GEMMA introduces the following unique advantages and contributions:

- *Is multi-objective.* This is the first approach adopting multi-objective optimization to choose colors for mobile apps with the main goal of reducing energy consumption. Multi-objective optimization avoids a direct aggregation of different, potentially conflicting objectives, and, as it will be shown in our empirical evaluation, it allows developers to evaluate different possible solutions that optimize specific objectives or achieve a compromise between two or three of them;
- *Accounts for multiple screens and the duration of time they are displayed.* Since a GUI consists of multiple screens, it is particularly important to optimize colors for screens that are used longer in typical usage scenarios, while screens that just appear for few moments may have lesser contribution to the overall energy consumption. This is accounted for in the GEMMA’s energy objective function;
- *Produces pleasant and consistent color combinations.* In addition to energy consumption and contrast, this is the first approach taking GUI design principles—at

¹Gui Energy Multi-objective optiMization for Android apps

least in terms of choosing colors—into account. First, for the color palette generation we combine the original color scheme in an app with three models based on color theory harmonies. Then, GEMMA generates compositions such that if two or more components have a given color (say yellow) in the original palette, they will appear with the same color (say green) in the proposed solutions, so to avoid producing inconsistent GUIs. Finally, as mentioned before, the multi-objective optimization has a third, “design-related” objective aimed at choosing colors that are not too distant from the original ones. To have an idea of what does this mean in terms of generated color compositions, Figure 5.1 shows the original design and an example of a GUI automatically generated by GEMMA for the Learn Music Notes app.

To validate the quality of the color schemas recommended by GEMMA, we used it to generate optimized GUIs for 27 free apps available in Google Play and for five commercial apps developed by Italian companies. The colorfulness of the GUIs derived for 27 of those apps was evaluated by 104 mobile apps users in an online survey; also, since GEMMA is implemented in a publicly available webapp, we asked developers and managers of three Italian companies to use it on their apps and provide us feedback concerning the GUIs generated by GEMMA for their apps and the usefulness of our tool. Finally, we used a power monitor-based infrastructure to measure the actual energy saved by adopting the GUIs generated by GEMMA.

Overall, GEMMA has been evaluated across four dimensions: (i) the ability to optimize the three objectives, (ii) the actual energy saved by adopting the generated GUIs, (iii) the extent to which potential users consider the choices of colors acceptable enough, and (iiii) the extent to which the original developers of commercial apps would be willing to account for GEMMA’s recommendations.

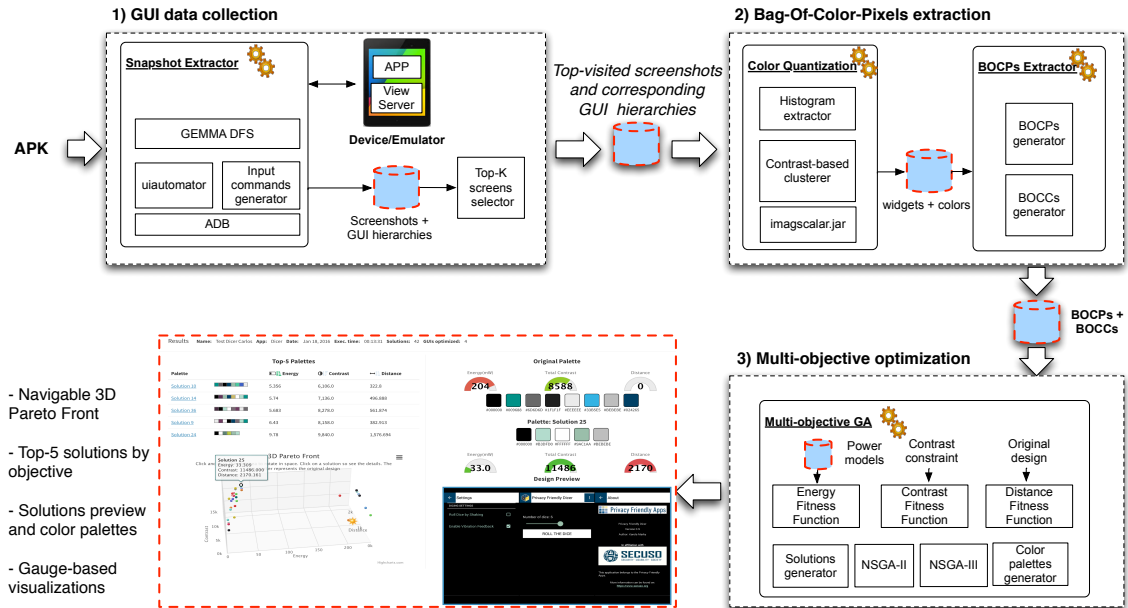


Figure 5.2: The GEMMA approach and components

5.1 Optimizing Energy Consumption of GUIs with GEMMA

This section describes GEMMA, a multi-objective search-based optimization technique that suggests possible color alternatives achieving a tradeoff between three desirable and possibly conflicting objectives: (i) reducing energy consumption on OLED displays; (ii) increasing contrast between adjacent GUI elements; and (iii) preserving a consistency in the color usage with respect to the original design (*i.e.*, proposing color schemas that are close to the one adopted by developers in the original app’s GUI). In addition to nearly optimizing the objectives mentioned above, GEMMA can also verify that the proposed GUIs satisfy some specified constraints. For instance, the current implementation of GEMMA does not consider GUIs’ color schemas as valid in which adjacent elements have the same color or colors having a very low contrast between them, in order to promote the readability of the recommended GUIs.

To solve the optimization problem, first, we obtain a power model of the corresponding display, relating pixel colors to power consumption (Section 5.1.1). Then, we identify

GUI elements composing each screen (Section 5.1.2). Finally, we define a multi objective Genetic Algorithm (GA) aimed at finding near-optimal solutions to the stated problem (Section 5.1.3). The GEMMA approach and the underlying components are depicted in Figure 5.2. Each one of the components and phases in the GEMMA approach are described in the following subsections.

5.1.1 Estimating the Power Consumption

The power consumption of OLED screens can be estimated by using the color components of each pixel [91, 90, 154, 235]. According to the specific pixel matrix of the screen, each color in a pixel is rendered by using a combination of *red*, *green*, and *blue* sub-pixels at different levels (*a.k.a.*, intensity) in the standard RGB (sRGB) color model. In the case of OLED screens the power consumption of sub-pixels depends on the color component level. The power consumption of an OLED pixel is a linear function of its linear RGB values, *i.e.*, the gamma decoding result from its values in the standard RGB space [91, 90]. Therefore, given a power profile $P_{\langle color \rangle}(level)$ that returns the power consumption of a color level, the power consumption of a *pixel* in the position $\langle x, y \rangle$ with $color_{x,y}$ and components $R_{x,y}$ (red), $G_{x,y}$ (green), and $B_{x,y}$ (blue) is estimated as in Equation 5.1, and the total power consumed by the screen when painting a GUI is the sum of pixel power consumption over all the $X \times Y$ pixels in the screen (Equation 5.2).

$$P(color_{x,y}) = P_{\langle R \rangle}(R_{x,y}) + P_{\langle G \rangle}(G_{x,y}) + P_{\langle B \rangle}(B_{x,y}) \quad (5.1)$$

$$TP(GUI) = \sum_{x=1}^X \sum_{y=1}^Y P(color_{x,y}) \quad (5.2)$$

The power profile of the color components represented by the $P_{\langle color \rangle}$ functions is screen specific. In fact, this characteristic of the OLED screen has been exploited before to build power models that estimate the consumption of GUIs [91, 90, 154, 235]. In our case we built the power model for a SUPER AMOLED screen (1080×1920 pixels) in a Samsung

Galaxy S4, by using a monsoon power monitor [193]. To define each $P_{\langle color \rangle}$ function, we measured the current drawn by the screen when painting it with all the pixels set to black (*i.e.*, *idle* period) during 30 seconds, and full screen with all the pixels set to the same primary color component for each $level \in [1, 255]$ during 30 seconds (*i.e.*, *measurement* period). We are not considering the effect of the screen brightness, consequently, we set the display brightness to the maximum value. For both, idle and measurement periods we computed the average current of a pixel dividing the raw value, measured with the power monitor, by the total number of pixels in the screen (1080×1920). After collecting all the values, we subtracted the current in $idle_k$ from the current in the corresponding $measurement_k$ to account only for the current drawn by the screen pixels. Finally, we removed noise by using the Tukey’s smoother implemented in *R* [228]. The measurements after smoothing are presented in Figure 5.3.

To derive the power model, we followed the same procedure previously adopted in [91, 90]. With the 255 measurements representing the power profile of a primary color in sRGB (Fig. 5.3), we transformed the intensities to linear RGB, and then we found the linear function describing the power consumption (Figure 5.4). The linear RGB simplifies the construction of the model without loss of information [90]. With the power models using as input the color intensities in linear RGB, it is possible to estimate the current drawn by all the pixels in the Galaxy S4 AMOLED screen when a specific GUI is painted, by using Equation 5.2. The model in Figure 5.4 shows current measurements in mA (milli Amperes); the current is proportional to power (Watts) because the voltage is constant during the measurements; thus, there is no need to plot the results in terms of Watts. These measurements are consistent with the ones by Dong *et al.* [90, 91] for OLED screens.

5.1.2 Extracting Color Composition from GUIs

Using a power model to estimate the power consumption of GUIs requires extracting pixel-based representations of the GUIs. This extraction can be done online when the target GUIs are displayed on the device’s screen [90], or offline by relying on screenshots [235] or

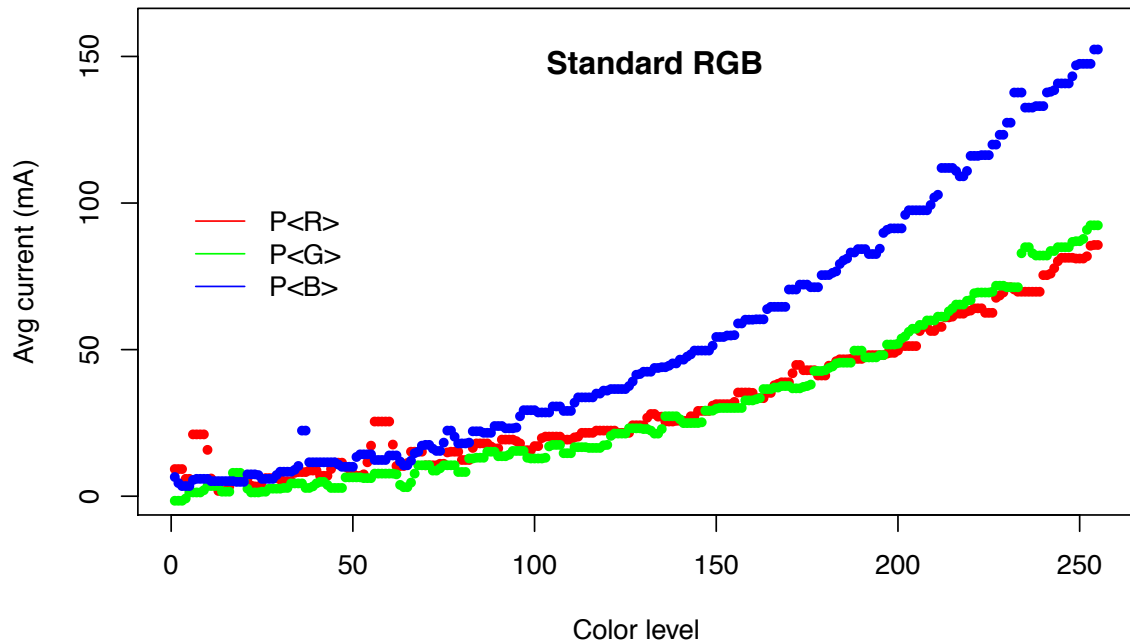


Figure 5.3: Current (mA) consumption models for primary colors of the Galaxy S4 SUPER AMOLED screen in standard RGB color space.

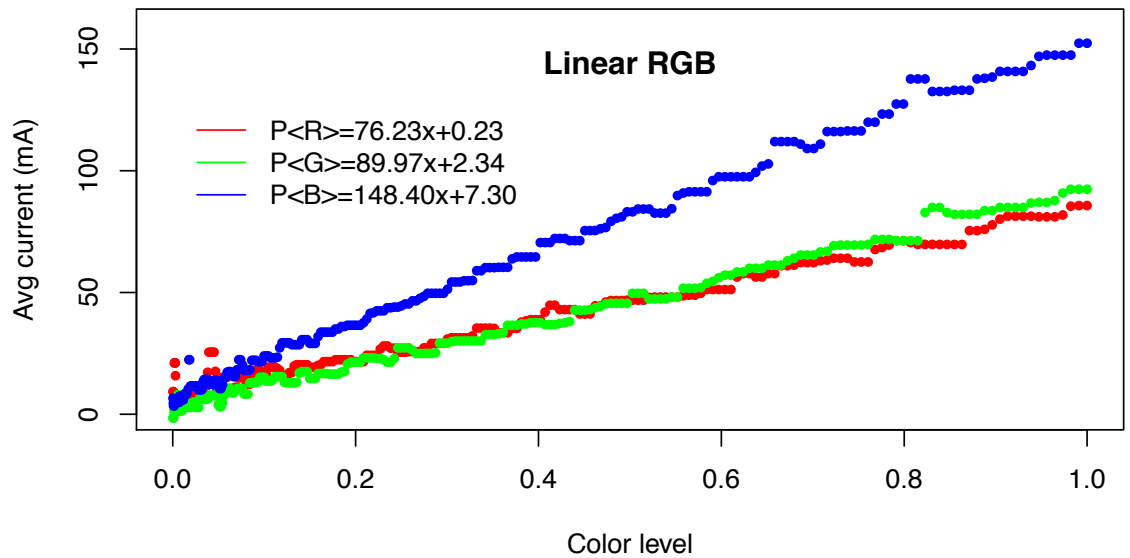


Figure 5.4: Current (mA) consumption models for primary colors of the Galaxy S4 SUPER AMOLED screen in linear RGB color space.

server-side code[154]. The representation is usually the GUI-based color histogram or a set of tuples $\langle \text{pixel}, \text{red-level}, \text{green-level}, \text{blue-level} \rangle$. GEMMA relies on dynamic analysis of an app to extract GUI-based information, since the GUI is shown on the device (or emulator), and the Android *View Server*² running on the device can be queried to collect features of the GUI components such as location, dimensions, text, and component’s hierarchy. With this information available, the extraction can be focused on detecting the color scheme (*i.e.*, individual colors and composition) instead of detecting shapes representing GUI components. Moreover, the information provided by the *View Server* allows us to distinguish components with images, components with/without text, and containers. This is particularly important to avoid transformation/distortion of images/logos that belong to the GUI or even to third-parties. GEMMA does not derive color schemes for images/logos in the GUI; the color schemes are generated only for backgrounds, borders, and fonts of Android GUI components. In this sense, we do not expect significant reductions in the power consumption of GUIs composed mostly of images. Also GEMMA is aimed only at getting pixel-based representation for native-apps, conversely to [154] that focuses on web apps for mobile devices.

In terms of representation, GEMMA extracts bag-of-color-pixels (BOCP) and bag-of-color-components (BOCC) of Android app GUIs. Both BOCP and BOCC are hash-map structures that use colors as keys. An entry in a BOCP is a collection of pixels with the same color in the GUI, and an entry in a BOCC is a collection of components having pixels with the same color in the GUI. The BOCP/BOCC extraction procedure starts with getting a snapshot of the current GUI in the device, and the components in the GUI, by querying the *View Server* with the *UI Automator utility* in the Android SDK. The snapshot contains an XML-based representation of the GUI hierarchy (including location, type, dimensions, and text for each component), and a screenshot of the GUI. The whole procedure is described with Algorithm 3. It is worth noting that our BOCP/BOCC

²The Android *View Server* is one of the services available in the Android OS. Therefore, it is always available on Android devices (or emulators) and can be easily queried with utilities from the Android framework such as `uiautomator` and `hierarchyviewer`.

Algorithm 3: Extracting BOCP and BOCC from a targeted set of GUIs in an Android native app

Input: $k, r, GUIS$
Output: $BOCP, BOCC$

```
1 begin
2    $BOCP = \emptyset, BOCC = \emptyset;$ 
3   foreach  $GUI \in GUIS$  do
4      $S = getUISnapshot(GUI);$ 
5      $C = getComponentsFromSnapshot(S);$ 
6     foreach  $c \in C$  do
7       if  $c \neq image$  then
8          $pixels_c = getPixels(S, c);$ 
9          $hist_c = getPixelHistogram(pixels_c);$ 
10         $hist_c = sort(hist_c);$ 
11         $medoids = getMedoids(hist_c, k, r);$ 
12        foreach  $pixel \in pixels_c$  do
13           $color = getColor(pixel);$ 
14           $color^* = getClosest(color, medoids);$ 
15           $add(BOCP[color^*], pixel);$ 
16           $add(BOCC[color], c);$ 
```

extraction procedure enables consistency of the generated solutions (i.e., the color composition is the same for all the GUIs), because the BOCP/BOCC represent the GUIs as a whole entity instead of individual screens.

To collect the snapshots, GEMMA was designed to support two modes: (i) a manual mode in which the developer can select directly on the device the GUIs to analyze, and (ii) an automatic ripping mode in which the app is explored systematically using a Depth-First-Search (DFS) strategy. Our rationale for the two modes is that while the manual mode requires the developer to have a device and ask GEMMA locally to analyze the GUIs, the automatic mode allows for a cloud-based or web-based approach in which the developer only has to submit an APK to the GEMMA system. Regarding the automatic mode, it executes click events on all the clickable components and iteratively builds a tree of windows, components and transitions; the exploration is performed until all the

Algorithm 4: Color medoids detection

Input: $hist_c, k, r$
Output: $medoids$

```
1 begin
2    $medoids = \emptyset;$ 
3    $medoids[1] = hist_c[1];$ 
4    $index = 2, medoidIndex = 1,;$ 
5   while  $|medoids| \leq k \ \&\& \ index \leq |hist_c|$  do
6      $color = hist_c[index];$ 
7     if  $Lum(color, medoids[medoidIndex]) \geq r$  then
8        $medoidIndex ++;$ 
9        $medoids[medoidIndex] = color;$ 
10     $index ++;$ 
```

clickable components (discovered during the exploration) have been exercised³. After the automatic exploration, GEMMA selects the snapshots of the most visited GUIs. The number of top-visited GUIs is selected a-priori by the developer when submitting the APK. It is worth noting that the automatic selection of the most visited GUIs is only enabled in the automatic mode.

Once the GUIs-to-analyze have been selected (manually or automatically), the corresponding screenshots and GUI hierarchies are used to identify the representative colors of each component in all the GUIs. To identify the colors in the components, we used a clustering-based quantization on the color histogram of each component. Only components different from `ImageView` and `ImageButton` were considered. Pixels belonging to each component are detected by traversing the GUI hierarchy in a bottom-up fashion, *i.e.*, deepest level is processed first. A pixel in the screen is assigned to only one component, and the priority for this assignment is based on the depth of the GUI hierarchy. These heuristics allow assigning pixels according to the z-index in the GUI; if there are components inside a container, the pixels are first assigned to the components, and then, the leftover pixels are assigned to the container. The set of pixels assigned to a

³DFS-based exploration is a widely used strategy for automated ripping, execution, and testing of mobile apps [55, 198, 171, 194, 195]

component/container are then used to derive a color histogram.

Top- k colors (*i.e.*, medoids) are detected in the histogram of each component while avoiding gradients and font shadows that are automatically injected by Android. For that, GEMMA traverses the sorted histogram—from high to low frequency—looking for changes in the contrast ratio higher than a threshold r as described in Algorithm 4. As for the contrast ratio computation between two colors a and b , we used the luminance-based contrast ratio (*a.k.a.*, relative luminance) [13] $Lum(a, b)$ that better accounts for differences between text and background colors:

$$T(level) = \begin{cases} \frac{level}{12.92} & \text{if } level \leq 0.03928 \\ \left(\frac{level+0.055}{1.055}\right)^{2.4} & \text{otherwise} \end{cases} \quad (5.3)$$

$$L(a) = 0.216 \cdot T\left(\frac{red_a}{255}\right) + 0.7152 \cdot T\left(\frac{green_a}{255}\right) + 0.0722 \cdot T\left(\frac{blue_a}{255}\right) \quad (5.4)$$

$$Lum(a, b) = \begin{cases} \frac{L(a)+0.05}{L(b)+0.05} & \text{if } L(a) > L(b) \\ \frac{L(b)+0.05}{L(a)+0.05} & \text{otherwise} \end{cases} \quad (5.5)$$

To illustrate the contrast-based clusters detection let us assume a text edit component with dark gray gradient (background) and white font; a lot of “grays” will surface at the beginning of the sorted histogram, and then white will appear. However, the contrast ratio between the grays is going to be low (close to one), and the ratio between grays and white is going to be higher. Thus, the top-1 color in the histogram is the medoid (*i.e.*, the most frequent color in a cluster) for the first cluster (*i.e.*, the most frequent gray in the histogram), then a new cluster (*i.e.*, text) is detected when the contrast ratio is higher than r . This procedure is applied until k medoids are detected. The values we adopted for k and r are reported in Section 5.3.4. Note that our definition of medoid is the most-frequent color in a cluster instead of the average, because averaging the colors

in a cluster might change the characteristics of the colors defined by the developers in the original design.

Finally, the colors of the GUI components are discretized (*i.e.*, color quantization) using the color medoids in the histogram. It means that the pixels are assigned the closest quantized color (*i.e.*, color medoid) to the original pixel color. The BOCP is a hash-map structure for all the pixels in the GUI, in which the key is a quantized color $color^*$ and the value is a list of pixels assigned to that color. The BOCC is a hash-map too, in which the key is a quantized color $color^*$ and the value is a set of the components associated with the pixels in $BOCP[color^*]$. Both structures, BOCP and BOCC allow for a simple and powerful representation of a set of GUIs-to-analyze in the sense that generating a new consistent color composition for the GUIs-to-analyze consists in finding new values for the keys (*i.e.*, values for the $color^*$ keys) in the structures. Therefore, a “new” set of $color^*$ keys is the color palette for a new color composition.

5.1.3 Multi-objective Optimization Model

In the following, we describe the GA component in GEMMA. The values we use for the GA parameters are reported in Section 5.3.4. The GA has been implemented using *jMetal* [94].

Representation. Our representation contains a gene for each key in a BOCP, *i.e.*, each gene will define the color to all the pixels assigned to that key in the BOCP. We indicate a generic solution of our optimization problem as S , and with $S[i]$ we denote the color this solution assigns to the i -th key in a BOCP structure.

Multi-Objective Genetic Algorithm. Rather than evaluating each solution according to a single fitness function, a multi-objective GA produces *frontiers* composed of *Pareto-optimal* solutions. A solution X is said to be *Pareto-optimal* if-and-only-if it is non-dominated by any other solution within the search space, *i.e.*, if-and-only-if no other solution Y exists which would improve one of the objective functions, without worsening other objectives. All the solutions that are not dominated by any other solution are said

to form a *Pareto-optimal set*, while the corresponding *objective vectors* (containing the values of the objective functions) are said to form a *Pareto frontier*. Identifying a Pareto frontier is useful because the software engineer can use the frontier to make a well-informed decision that balances the trade-offs between the different objectives. In our context, one could select color choices achieving the lowest energy consumption, having the highest contrast, being the closest to the original design, or a compromise among these objectives. The specific multi-objective GAs we use for GEMMA is the Non-dominated Sorting Genetic Algorithms NSGA-II [88], which tries to ensure diversity in the evolving populations to avoid the situation where populations have been filled only with dominating solutions (because of the elitism effect, *i.e.*, best solutions are preserved). The key idea behind NSGA-II is to use an elitist selection strategy for replacement (with respect to fitness and spread). For each generation, parents and offsprings are gathered in a mating pool. A fast, non-dominated sorting approach is used to rank these individuals in subsequent Pareto fronts. Then, the next generation is built by preferring the individuals with the lower non-domination ranks. In case of individuals with the same rank, the selection depends on a diversity preservation mechanism. A density measure—*crowding distance*—is computed in the objective space. The tournament used to select parents for reproduction is based on this density measure.

Initial Population Generation. Each individual in the initial population has the same number of genes (*i.e.*, color keys) than in the original BOCP. In order to assign values (*i.e.*, colors) to the genes in the initial individuals, random colors are selected from a color palette composed of 512 colors. The color codification is the single `int` pixel in the `TYPE_INT_ARGB` representation provided by Java. Our color palette generation strategy is not based on color transformations from the original design or predefined palettes (which reduce drastically the universe of color compositions) as previous work do [91, 154, 235]. Our strategy consist on combining colors in the original design with color palettes derived using heuristics from the color theory field. The usage of color composition heuristics allows GEMMA to rely on a set of colors selected to provide visual

harmonies instead of totally random color palettes that might drive to unpleasant GUIs. Therefore, the GEMMA palette includes m colors from the original BOCP, white, black, a palette in equidistant color harmony with $(512 - m - 2)/3$ colors, a monochromatic palette of $(512 - m - 2)/3$ colors with difference in saturation and brightness, and the equidistant harmony palette, but with random saturation and brightness for each of the colors. The starting point of the equidistant harmony and monochromatic palette (*i.e.*, initial values for the tones) are randomly selected from the HSB/HSV color wheel. Our choice for a diverse palette that includes equidistant and monochromatic harmonies is to provide the GA with colors that can drive it to producing visually appealing GUIs.

Color generation techniques explores the color wheel spectrum with different selection heuristics looking for specific angles, symmetries, and value and saturation contrasts, while looking visual appealing composition. For instance, equidistant harmony is the rule for color composition in the HSB/HSV (Hue Saturation Brightness/Value) color space that derives a color palette with the same saturation and brightness, but modulating the tone (*i.e.*, hue) by taking proportional spaces in the HSB/HSV color wheel. Because the tone in the HSB/HSV space varies from 0° to 359° , and each integer value corresponds to one color (*e.g.*, 0° is red, 120° is green, and 240° is blue), equidistant colors can be derived by dividing the hue space in a target number of colors. For instance, an equidistant harmony of 20 colors will have a tone difference of $(360/20)^\circ$ between each color; if the harmony starts at a 10° tone with fixed values S for saturation and B for brightness, the colors in harmony would be $\langle 10^\circ, S, B \rangle, \langle 28^\circ, S, B \rangle, \langle 46^\circ, S, B \rangle, \dots, \langle 334^\circ, S, B \rangle, \langle 352^\circ, S, B \rangle$. Equidistant harmonies can also derive into palettes with value and saturation contrast by changing randomly the saturation and brightness of the colors in the original equidistant harmony (see Figure 5.5-middle). Monochromatic palettes are generated by using the same hue but varying the saturation and brightness randomly for each color in the palette (*e.g.*, 20 colors with the same hue but with different saturation and brightness as in Figure 5.5-bottom). In the three composition schemes we used a brightness range $[0.2, 0.9]$ to avoid colors that are too bright or too dark. Its worth nothing that the starting point for

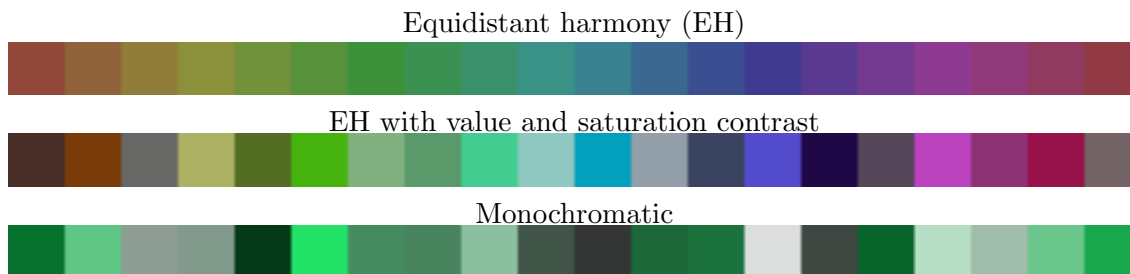


Figure 5.5: Example of palettes (20 colors each) with equidistant harmony (top), equidistant harmony with random saturation and brightness (middle), and monochromatic scale (bottom).

the equidistant harmonies and monochromatic schemes in the color palette for the initial population are generated randomly at the beginning of the GA execution, therefore, each GA execution will have a different color palette.

Genetic Operators. In order to evolve GA individuals, we use *selection*, *crossover* and *mutation* operators. The *crossover* operator is the *one-point* crossover. Given two *parents*, a one-point crossover cuts them—*i.e.*, the array of BOCP—in a random position p , and then all genes after position p are exchanged to produce the offspring. Crossover is applied to individuals of the population with a probability p_{cross} . The *mutation* operator is the uniform bit-flip mutation that, with probability p_{mut} , changes the color for one gene to a different color randomly selected from the palette. The *selection* operator is the binary tournament selection operator described by Deb *et al.* for NSGA-II [88]. A binary tournament selection holds a competition between two solutions, selecting the firstly ranked one with a probability p , and the second one with probability $p \cdot (1 - p)$.

Fitness functions and constraints. Let us consider an app having ns screens SCR_w with $w = 1, 2, \dots, ns$. For each of these screens, let us assume that it has been estimated that, on average, the h -th screen is being displayed for a percentage pt_h of the total application usage time. Such estimates can be obtained by profiling application usages, and are needed so to prioritize low energy consumption on screens that are displayed for a long duration of the overall usage time. That is, having bright colors on a splash screen

appearing for a few seconds is not as crucial as having them on the main application GUI. Now, let us also assume that each screen has a $X \times Y$ resolution, and that, given a pixel in position $\langle x, y \rangle$ of the screen h , the function $B(x, y, h)$ returns the index of the BOCP entry that the pixel belongs to. Therefore, given a GA individual solution S , the color that such a solution assigns to a pixel $\langle x, y \rangle$ in screen h is $S[B(x, y, h)]$.

The first fitness function (to be minimized) is the *Energy Consumption Fitness (ECF)*. Given a solution S , it computes its estimated consumption per unit of time based on Equation 5.1, as in the following:

$$ECF(S) = \sum_{h=1}^{ns} p_h \cdot \sum_{x=1}^X \sum_{y=1}^Y P(S[B(x, y, h)]) \quad (5.6)$$

where $ECF(S)$ sums the estimated power consumptions of all the pixels in all the screens, weighting them for the estimated fraction of time p_h for a particular screen in use.

The second fitness function (to be maximized) is the *contrast fitness (CF)*, which relies on the GUI model extracted as explained in Section 5.1.2. Let us consider that each screen h is composed of nc_h components $C_{i,h}$, the $Adj(C_{i,h})$ function returns the set of adjacent components to $C_{i,h}$, and $BC(C_{i,j})$ returns the index of the entry in the BOCC to which the component belongs.

$$TCon(C_{i,h}) = \sum_{C_{j,h} \in Adj(C_{i,h})} Con(S[BC(C_{i,h})], S[BC(C_{j,h})]) \quad (5.7)$$

$$CF(S) = \sum_h \sum_{i=1}^{nc_h} TCon(C_{i,h}) \quad (5.8)$$

That is, $CF(S)$ sums, for each screen the contrast between each component $C_{i,h}$ in the screen h and all its adjacent components $C_{j,h} \in Adj(C_{i,h})$, using the function $Con(a, b)$, that returns the contrast between two colors a and b (in terms of brightness) using a formula defined by W3C [233]:

$$Con(a, b) = \left| \frac{299 \cdot red_a + 587 \cdot green_a + 114 \cdot blue_a}{1000} - \frac{299 \cdot red_b + 587 \cdot green_b + 114 \cdot blue_b}{1000} \right| \quad (5.9)$$

Conversely to the W3C luminosity-based contrast ratio used during the color quantization (Section 5.1.2) that focuses on contrast between text and background, the brightness-based contrast measurement $Con(a, b)$ is designed to compare background colors.

The third fitness function (to be minimized) is the *design fitness (DF)* and it aims at producing GUIs with a set of colors that are close to the ones used in the original design. Basically, while it can be acceptable to swap colors with respect to the original design—*e.g.*, to use the darker color for the background and the brighter color for the text or for small components—or to have relatively similar colors, changing completely the palette might result in a GUI with a choice of colors deviating drastically from the designer’s intention. For this reason, the idea behind the third fitness function for a solution S is to use the distance between each color $S[i]$ in S and the closest color $Or[j]$ in the original palette. To compute the distance between two colors a and b , we use, according to existing literature [217], the Euclidean distance along the green, red, and blue components of the color. Given $d(a, b)$ the distance between two colors a and b , $BOCC_{Or}$ the BOCC for the original design, and $Or^*(color)$ is a function that, given a color, returns the closest color in the original palette:

$$\alpha(color) = \begin{cases} 1 & \text{if } BOCC[color] \neq BOCC_{Or}[Or^*(color)] \\ 2 & \text{otherwise} \end{cases} \quad (5.10)$$

$$DF(S) = \sum_{i=1}^n \alpha(S[i]) \cdot d(S[i], Or^*(S[i])) \quad (5.11)$$

Namely, for each color in $S[i]$, $DF(S)$ sums the distance to the closest color $Or^*(S[i])$ according to a penalization factor (α) defined by Equation 5.10. The rationale here is that we want to penalize twice color differences when they occur on the same components (*i.e.*, same entry in the BOCC), though this can be a conflicting objective with respect to

the $ECF(S)$. Therefore, if the color $S[i]$ is used in S for the same set of components with color $Or^*(S[i])$ in the original design, the distance is penalized twice.

Besides the three fitness functions, we also define a constraint aimed at avoiding solutions with low contrast. Specifically, we consider one constraint violation every time there is a pair of adjacent components $(C_{i,h}, C_{j,h})$ when:

$$Con(S[BC(C_{i,h})], S[BC(C_{j,h})]) < CnTh \quad (5.12)$$

where $CnTh$ is a contrast threshold. Constraints are handled using the mechanism defined in NSGA-II [88]. Constraint violations influence the binary tournament selection, and this is done using an additional domination principle named constraint domination. Namely, (i) feasible solutions (not violating any constraints) are ranked better than infeasible ones, (ii) feasible solutions are ranked in terms of the objectives dominance, and (iii) infeasible solutions are ranked based on the number of violated constraints. As a consequence of the latter point, solutions violating constraints are not necessarily discarded, but, for the sake of diversity, they survive. Clearly, once the resulting Pareto front has been obtained when the evolutionary algorithm stops, solutions violating constraints are discarded in case they are still present.

5.2 GEMMA's Architecture

GEMMA's architecture is outlined in Figure 5.6. GEMMA is composed of four main components: (i) the user interface implemented as a web client, (ii) the Execution Engine (EE) that executes the requests for GUI optimizations, and (iii) a NoSQL engine used to asynchronously communicate the web client and the EE, and (iv) a data collection APK that can be installed on any physical device. The GEMMA's architecture is an instance of the infrastructure described in Section 3.3.

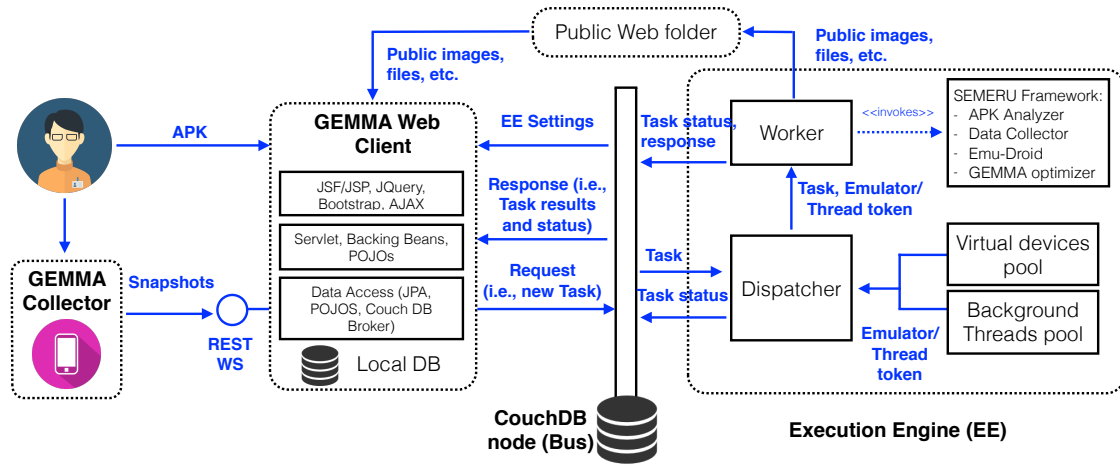


Figure 5.6: Architecture of the GEMMA Web Client and the Execution Engine.

5.2.1 The GEMMA Web Client

Android developers and GUI designers can access GEMMA’s features by means of a web client. The goal of the web client is to provide users with an easy way to: (i) request the execution of GEMMA tasks in the EE for optimizing the energy consumption of the GUIs in target apps, (ii) explore the Pareto front of solutions generated by GEMMA, and (iii) visualize the differences of the proposed solutions in terms of energy savings, contrast improvement, distance from the original design, and color palettes. A request is a set of attributes describing the optimization task, *e.g.*, APK to analyze, task name, number of GUIs to analyze. Therefore, the requests for optimizations are saved by the web client in the CouchDB engine as JSON documents. The APKs of each request are stored in the CouchDB engine by using its “attachments” feature [14]. Then, when the request is dispatched by the EE to an internal worker (more details later), the request is removed from the CouchDB engine, and a document is created for the task with all the information generated during the processing (*e.g.*, status, solutions, energy improvements). The web client checks on demand for updates of the tasks running on the EE, and locally updates the status of the non-finished tasks. When tasks are complete, all task info is synchronized

locally (*i.e.*, copied to the local DB) and removed from the CouchDB engine.

As far as technologies used and implementation details are concerned, the client is a Java web application that uses Java Server Faces, Bootstrap, and JQuery for the presentation layer. The charts (Pareto Front and gauge-style) are implemented using the HighCharts library. The web client uses a local MySQL database to store the information of the tasks once they are processed by the Execution Engine. The data access layer is realized using the EclipseLink implementation of JPA. The communication between presentation and data access layers is done with a Servlet, backing beans (JSF), and POJOs.

5.2.2 The GEMMA Collector

User can start a request through the GEMMA Web Client, by submitting an APK, which is executed automatically on the EE. The systematic exploration in the EE automatically selects the GUIs to analyze. However, to allow users to select the GUIs under analysis we developed an Android app, called GEMMA Collector, which runs on any Android-enabled device and connects to the business logic of the GEMMA Web Client via a REST web service. The GEMMA Collector relies on the Accessibility Services API and the uiautomator from the Android Framework to collect screenshots and GUI hierarchy of the GUIs selected by the user. The corresponding files, collected at the device, are sent to the GEMMA Web Client as a ZIP file. Then, the GEMMA Web Client creates a request for the EE. This request does not contain an APK, because no systematic exploration of the app is required on the EE, since the snapshots of the GUIs to analyze have been already collected by the user (*i.e.*, no virtual devices are required to execute the app in the EE). The GEMMA Collector also allows to reduce the amount/time of processing required in the EE. After the analysis, independently of the client selected by the user (*i.e.*, GEMMA Collector or GEMMA Web Client), the results of the optimization process will be available at the GEMMA Web Client. Android apps that require authentication data (*e.g.*, login and password) can not be executed automatically by the systematic exploration. This limitation is not present when using the GEMMA collector, because

the users can input private data in their apps (directly on the device) and navigate the GUIs without any restriction.

5.2.3 The GEMMA Execution Engine

The Execution Engine (EE) is the heart of GEMMA. The EE is in charge of executing the optimization tasks. In particular, the EE: (i) automatically executes the APK on a virtual device by using systematic exploration similarly to what has been done in previous work [171, 194]; (ii) selects the GUIs to optimize according to the number of GUIs requested by the user and an execution time-based heuristic (*i.e.*, the GUIs are ranked according to the number of visits during the systematic exploration)⁴; (iii) analyzes the selected snapshots (*i.e.*, GUI screenshot and GUI hierarchy tree) to identify GUI components, containers, and salient colors on the components; (iv) builds the data structures required for the Genetic Algorithm (GA) execution; (v) executes the multi-objective GA; and finally (vi) saves the results to the CouchDB client and copies screenshots illustrating the suggested color compositions into a folder that can be accessed by the web client.

The EE runs as a Java daemon that queries the requests in the “bus”, and then dispatches the requests to workers (*i.e.*, the units in charge of running GEMMA’s tasks) following a FIFO policy. During the dispatching process, the EE verifies the availability of free Emulators and background threads. If no emulators or threads are available in the EE, the dispatcher keeps the tasks on queue. Otherwise, the task is assigned to a worker that requires one background thread and one emulator. Once a worker is dispatched with a task, an emulator, and a thread, start to run (asynchronously) the GEMMA tasks listed before. During the execution, the workers update the status of the assigned tasks directly to the “bus”, and after completion, the generated artifacts (*i.e.*, solutions) are updated in the task document (in the CouchDB engine), and the solution screenshots are copied into a public web folder. Note that, because there are no synchronous messages between

⁴Note that conversely to our original approach [167] in which users selected manually the GUIs to analyze, GEMMA web automatically detects the salient GUIs.

Requests/Tasks		New GEMMA		Carlos Bernal				
Finished								
Date	App	Version	Name	Time	Solutions	MEC	MC	MD
15/11/2015	Learning Music Notes	1.2	Gemma Process 1	00:45:50	20	152.89	19,698.0	643.957
Queue								
Date	App	Version	Name	Time	Status			
12/11/2015	DiabetesPlus	1.0.4	Third process	12:54 PM	Waiting for thread			
13/11/2015	Walmart	3.1.0	Second process	08:54 AM	Running: Genetic Algorithm			
14/11/2015	Fit Brains Trainer	1.3.1	First process	10:54 AM	Running: Systematic Exploration			

Figure 5.7: List of user requests (*i.e.*, GEMMA tasks) in the GEMMA web client.

the workers and the web client notifying when a task is finished, the clients should query the “bus” and synchronize their local databases to reflect the responses/results generated with each task. The workers rely on components (*i.e.*, APK-Analyzer, Data-collector, Emu-droid, and the GEMMA’s optimizer) previously developed by the authors for static and dynamic analysis of Android apps and used in [171, 194, 167, 195]. The GEMMA’s optimizer, uses the approach described in Section 5.1; the implementation relies on the NSGA-II algorithm provided by JMetal [94]. The optimizer generates Pareto-fronts of solutions while optimizing the following objectives: (i) energy consumption, estimated through regression models, (ii) contrast between adjacent component (a minimum contrast value is also added as problem constraint), and (iii) color distance with respect to the original color composition. The algorithm produces solutions considering, as available colors, the original ones, black, white, and sets of colors achieving equidistant harmony. Noticeably, the fitness function weights the energy consumption of each app screenshot based on the estimated proportion of time it is being displayed during an usage scenario.

5.2.4 GEMMA in Action

After login, a user can (i) request for a new GEMMA task, (ii) check the status of her requests/tasks, or (iii) check the solutions generated with a finished GEMMA task. GEMMA web client has a form for submitting a new request that includes (i) the name of the “request”, (ii) the name of the app, (iii) the APK to analyze, and (iv) the number of GUIs to analyze in the APK. After a request is submitted, the user can check the status of the execution on the GEMMA EE. Figure 5.7 depicts the window in GEMMA for listing the requests, which groups finished and unfinished tasks. For example, the list of requests in Figure 5.7 shows that the user “Carlos Bernal” has three unfinished requests for the apps *Diabetes Plus*, *Walmart*, and *Fit Brains Trainer*, and one finished task for *Learn Music Notes*. For the unfinished requests, the web client shows that one is still waiting for dispatching (*i.e.*, “Waiting for thread”), another is in execution mode and running the “Systematic Exploration”, and the last one is on “Running: Genetic Algorithm”. If errors/exceptions occur during processing, the status of the request is set to “Error” with a link to the complete description of the error. The list of requests is updated on-demand, it means, when the user refreshes the window.

For finished tasks, GEMMA has a link to the name of the corresponding app, which leads to the details of the generated solutions (dashboard). The GEMMA dashboard, depicted in Figure 5.8, includes the following visual artifacts: (i) a table-based summary (sorting capabilities are included) listing the top-5 solutions for each optimization objective ; (ii) A 3D Pareto front chart (navigable) with the optimized solutions; (iii) color palettes of the original design and a specific solution selected on the Pareto front; (iv) an image of the visual appearance of the selected solution; and (v) a gauge-style visualization comparing a solution to the original design of the three optimization objectives (*i.e.*, energy, total contrast of the GUI components, and distance to the original design). The Pareto front (Figure 5.9) uses a 3D space to plot the energy consumption of the solutions in the x-axis, the contrast in the y-axis, and the distance to original design in the z-axis. The Pareto

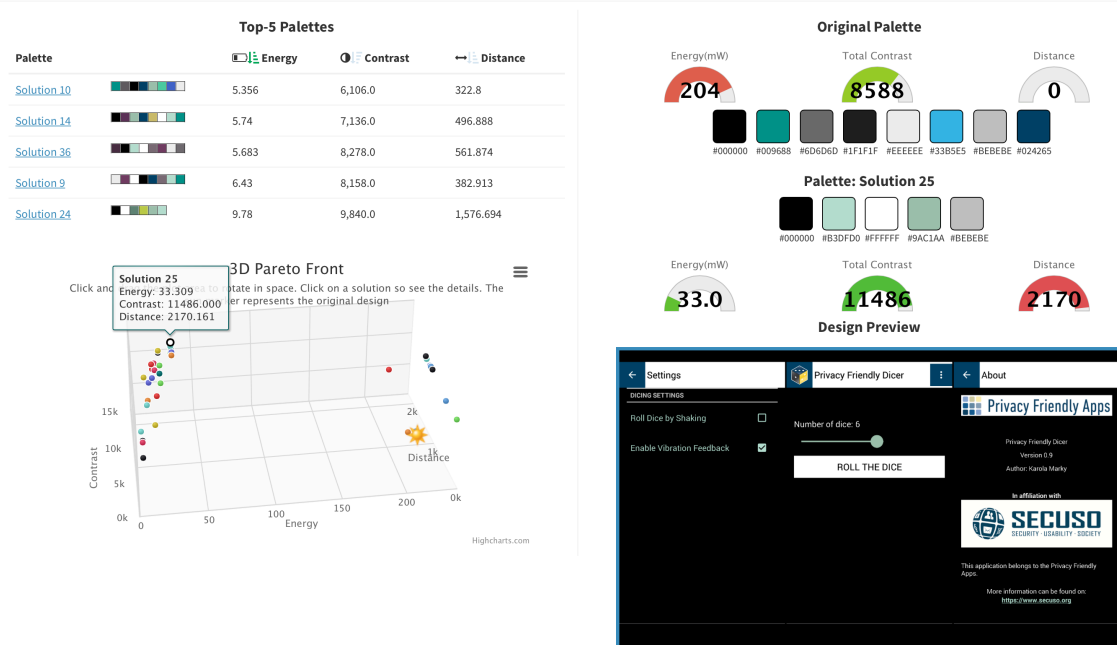


Figure 5.8: Dashboard of GEMMA’s solutions for the *Privacy Friendly Dicer* app.

front also shows the original design using a sun marker. For example, Figure 5.9 depicts the results of a real execution of GEMMA for the *Learn Music Notes* app, in which 20 solutions were generated.

When a user moves the mouse pointer over a solution in the Pareto front, GEMMA displays a contextual window (Figure 5.9) listing the values of the solution for energy consumption, contrast, and distance from the original design. In addition, when clicking on a solution, GEMMA updates the other visual artifacts in the dashboard, to plot the information corresponding to the selected solution. The Pareto front can be downloaded as an image.

Concerning the GEMMA Collector, once the APK is installed in the device, the GUIs collection workflow (on an Android-enabled device) starts with a window showing all the apps installed in the device (See Fig. 5.10-left). Then, after selecting an app, the GEMMA Collector launches the app and render to overlay buttons that allows for (i) getting an

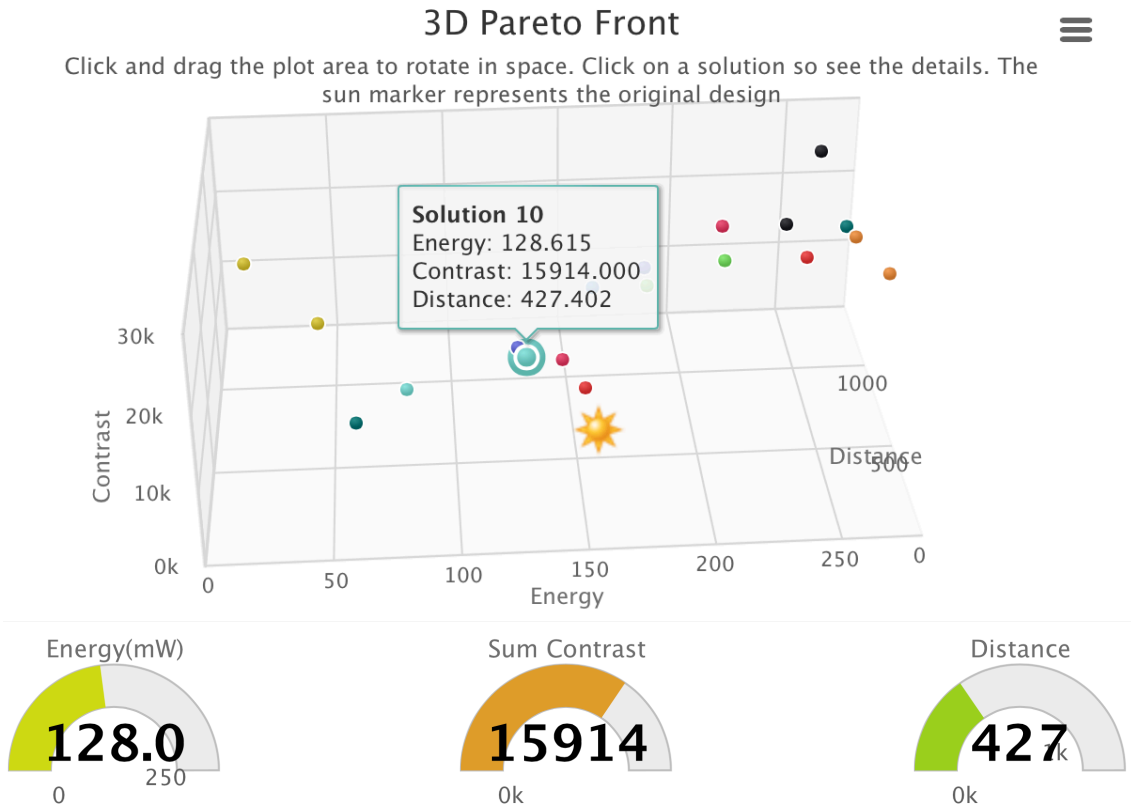




Figure 5.9: Pareto Front (top) and gauge style visualizations (bottom) in GEMMA .

snapshot of the current GUI , or (ii) submit the files to the REST WS running on the GEMMA Web Client  (See Fig. 5.10-center). Before submit the files, the GEMMA Collector lists a preview of the collected GUIs and tex inputs for authentication purposes (See Fig. 5.10-right).

5.3 Empirical Study Design

The *goal* of the study is to evaluate GEMMA in terms of (i) the energy savings that could be achieved by running Android apps when adopting the GUI color design recommended by GEMMA; (ii) the colorfulness of the GUIs that GEMMA produces as assessed by mobile apps users; and (iii) GEMMA’s suitability in an industrial context, when applied to

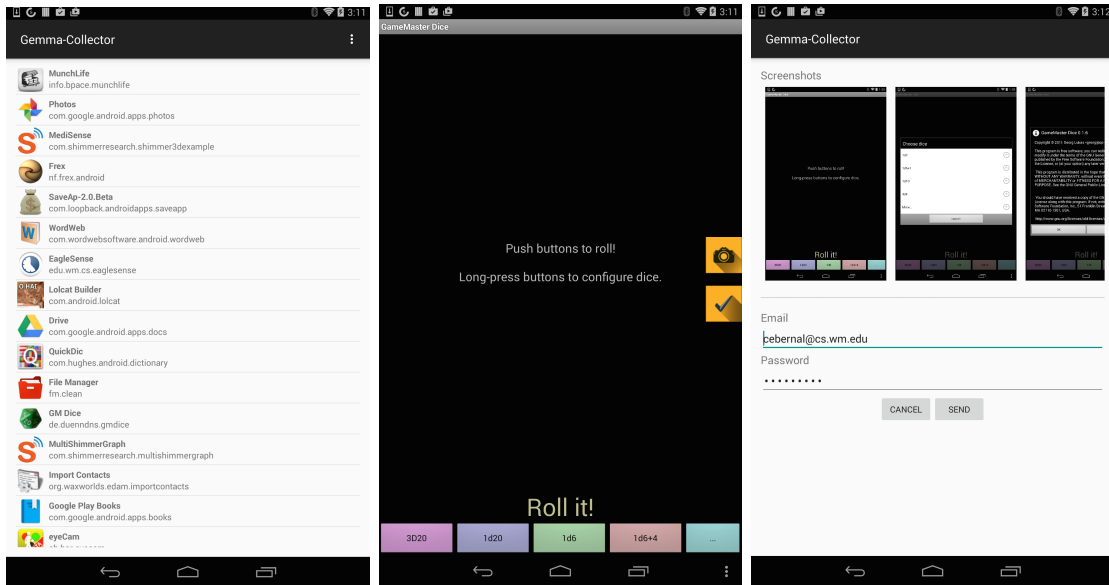


Figure 5.10: Screenshots of the GEMMA Collector Android app when running on a Nexus 7 tablet and collecting GUIs information for the Game Master Dice app. The screenshots in this Figure are: initial activity (left), collection view (center), pre-submission activity (right).

minimize GUI energy consumption of existing commercial apps. The *context* of the study consists of 27 apps from the Google Play and F-droid markets, 104 mobile app users, and three software companies (for a total of three project managers involved). The *quality focus* is the effectiveness of GEMMA in producing GUIs aimed at reducing the energy consumption of Android apps while also keeping a visually attractive color composition and ensuring sufficient contrast.

All the materials used in our study are publicly available in our replication package [164].

5.3.1 Research Questions

In the context of our study we formulated the following research questions (RQ):

- **RQ₀:** *Which multi-objective optimization approach is suitable for GEMMA?* This RQ is preliminary to the other research questions. Our goal in RQ₀ is to identify a

suitable search-based algorithm for the specific multi-objective problem we defined in Section 5.1. To this aim, we assess GEMMA’s ability in optimizing the three fitness functions formulated in Section 5.1 when using three different multi-objective GAs, namely (i) the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [88], the Pareto Archived Evolution Strategy (PAES) [148], and the Strength Pareto Evolutionary Algorithm 2 (SPEA2) [248].

- **RQ₁**: *To what extent is GEMMA able to optimize the GUI energy consumption, contrast, and design objectives?* This RQ aims at investigating the effectiveness of GEMMA in reducing the energy consumption of the apps’ GUIs while keeping a high contrast between its underlying components and staying as close as possible to the original design. Such investigation is merely performed in terms of the metrics used to compute the fitness functions defined in Section 5.1.
- **RQ₂**: *Are the color compositions generated by GEMMA visually attractive as perceived by Android users?* This RQ focuses on the colorfulness of the GUIs generated by GEMMA, as it is perceived by humans instead of being evaluated in terms of metrics. In visual aesthetics, colorfulness is the factor used to evaluate elements related to individual colors and compositions [196]. We do not focus on the whole concept of visual attractiveness; a complete appraisal of visual aesthetics should include other facets, such as simplicity, diversity, and craftsmanship that are related to the GUI layout and design concepts [196]. Therefore, we focus on the colorfulness only, because GEMMA only generates alternative color schemes and keeps the GUI layout intact.
- **RQ₃**: *Do the low energy-consumption color compositions generated by GEMMA have a tangible impact on the energy consumption of Android apps?* Because GEMMA relies on power models extracted a-priori at pixel level for estimating energy consumption, we also wanted to measure the actual improvements in terms of energy drawn from the battery and state of discharge when executing Android apps in real de-

vices after modifying the GUI appearance with the color compositions generated by GEMMA. In other words, **RQ₃** allows the user to answer the question “If I run the app optimized by GEMMA, how much do I gain in terms of battery life as compared to the original app?”

- **RQ₄**: *Would actual developers of mobile applications consider changing colors in an app as recommended by GEMMA?* For an approach like GEMMA a successful technological transfer is the main target objective. In this research question, we investigate the industrial applicability of GEMMA in the context of three software companies developing Android apps.

Note that our five research questions have a high degree of complementarity in GEMMA’s evaluation. **RQ₀** is preliminary to the others, and aims at selecting a suitable search-based algorithm to be used in GEMMA. **RQ₁-RQ₃** quantitatively analyze the efficacy of GEMMA in producing energy saving GUI color schemas while still keeping those GUIs visually attractive for the apps’ users. Finally, the quality of the GEMMA solutions is qualitatively investigated in the industrial context in our **RQ₄**.

5.3.2 Choice of Multi-Objective Optimization Techniques for GEMMA’s Evaluation

In the following, we provide a brief description of the three employed multi-objective optimization techniques, and we give elements to justify their choice.

NSGA-II [88] tries to ensure diversity in the evolving populations to avoid the situation where populations have been filled only with dominating solutions (because of the elitism effect, *i.e.*, best solutions are preserved). The key idea behind NSGA-II is to use an elitist selection strategy for replacement (with respect to fitness and spread). For each generation, parents and offsprings are gathered in a mating pool. A fast, non-dominated sorting approach is used to rank these individuals in subsequent Pareto fronts. Then, the next generation is built by preferring the individuals with the lower non-domination ranks.

Table 5.1: Android apps considered in our study.

App	Category	RQ ₀	RQ ₁	RQ ₂	RQ ₃	RQ ₄
360 security	Tools	✓	✓	✓		
BoardGameGeek	Books & Reference	✓	✓	✓		
Bollate	Travel & Local	✓	✓	✓		✓
Clean master	Tools	✓	✓	✓		
Daily Money	Finance	✓	✓	✓		
Diabetes plus	Medical	✓	✓	✓		
FitBrains	Education	✓	✓	✓		
Google Play Music	Music & Audio	✓	✓	✓		
Groupon	Shopping	✓	✓	✓		
Learn music notes	Puzzle	✓	✓	✓	✓	
Lege Appen	Medical	✓	✓	✓		
Mathtools	Education	✓	✓	✓		
Notes (Keep)	Productivity	✓	✓	✓		
Permission Friendly Apps	Tools	✓	✓	✓		
Privacy Friendly Dicer	Board	✓	✓	✓	✓	
Petrella	Travel & Local	✓	✓	✓		✓
QuickOffice	Productivity	✓	✓	✓		
Simple deadlines	Productivity	✓	✓	✓	✓	
Sing Happy Birthday Songs	Music & Audio	✓	✓	✓		✓
Tasks	Productivity	✓	✓	✓	✓	
Tasks: Astrid To-Do List	Productivity	✓	✓	✓	✓	
The Weather Channel	Weather	✓	✓	✓		
TOE - tic tac toe	Puzzle	✓	✓	✓		
Walmart	Shopping	✓	✓	✓		
WorkTime	Productivity	✓	✓	✓		
Italian app 1	–	✓	✓	✓		✓
Italian app 2	–	✓	✓	✓		✓

In case of individuals with the same rank, the selection depends on a diversity preservation mechanism. A density measure—*crowding distance*—is computed in the objective space. The tournament used to select parents for reproduction is based on this density measure.

PAES [148] is one of the simplest multi-objective optimization techniques, and has many variants working on a single individual (*i.e.*, performing a multi-objective hill-climbing) or storing a population of individuals. The basic idea behind PAES is that, when offspring is generated, it is accepted when it dominates the parent, whereas a new offspring is generated if parents dominate the generated offspring. If there is no domina-

tion relationship, a comparison is performed with previously generated solutions, which are maintained in an archive of nondominated solutions.

SPEA2 [248] is an evolution of the SPEA [249] algorithm. SPEA produces offsprings by performing (through tournament selection) the mating of the current population with solutions stored in an archive built by adding all nondominated population members and by removing those that become dominated. The archive is bounded as exceeding solutions are removed by means of a clustering technique that reduces the number of solutions by preserving the characteristics of the set of archived individuals. SPEA has limitations due to (i) possible lost of outer solutions to the nondominated sets when performing the clustering, and (ii) little information captured when many individuals of the population nondominate each other. SPEA2 overcomes these limitations by using (i) an alternative truncation method instead of clustering, and a fixed archive size, and (ii) a fine-grained fitness computation that mitigates issues related to the presence of many nondominated solutions in the population.

The choice of the algorithms to experiment is not random, but driven by evidence in the literature showing their effectiveness in dealing with multi-objective optimization problems [93?]. Also, note that we did not consider more complex search-based algorithms (*e.g.*, NSGA-III [87, 139]), since they are particularly suited when solving optimization problems characterized by the presence of several contrasting objectives [87, 139], while in our problem formulation there are “only” three objectives to optimize. The best search-based algorithm identified in the context of \mathbf{RQ}_0 will be used in the evaluations conducted to answer the other four RQs.

5.3.3 Context Selection

The list of the 27 apps considered in our study is reported in Table 5.1; links to the Google Play/F-Droid websites are reported with our replication package[164]. It is worth noting that, while all 27 apps have been used in the context of \mathbf{RQ}_0 , \mathbf{RQ}_1 , and \mathbf{RQ}_2 , only five apps were used for \mathbf{RQ}_3 since this RQ required to modify the original color compositions

used by the apps and to execute scripts on the apps multiple times (*i.e.*, 30 times) in order to assess the actual energy saving ensured by the color compositions generated by GEMMA. In addition, the investigation performed in the software companies (*i.e.*, **RQ₄**) is clearly limited to the set of five apps they developed and provided to us for the study.

In our study we targeted a diverse set of apps in terms of domain categories and types. For instance, our dataset includes open source, commercial, and free apps, but also apps developed by Google (*e.g.*, Google Play Music and Notes). Two of the authors inspected Google Play and F-Droid to randomly select a set of native apps (*i.e.*, non-HTML based) to use in the context of our study. The selected set of apps cover ten categories (*e.g.*, productivity, weather, *etc.*) that are listed in Table 5.1.

In the context of **RQ₂**, we involved 104 app users asking for their opinion about the look and feel of the GUIs generated by GEMMA. Since **RQ₂** only aims at assessing how visually attractive the generated color schemes are, the only requirement we had for participants is to have some basic experience with mobile apps.

For the investigation of real impact on energy consumption of Android apps (**RQ₃**), we measured the energy consumption of five open source Android apps when executed on a Samsung Galaxy S4 (the same device we used to extract the power model for GEMMA). In particular, we measured the current drawn by the battery when executing the the apps with (i) the original color palette, and (ii) the lowest-energy solution suggested by GEMMA. For the experiment, we used the following apps: Privacy Friendly Dicer, Learn Music Notes, Simple Deadlines, Tasks, and WorkTime. The selection criteria for the apps were the following: (i) considering apps having a diverse set of background colors to measure the real impact on apps originally with greedy/energy saving GUIs, (ii) different domain categories, and (iii) open source apps in which we were able to modify the GUI theme.

As for **RQ₄**, we involved three Italian companies, namely GenialApps [22], Next [35] and IdeaSoftware [28], that provided us a total of five apps to use in the context of our study. All the companies have a multi-year experience in mobile app development and

each of them enlist between ten and 20 developers. As it will be shown later, we performed semi-structured interviews with people from these companies with the aim of gathering qualitative feedback about the GUIs generated by GEMMA. In particular, two project managers (from GenialApps and Next) and one developer (from MediaStudio) participated in our study.

5.3.4 Data Collection

To answer our RQs we applied GEMMA on the 27 apps in order to generate color schemas for GUIs. As explained in Section 5.1, GEMMA is able to work on multiple GUIs of the same app. Note that, in order to answer **RQ₀**, we executed GEMMA using NSGA-II, PAES, and SPEA2 as the underlying methods for multi-objective optimization. Since each of the 27 apps includes a specific number of different GUIs (or GUI states), for the sake of consistency in our data analysis we decided to consider three GUIs per app as the objective of GEMMA’s optimization process. In particular, we always consider the main GUI of the app (*i.e.*, the one visualized when the app is launched on the device) plus two randomly selected GUI states.

We used the following settings for GEMMA:

1) *Extracting Color Composition from GUIs (Section 5.1.2)*: $k = 3$, to extract three different colors from each GUI component: background, text (or second main color), and the third color for the border. As for the contrast ratio r , we found 1.6 to be a common lower bound for the contrast between colors in the borders and the background as well as the background and the text in the Android apps considered in our study.

2) *GA parameters (Section 5.1.3)*: $p_{cross} = 0.9$; $p_{mut} = 1/|BOCP|$, where $|BOCP|$ is the number of bags of pixels identified in a GUI. Such parameters were the default ones in *jMetal* [94]; we at least checked whether a lower crossover rate or higher mutation did not produce better results than ours. The other choices were *population size=50* and *GA number of evaluations=50,000* (larger populations and higher number of evaluations did not produce better results). Finally, we set the minimum contrast ratio between adjacent

components (*CnTh*) to 4, as we found this value to be sufficient to ensure readability of a text over a background.

For each app we run GEMMA 30 times to account for the randomness of the multi-objective GAs [63]. At each run we stored for each solution in the Pareto front respecting the contrast ratio constraint (see Section 5.1) as well as for the original color schema of the app: (i) its energy consumption (ECF function), (ii) its contrast ratio (CF function), and (iii) its “distance” from the original design (DF function)—see Section 5.1.

To answer **RQ₂** we designed an online survey, aiming at collecting the participants’ opinions about the aspect of the GUIs generated by GEMMA. We wanted to verify if the energy saving GUIs generated by GEMMA are also attractive in terms of colorfulness. The first part of our survey included three questions aimed at gathering information about the background of the study’s participants:

1. *What is your current position?* bachelor student, master student, PhD student, developer, tester, graphic designer, other;
2. *How frequently do you use mobile apps?* never, occasionally (once a day), frequently (several times per day);
3. *Is the color composition of the GUI important for you in the overall judgment/rating of a mobile app?* 1=not important at all, 2=slightly important, 3=important, 4=very important.

The second part of the questionnaire asked participants for their feedback about the colorfulness of the original apps’ GUIs as well as of those recommended by GEMMA. Since GEMMA provides a set of solutions (*i.e.*, all those in the Pareto front respecting the contrast ratio constraint) as output and that we run GEMMA 30 times on each app, we needed to make a selection of the recommended GUIs to be shown to the survey’s participants. For each of the 27 apps involved, we showed to participants: (i) the original three selected GUIs; (ii) the GEMMA’s solution having the lowest energy consumption

across those generated in the 30 executions; and (iii) the GEMMA’s solution having the median energy consumption across those generated in the 30 executions. Thus, for each app A , participants examined, in a single page of the survey, three sets of GUIs, each one containing a different color schema for the three GUIs selected for A . For each of them, participants had to answer the following questions from the colorfulness evaluation as suggested by Moshagen and Thielsch [196]:

1. *Is the color composition visually attractive?*
2. *Do the colors match?*
3. *Is the choice of colors botched?*
4. *Are the colors appealing?*

The possible answers for each question were presented with a four-point Likert scale: 1=absolutely not, 2=more no than yes, 3=more yes than no, 4=absolutely yes [196]. Each participant evaluated the colorfulness of five apps in order to reduce the participants’ workload. Specifically, we created six groups of apps and we assigned round-robin each participant to a specific group aiming at balancing the number of evaluations for each app. Given the 104 participants in our study, each app has been evaluated by at least 12 participants. All GUIs for the same app were shown in the same page to allow an easy comparison between the different GUIs by participants. The order in which the apps were evaluated by the participants was randomized as well. None of the participants was aware of the experimental goals nor which of the GUIs in each page was the original one. It is worth noting that the Moshagen and Thielsch framework [196] includes several facets of visual aesthetics (*i.e.*, simplicity, diversity, colorfulness, and craftsmanship), however, our evaluation focused on the “colorfulness” facet because GEMMA generates solutions for the color compositions without modifying number, layout, locations, or dimensions of GUI the components.

Algorithm 5: Procedure for collecting energy measurements from real executions of Android apps.

Input: $app, Script$
Output: C, P

```
1 begin
2    $C = \emptyset, P = \emptyset;$ 
3   for  $exec \leftarrow 1$  to 30 do
4      $install(app);$ 
5     foreach  $step \in Script$  do
6        $exec(step);$ 
7        $waitForIdle();$ 
8        $startEnergySampling();$ 
9        $waitSeconds(20);$ 
10       $stopEnergySampling();$ 
11       $C_{\langle exec, step \rangle} = getAvgCurrentFromSample();$ 
12       $P_{\langle exec, step \rangle} = getAvgPowerFromSample();$ 
13    $uninstall(app);$ 
```

To collect the energy measurements required for **RQ₃** we followed a procedure similar to the one described by Linares-Vásquez *et al.* [165]. One of the authors collected execution scripts on the original versions of five apps using the `getevent` command in the Android OS and a Samsung Galaxy S4. The brightness of the display in the device was set to the maximum value, to avoid impact of automatic settings that adjust the brightness after certain time. Each script was intended to exercise as many features as possible in the app under analysis. The scripts were used to replay the collected scenarios automatically in the apps, therefore, the measurements have not been biased or impacted by human-based reproduction issues. We recorded the scripts mostly focused on exercising the components in the 3 screens analyzed in GEMMA and real usage scenarios. On average the scripts have 31 steps: this is 32 steps for Simple Deadlines, Privacy Friendly Dicer, Tasks: Astrid To-Do List; 33 steps for Tasks; and 29 steps for Learn Music Notes. For the energy collection, the Samsung Galaxy S4 was wired to a Monsoon power monitor to measure the current drawn by the phone during the scenario reproduction.

The procedure for collecting energy measurements on a given app is described in

Algorithm 5. Each script was executed on each app 30 times. The energy measurements were collected (i) after each step was performed and, and (ii) after the device was in the idle mode (Algorithm 5, lines 6 and 7). It is worth noting that our purpose is to measure the energy consumed by the screen when displaying the color compositions in the GUI; therefore, we do not collect energy measurements during the step execution or meanwhile the device is executing the action related to the step. The energy measurements were collected during a sampling period of 20 seconds (Algorithm 5, lines 8 - 10), which accounts for an average of 99,000 values given the sampling frequency of the power monitor (*i.e.*, 5KHz). The power monitor provided us with average Current (mili Amperes) and average Power (mili Watts) for each sampling period Algorithm 5, lines 11 and 12)⁵. For a given app, we collected average current $C_{\langle exec, step \rangle}$ and power measurements $P_{\langle exec, step \rangle}$ for each execution step and for each of the 30 executions. Note also, that each execution was a cold-start [171], *i.e.*, we installed/uninstalled the app to avoid issues with application data.

To address **RQ₄** we conducted semi-structured interviews with the project managers of the three involved companies. The interviews lasted for two hours with each company. Given that we had sufficient time available and only a few apps for each interview, for each app we showed the whole set of solutions provided by GEMMA in a single run, *i.e.*, the run among the 30 we executed that provided the largest set of solutions. This choice was made in order to have a larger set of alternative designs to discuss with the companies. We accompanied each of the shown solutions with information about its energy consumption and potential saving with respect to the original design.

During the interview, for each solution we asked the participant to answer the following question: “*Given the energy saving provided by this design with respect to the original design, would you adopt it in your app?*”, using a score on a four-point Likert scale: 1=absolutely no, 2=no, 3=yes, 4=absolutely yes. In addition, we also asked questions to

⁵We did not collect voltage measurements because the voltage output from the power monitor is constant. We used a voltage output of 3.8 volts, which is the one required by the battery in the Samsung Galaxy S4.

understand the principal reasons why the participants liked (or did not like) the suggested colors and which are the components for which the participants noticed a poor choice of colors. The interviews were conducted by one of the authors, who annotated the provided answers as well as additional insights about the GEMMA’s points of strength and weakness that emerged during the interviews.

5.3.5 Data Analysis

To answer **RQ₀** we show boxplots of the hypervolume [64] for the pareto-fronts generated by the three algorithms. Also, we report boxplots of the energy consumption, the contrast ratio, and the distance from the original design (measured as reported in Section 5.1) obtained by running GEMMA using the three experimented search-based algorithms (*i.e.*, NSGA-II, PAES, and SPEA2). GEMMA has been run 30 times on each app with each of the three algorithms. We compare the solutions generated by the three algorithms for six categories of solutions: having the lowest and the median energy consumption, having the highest and the median contrast ratio, and having the lowest and the median distance from the original design (*i.e.*, six solutions for each run).

We statistically compare the energy consumption, the contrast ratio, and the distance from the original design of the solutions generated by the three algorithms by using the Wilcoxon test [84]. The results are considered statistically significant at $\alpha = 0.05$. Since we perform multiple tests (*e.g.*, the solutions generated by NSGA-II are compared with both those generated by PAES and by SPEA2), we adjust our p -values using the Holm’s correction procedure [131]. This procedure sorts the p -values resulting from n tests in ascending order, multiplying the smallest by n , the next by $n - 1$, and so on. We also estimate the magnitude of the observed differences by using the Cliff’s Delta (or d), a non-parametric effect size measure for ordinal data [117]. Cliff’s d is considered negligible for $d < 0.148$ (positive as well as negative values), small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$ [117]. The same statistical comparison is also performed for the hypervolume of the pareto-fronts generated by the three algorithms.

To answer **RQ₁** we show boxplots of the energy consumption, the contrast ratio, and the distance from the original design (measured as reported in Section 5.1) obtained by running GEMMA 30 times on each app. Note that in **RQ₁** we only consider the solutions generated by the most suitable search-based algorithm selected as a result of **RQ₀**. We perform the comparison by considering the same categories of solutions previously mentioned (*i.e.*, having the lowest and the median energy consumption, having the highest and the median contrast ratio, and having the lowest and the median distance from the original design). We statistically compare the energy consumption and the contrast ratio of the generated solutions with those of the original design of the considered apps by using the same procedure described for **RQ₀** (*i.e.*, Wilcoxon test, Holm’s correction procedure, and Cliff’s Delta).

For what concerns **RQ₂**, we must firstly note that our goal is not to determine whether GUIs recommended by GEMMA are considered more attractive than the original GUIs, but, rather, whether they are acceptable enough, and to know what is the “price to pay” in terms of visual aesthetics for the energy saving solutions provided by GEMMA. To this aim, we report boxplots of the ratings assigned by participants to the colorfulness of the different categories of GUIs. Also, we statistically compare the ratings assigned to the original GUIs of the 27 apps to those assigned to generated solutions with lowest and median energy consumption by performing the same statistical analysis used in **RQ₁**.

In the context of **RQ₃**, we use descriptive statistics to compare the real energy measurements of original (a_0) and low-energy consumption color compositions (a_{gemma}) of an app a . Because we collected energy measurements (*i.e.*, current intensity and power) over 30 executions of a given app version, we analyze the results by aggregating the $C_{\langle step \rangle}$ and $P_{\langle step \rangle}$ series, using the average, min, and max. In that sense for the current we derived three series $\bar{C}_{\langle step \rangle}$, $C \uparrow_{\langle step \rangle}$, $C \downarrow_{\langle step \rangle}$, representing average, max, and min values; for the power we also derived the series $\bar{P}_{\langle step \rangle}$, $P \uparrow_{\langle step \rangle}$, and $P \downarrow_{\langle step \rangle}$. We use descriptive statistics to compare the energy measurements of original (a_0) and low-energy consumption color compositions (a_{gemma}) of an app a . In addition, we statistically compare the

Table 5.2: Hypervolume: Wilcoxon test (adjusted p -values) and Cliff’s delta (d).

Test	adj. p -value	d
NSGA-II vs PAES	<0.01	0.15 (Small)
NSGA-II vs SPEA2	<0.01	-0.12 (Negligible)
PAES vs SPEA2	<0.01	-0.25 (Small)

energy measurements using the same procedure in **RQ**₁ (Wilcoxon test with $\alpha = 0.05$ and Cliff’s d), but without the Holm’s correction because in **RQ**₃ we do not perform multiple comparisons. In order to have a practical implication of the results achieved, we compute the percentage of the battery charge consumed by an app version consuming a certain amount of Joules during the experiments. To this, given a $\bar{P}_{\langle step \rangle}$ series, first we compute the total energy in Joules. To compute the energy in Joules, for each value $\bar{P}_{\langle step \rangle}$ we derived $\bar{E}_{\langle step \rangle}$ using Eq. 5.13:

$$\bar{E}_{\langle step \rangle} = \rho \left(\bar{P}_{\langle step \rangle} mW \cdot \frac{1KW}{10^6 mW} \cdot \Delta t secs \cdot \frac{1hour}{3600secs} \right) \quad (5.13)$$

with $\rho = 3.6 \cdot 10^6$ the conversion constant from $KW \cdot hour$ to $Joule$, and Δt the duration of the sampling period for $step$ (i.e., 20 secs). Finally, given a series $\bar{E}_{\langle step \rangle}$, we compute the percentage of the total battery charge consumed on average by each step $\bar{B}_{\langle step \rangle}$ [165]:

$$\bar{B}_{\langle step \rangle} = \bar{E}_{\langle step \rangle} J \cdot \frac{1h}{3600secs} \cdot \frac{1}{3.8V} \cdot \frac{1000A}{2600mAh} \cdot 100 \quad (5.14)$$

In our context, the Galaxy S4 is equipped with a 2,600 mAh, 3.8V battery. Thus, an energy consumption of 0.01 J will consume $2.81 \cdot 10^{-5}\%$ of the total battery charge.

Finally, for **RQ**₄ we qualitatively discuss the outcomes of the semi-structured interviews.

5.4 Study Results

This section reports the analysis of the results for the three research questions formulated in Section 5.3.1.

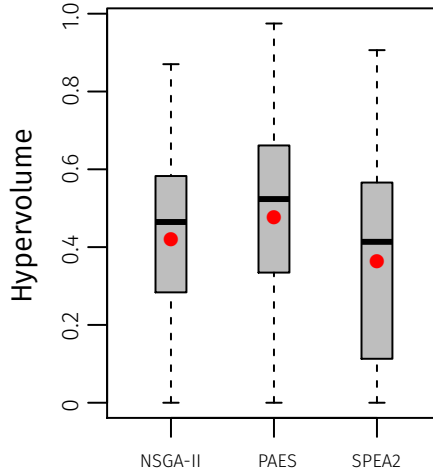


Figure 5.11: Hypervolume for the three algorithms.

5.4.1 RQ₀: Which multi-objective optimization approach is suitable for GEMMA?

Figure 5.11 shows the boxplots of the hypervolume obtained by the three experimented algorithms (*i.e.*, NSGA-II, PAES, and SPEA2) when running each of them 30 times on the 27 subject apps. PAES is the algorithm obtaining the highest hypervolume (mean=, median=), followed by NSGA-II (mean=, median=) and SPEA2 (mean=, median=). Table 5.2 reports the results of the Wilcoxon test (adjusted p -values) and the Cliff's d effect size when comparing the distributions of hypervolume generated by the three algorithms. The algorithm achieving the highest hypervolume is highlighted in **bold** for each test. From the results in Table 5.2 it is clear that (i) PAES is able to achieve a significantly higher hypervolume with respect to the other two algorithms (*i.e.*, p -value < 0.05), and (ii) the effect size is small or negligible, thus highlighting marginal differences in the hypervolume obtained by the three algorithms.

To have a more fine-grained comparison of the ability of the three algorithms in optimizing the three objectives defined in Section 5.1.3, Figure 5.12 reports the distributions of ECF—Figure 5.12-(a)—, CF—Figure 5.12-(b)—and DF—Figure 5.12-(c)—for the solu-

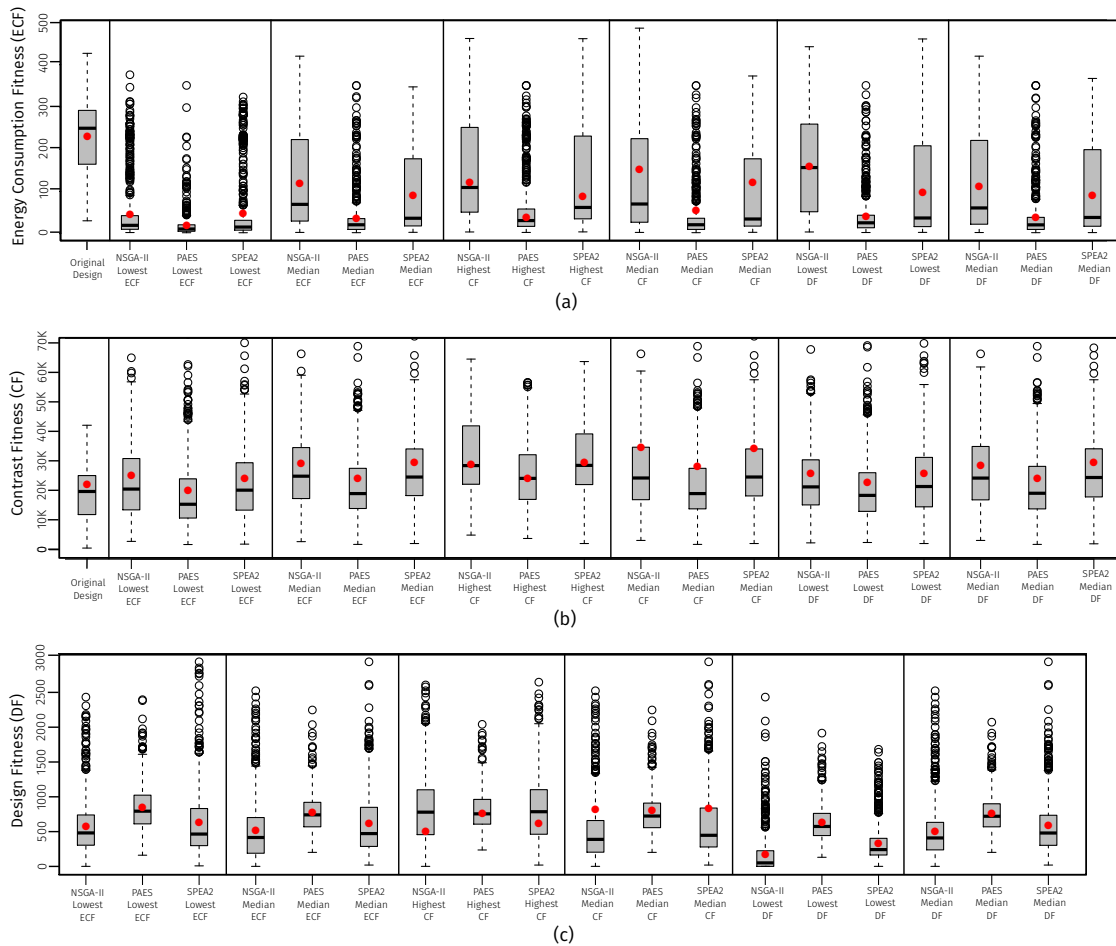


Figure 5.12: Boxplots of ECF, CF, and DF for different solutions generated by the three algorithms.

tions in the Pareto front obtained at each GEMMA’s run having the: *lowest ECF*, *median ECF*, *highest CF*, *median CF*, *lowest DF*, and *median DF*. Results are shown for all three algorithms. Note that Figure 5.12 does also show the ECF and CF of the original apps’ design, that will be discussed in the context of RQ₁. In addition, Tables 5.3, 5.4, and 5.5 reports the results of the Wicoxon test and the Cliff’s *d* effect size when comparing the ECF, CF, and DF, respectively, of the solutions generated by NSGA-II, PAES, and SPEA2. Again, the algorithms generating the best value for a fitness function (*i.e.*, the lowest value for ECF, the highest for CF, and the lowest for DF) is highlighted in **bold** for

Table 5.3: ECF: Wilcoxon test (adjusted p -values) and Cliff’s delta (d).

Test	adj. p -value	d
lowest ECF		
NSGA-II vs PAES	<0.01	-0.36 (Medium)
NSGA-II vs SPEA2	<0.01	-0.12 (Negligible)
PAES vs SPEA2	<0.01	0.23 (Small)
median ECF		
NSGA-II vs PAES	<0.01	-0.56 (Large)
NSGA-II vs SPEA2	<0.01	-0.21 (Small)
PAES vs SPEA2	<0.01	0.35 (Medium)

Table 5.4: CF: Wilcoxon test (adjusted p -values) and Cliff’s delta (d).

Test	adj. p -value	d
highest CF		
NSGA-II vs PAES	<0.01	-0.25 (Small)
NSGA-II vs SPEA2	0.62	-0.016 (Negligible)
PAES vs SPEA2	<0.01	0.25 (Small)
median CF		
NSGA-II vs PAES	<0.01	-0.22 (Small)
NSGA-II vs SPEA2	0.62	0.016 (Negligible)
PAES vs SPEA2	<0.01	0.24 (Small)

each test. From the analysis of the boxplots and of the results achieved in the statistical tests, we can observe that:

1. *PAES is the best algorithm in optimizing the ECF function (i.e., is the one generating solutions having the lowest the energy consumption).* The difference with respect to the solutions generated by the other two algorithms is statistically significant with a medium or large effect size, depending on the specific solutions object of the comparison (see Table 5.3). NSGA-II ranks second in terms of ECF values, while the worse algorithm from this perspective is SPEA2.
2. *PAES is the worst algorithm in optimizing the CF function (i.e., is the one generat-*

Table 5.5: DF: Wilcoxon test (adjusted p -values) and Cliff’s delta (d).

Test	adj. p -value	d
lowest DF		
NSGA-II vs PAES	<0.01	0.82 (Large)
NSGA-II vs SPEA2	<0.01	0.53 (Large)
PAES vs SPEA2	<0.01	-0.66 (Large)
median DF		
NSGA-II vs PAES	<0.01	0.54 (Large)
NSGA-II vs SPEA2	<0.01	0.15 (Small)
PAES vs SPEA2	<0.01	-0.40 (Medium)

ing solutions having the lowest contrast). NSGA-II and SPEA2 are comparable for what concerns the contrast of the generated solutions, with no significant difference between them (see Table 5.4). Instead, both NSGA-II and SPEA2 generate solutions having a contrast significantly higher than the contrast of the solutions generated by PAES. The effect size is small for all solutions but for *lowest DF*, for which is negligible.

3. *NSGA-II is the best algorithm in optimizing the DF function (i.e., is the one generating solutions closer to the original design)*. PAES is again the worse algorithm in this case. The difference is statistically significant for all solutions but for *highest CF*, and the effect size is generally large (see Table 5.5).

Thus, while PAES is able to achieve the best results in terms of energy consumption (ECF), it also achieves the worst results when it comes to optimize the contrast (CF) and the distance from the original design (DF) of the generated solutions. NSGA-II, while being the second choice for what concerns ECF, is the best performing algorithm in the optimization of CF and DF. Also, while the comparison of the solutions generated by GEMMA and the original design is detailed in RQ₁, Figure 5.12-(a) clearly shows the substantial reduction of energy consumption ensured by the NSGA-II solutions with respect to the original design. Since our goal is to reduce energy consumption *while* generating pleasant design solutions close to the original designs, we opted for using NSGA-II as the search-based optimization algorithm in GEMMA. This is the algorithm that we used to generate the results for our next research questions.

5.4.2 RQ₁: To what extent is GEMMA able to optimize the GUI energy consumption, contrast, and design objectives?

We base our results discussion for this research question on the boxplots in Figure 5.12, by only comparing the distribution of values for the three fitness functions described in

Table 5.6: ECF and CF: Wilcoxon test (adjusted p -values) and Cliff’s delta (d).

Test	adj. p -value	d
ECF		
original design vs lowest ECF	<0.01	-0.87 (Large)
original design vs median ECF	<0.01	-0.53 (Large)
original design vs highest CF	<0.01	-0.37 (Medium)
original design vs median CF	<0.01	-0.52 (Large)
original design vs lowest DF	<0.01	-0.35 (Medium)
original design vs median DF	<0.01	-0.56 (Large)
CF		
original design vs lowest ECF	<0.01	0.09 (Negligible)
original design vs median ECF	<0.01	0.24 (Small)
original design vs highest CF	<0.01	0.44 (Medium)
original design vs median CF	<0.01	0.24 (Small)
original design vs lowest DF	<0.01	0.11 (Negligible)
original design vs median DF	<0.01	0.22 (Small)

Section 5.1.3 exhibited by the *original design*⁶ and by solutions in the Pareto front obtained at each GEMMA’s run when using NSGA-II. Again, we focus on solutions having the: *lowest ECF*, *median ECF*, *highest CF*, *median CF*, *lowest DF*, and *median DF* at each run. In addition, Table 5.6 reports the results of the Wilcoxon test (adjusted p -values) and the Cliff’s d effect size. We compared the values for the ECF (energy consumption) and CF (contrast ratio) fitness functions between the original designs and the different categories of solutions considered in this research questions. The design achieving the best value for a fitness function (*i.e.*, the lowest value for ECF or the highest value for CF) is highlighted in **bold** for each test.

Figure 5.12-(a) highlights that the design solutions generated by GEMMA have a lower energy consumption as compared to the original design. In particular, picking on each Pareto front the solution having the lowest value for ECF ensures a mean energy saving of 79% (median 92%). In other words, the most energy efficient colour schemas recommended by GEMMA consume, on average, one fifth of the original color design. Interestingly enough, the other categories of solutions proposed by GEMMA and considered in this research question offer a substantial energy savings as compared to the original design. For instance, the solution ensuring the highest contrast level between the GUI components (*Highest CF* in Figure 5.12) offers a mean energy saving of 32% (median 56%) with respect

⁶For the original design there is no value for the DF, since this fitness function reflects the distance of the color schema of a solution from the original design.

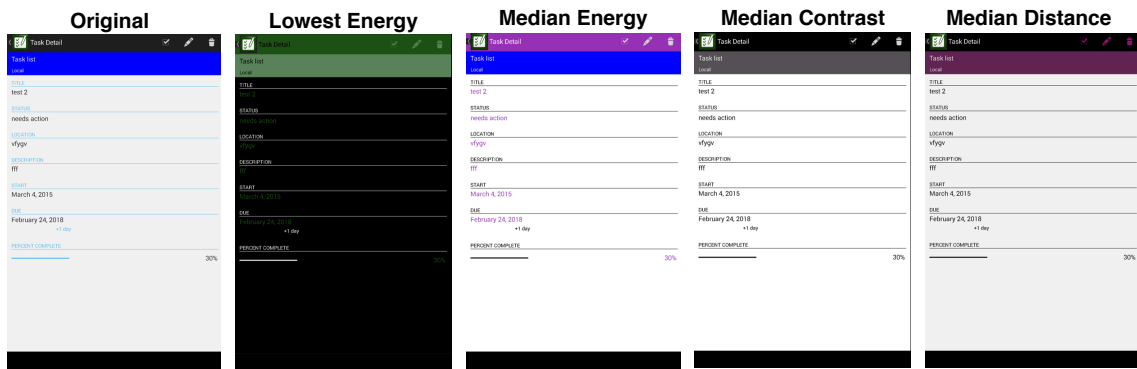


Figure 5.13: Original design *vs* GEMMA’s solutions for the Tasks app.

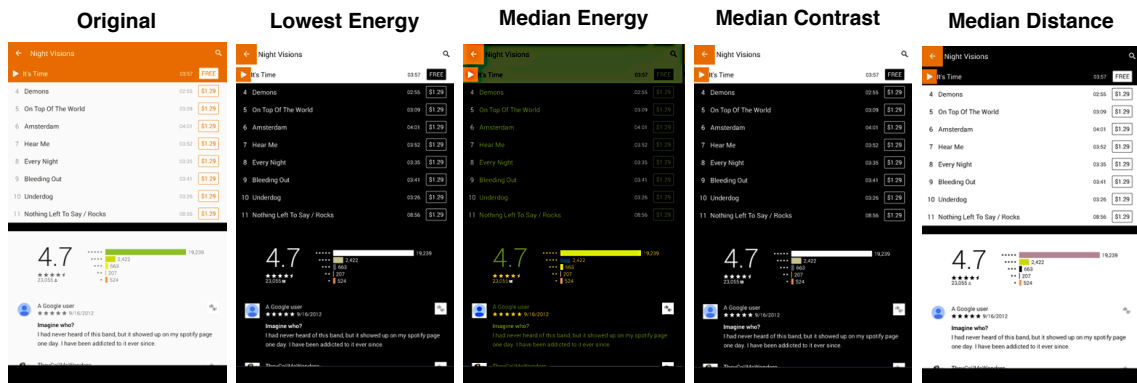


Figure 5.14: Original design *vs* GEMMA’s solutions for the Play Music app.

to the original design. The statistical comparison in terms of energy consumption (ECF) between the original design and the solutions proposed by GEMMA (top part of Table 5.6) always highlight a statistically significant difference in favor of the GEMMA’s solution accompanied by a large effect size for all categories of solutions but the one ensuring the highest contrast level and the lowest distance from the original design (medium effect size).

The results in Figure 5.12-(b) highlight that, besides ensuring a lower energy consumption, the color schemas recommended by GEMMA also help in improving the contrast ratio between the GUI’s components (*i.e.*, higher values for the CF function). The GEMMA’s solutions providing the best contrast ratio are able to obtain a mean improvement of the contrast of 51% (median 37%). It is surprising to see how GEMMA’s solutions

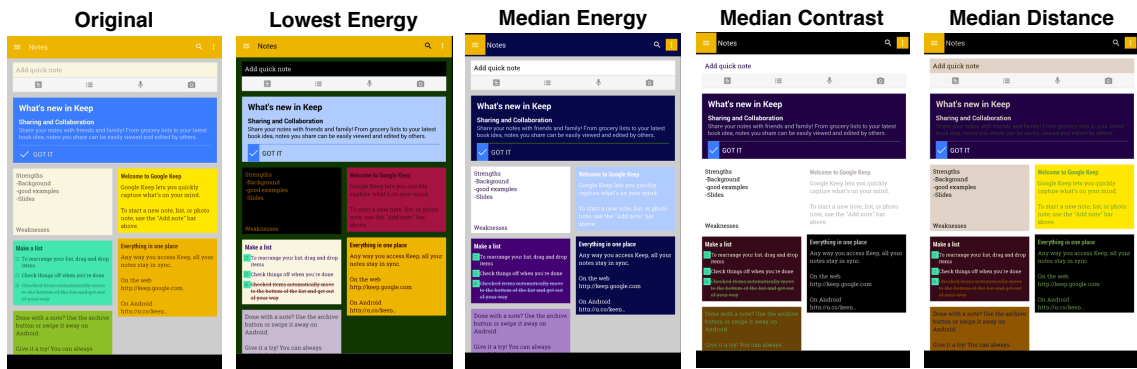


Figure 5.15: Original design *vs* GEMMA’s solutions for the Keep app.

having the lowest energy consumption are able to obtain a contrast in line with that of the original design (mean +11%, no difference in terms of median). While this result does not ensure the pleasantness of the color schemas generated by GEMMA (this aspect is investigated in **RQ₂** and **RQ₄**), at least it should ensure the readability of the generated GUIs (*i.e.*, high contrast between the GUIs components). The results of the statistical test (bottom part of Table 5.6) show that the contrast ratio ensured by the GEMMA’ color schemas is significantly higher with respect to the original ones. The effect size ranges between negligible (for the solutions ensuring the lowest ECF and DF values) and medium (for the solutions having the highest CF). While the negligible effect size achieved for solutions in the *lowest ECF* group might seem like a negative result, it is worth noting that such color schemas are able to achieve a mean reduction of energy consumption of 79%, while also being able to improve its contrast ratio (even with a negligible effect size).

Examples of the solutions generated by GEMMA for three of the subject apps in our study are shown in Figures 5.14 to 5.13 ⁷. For instance, the proposed lowest energy consumption solution for Tasks (Figure 5.13) offers energy consumption savings of up to 53% as well as an increase in terms of contrast ratio by 31%.

Finally, by looking at Figure 5.12-(c), it is interesting to understand what happens to the DF fitness function. Remember that this function equals zero when the evaluated

⁷More examples for the analyzed apps are in our online appendix

solution (*i.e.*, color schema) exactly matches the original design. In all categories of solutions we selected and reported for NSGA-II, there are solutions having zero as value for DF (as visible by the lower whisker of the boxplots “touching” the zero value). This indicates that DF function in GEMMA lets the GA search for solutions that very close to the original design.

5.4.3 RQ₂: Are the color compositions generated by GEMMA visually attractive as perceived by Android users?

The sample of 104 survey participants is distributed (in terms of current position) as in the following: 51 Bachelors, 17 PhDs, 15 industrial developers, 12 Masters, 3 product/project manages, 1 business analysis, 1 executive director, 1 professional graphical designer, 1 pediatrician, 1 high school student, and 1 participant that reported herself as a “final user”. 66 participants are from European countries, and 38 from the Americas (North, Center, and South). In addition, 86 participants (82.69%) reported they use mobile apps frequently (*i.e.*, several times per day), and 18 participants reported to occasionally use mobile apps (*i.e.*, once a day). Concerning the importance of color compositions of the GUI in the overall judgment/rating of a mobile app, 29 participants (27.89%) reported it is “very important”, 57 (54.81%) answered it is “important”, and only 19 participants (18.27%) think it is “slightly important”. Previous studies have analyzed the impact of complex GUIs in the apps ratings [223, 226], however no study has explored yet the impact of colors compositions on ratings. Note that our study is not about impact of color compositions on ratings, however, we asked our participants about the importance of color compositions to assure they care about colors compositions and provided useful answers. None of the participants answered “not important at all”, therefore we kept all the responses as useful for our study.

Boxplots in Figure 5.16 show the distribution of answers for the visual aesthetics-related questions from the 104 participants. The boxplots summarize the answers regarding the *original design*, *lowest ECF*, and *median ECF* solutions of the 27 considered apps.

In addition, Table 5.7 reports the results of the Wilcoxon test (adjusted p -values) and the Cliff’s d effect size. Figure 5.16 highlights that the solutions generated by GEMMA have a “cost” in terms of visual aesthetics as perceived by end users. Answers from the participants to the colorfulness factor—described by the four questions presented in Section 5.3.4—show a low evaluation for the solutions with the lowest ECF (*i.e.*, the one saving the maximum amount of energy). Although end users do not totally dislike the *lowest ECF* solutions, the difference with the original design is statistically significant with a large/medium effect size. By looking at the comments left by participants and justifying their low ratings for the solution having the lowest ECF, it appears clear that most of them simply do not like GUIs having dark colors as main base: “*Black backgrounds can look very good but not when there is a lot of text*”; “*I prefer other colors than black for the mobile applications*”.

Instead, participants found the *median ECF* solutions more visually appealing than the *lowest ECF* ones. The differences reported between the *original designs* and the *median ECF* solutions are small for the investigated colorfulness factors. Indeed, Figure 5.16 highlights how the median and average answers for the four questions are “close” when comparing the *original designs* with the *median ECF* solutions. This is also confirmed by the statistical analysis reported in Table 5.7: while there is a significant difference in the participants’ ratings in favour of the apps’ *original design* when comparing it with the *median ECF* solutions, such a difference only results in a small effect size. Remember that, as shown in the context of RQ₁, solutions having a *median ECF* are still able to achieve a substantial energy savings with respect to the original design (42%, on average). This illustrates GEMMA’s ability to balance multiple objectives, *i.e.*, generating energy-saving GUIs that use colors close to the original composition and being accepted by end users.

RQ₃: Do the low energy-consumption color compositions generated by GEMMA have a tangible impact on the energy consumption of Android apps? Figure 5.17 and Figure 5.18 depict the average current intensities (*i.e.*, $\bar{C}_{(step)}$), and battery consumption

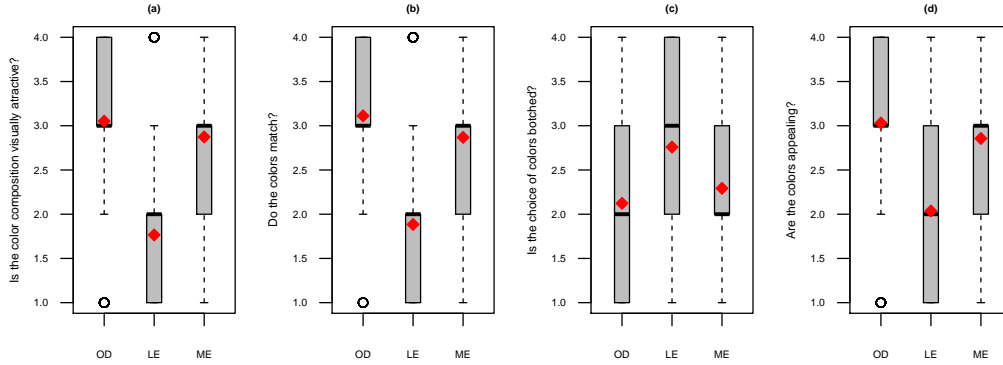


Figure 5.16: RQ₂: Boxplots of answers provided by participants. OD=Original Design, LE=Lowest ECF, ME=Median ECF.

Table 5.7: RQ₂: Wilcoxon test (p -value) and Cliff’s delta (d).

Test	adj. p -value	d
Is the color composition visually attractive?		
original design vs lowest ECF	<0.01	-0.66 (Large)
original design vs median ECF	<0.01	-0.12 (Small)
Do the colors match?		
original design vs lowest ECF	<0.01	-0.65 (Large)
original design vs median ECF	<0.01	-0.16 (Small)
Is the choice of colors botched?		
original design vs lowest ECF	<0.01	0.33 (Small)
original design vs median ECF	<0.01	0.10 (Small)
Are the colors appealing?		
original design vs lowest ECF	<0.01	-0.55 (Large)
original design vs median ECF	<0.01	-0.12 (Small)

(*i.e.*, $\bar{B}_{(step)}$) collected for the two versions (original, and lowest energy GUI consumption) of the five analyzed apps. It is worth nothing that the values of series for current, power, energy, and battery consumption have the same trend; the differences in the series are a dimensions change with a scalar multiplication over the original series (*i.e.*, current). Therefore, there is no need to plot all the series, however, we plot the average battery consumption to give the reader values that are more natural (compared to mA or mW) and easier to be understood. In addition, we computed the time (in hours) required to drain (t_{drain})the battery of the GS4 when replaying the execution scenarios continuously, on the original version of the apps and the lowest energy GUI consumption provided by

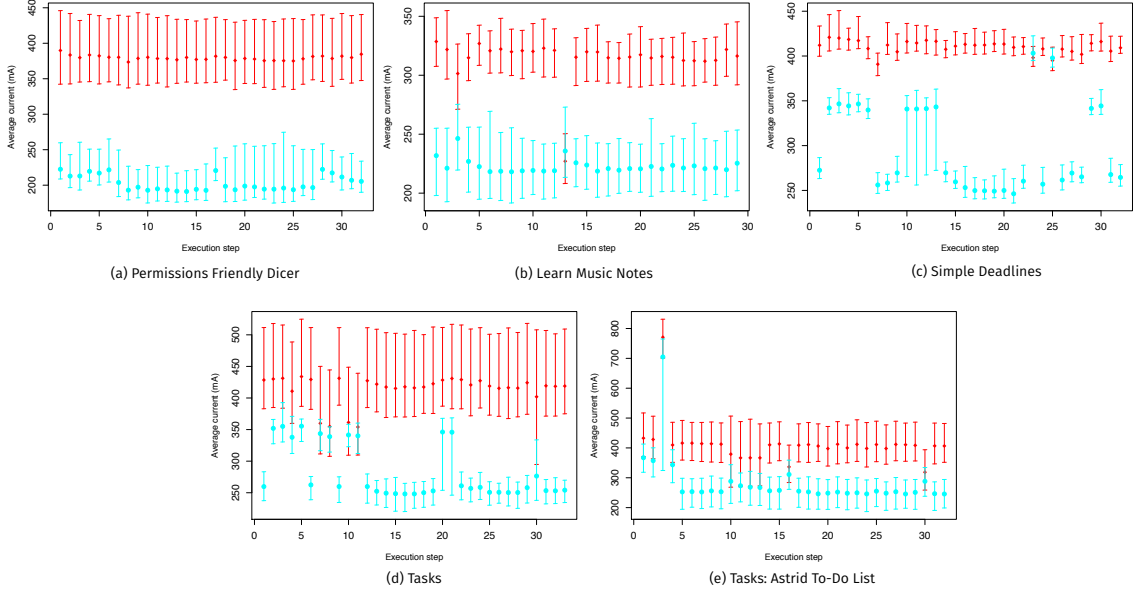


Figure 5.17: Current (mA) drawn by the analyzed apps. Each plot depicts the range (*i.e.*, min to max) of the average current for each step across the 30 executions for (i) original app (in red), and (ii) modified version with GEMMA GUI composition for low energy-consumption (in cyan). The circles depict the averages in each range.

GEMMA; to this we computed the time-to-drain-battery similarly to [216, 215]:

$$t_{drain} = \frac{100}{\sum_{step \in Script} \bar{B}_{(step)}} \times duration(secs) \times \frac{1hour}{3600secs} \quad (5.15)$$

It is worth noting that *duration* is the total time in seconds of the sampling period during the execution of a *Script*; Equation 5.15 computes t_{drain} based on the total percentage of battery consumption, *i.e.*, the sum of the battery consumption over all the steps in an execution *Script*, therefore, the 100 figure in the numerator of equation 5.15 is to account for the percentage values in the denominator.

The results are listed in Table 5.9, and for the five analyzed apps, when using the lowest ECF solution, there is a clear improvement in terms of the time required to drain the battery, *i.e.*, the GEMMA solution improved the battery life.

Note that there are some cases shown in the plots in which there is no significant

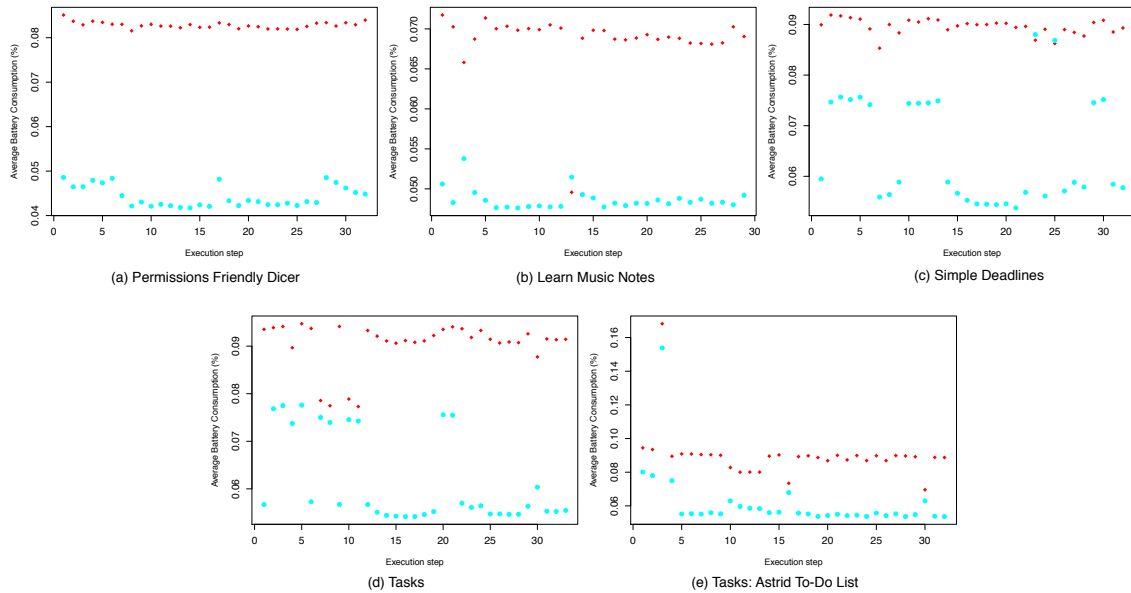


Figure 5.18: Percentage of battery consumed by the analyzed apps. Each plot depicts the average consumption per each step across the 30 executions for (i) original app (in red), and (ii) modified version with GEMMA GUI composition for low energy-consumption (in cyan).

improvement: (i) step 13 for the Learn Music Notes app Figure 5.17-(b); (ii) steps 23 and 25 for the Simple Deadlines app—Figure 5.17-(c); and steps 16 and 30 for the Tasks: Astrid To-Do List app—Figure 5.17-(e). In the case of step 13 of Learn Music Notes, the percentage of consumed energy is also in favor of the original design; the reason for this surprising result is that step 13 started a large black-background dialog that is displayed over the main GUI; therefore, the changes to the GUI do not have a big impact (in terms of energy consumption) after step 13, because the black dialog is painted on most of the screen. The case of the Simple Deadlines app is illustrated in Figure 5.19; the steps included displaying a drawer GUI component (with white background) which occupies a large portion of the screen. The screens collected by GEMMA during the analysis of Simple Deadlines did not include the drawer, therefore, the color compositions proposed by GEMMA did not consider the drawer. The case of Tasks: Astrid To-Do List is similar to Simple Deadlines; steps 16 and 30 display a modal dialog with white background that

Table 5.8: RQ₃: Wilcoxon test (p -value) and Cliff’s delta (d) for pairwise comparisons of energy measurements (current in mA): **original design** vs lowest ECF

App	p -value	Cliff’s d
Privacy Friendly Dicer	<0.01	0.97 (Large)
Learn Music Notes	<0.01	0.99 (Large)
Simple Deadlines	<0.01	0.96 (Large)
Tasks	<0.01	0.96 (Large)
Tasks: Astrid To-Do List	<0.01	0.92 (Large)

Table 5.9: Battery life (in hours) when using the **original version** of a mobile application, and app modified with the GEMMA solution with the lowest energy consumption (lowest ECF). The column *Diff.* lists the relative difference in percentage, i.e., $\frac{\text{lowestECF} - \text{original}}{\text{original}}$.

Application	Original Design	Lowest ECF	Diff.
Privacy Friendly Dicer	6.707608	12.53452	86.87%
Learn Music Notes	8.093131	11.41668	41.07%
Simple Deadlines	6.201243	8.570963	38.21%
Tasks	6.145774	9.009598	46.60%
Tasks: Astrid To-Do List	6.182163	9.052713	46.43%

was not analyzed by GEMMA (Figure 5.20).

In summary, there is a substantial difference between the energy measurements (*i.e.*, current and battery consumption) of the original design and the color composition recommended by GEMMA. The strong difference is confirmed by the Wilcoxon test and the Cliff’s delta effect size (listed in Table 5.8). The energy savings with GEMMA’s GUIs is statistically significant with a large effect size for all five apps analyzed in our study.

5.4.4 RQ₄: Would actual developers of mobile applications consider changing colors in an app as recommended by GEMMA?

In the following, we report the results of the interviews we conducted with project managers of three Italian companies aiming at analyzing the practical applicability of GEMMA in a real development context. As explained in Section 5.3.4, we asked the participants to answer the following question: “Given the energy saving provided by this design with respect

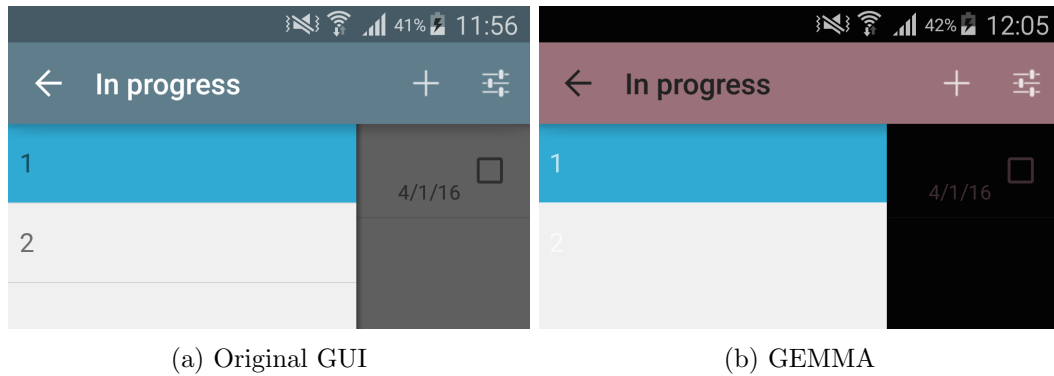


Figure 5.19: Example of GUI without significant improvement in the energy consumption for the Simple Deadlines app.

to the original design, would you adopt it in your app?”, using a score on a four-point Likert scale: 1=absolutely no, 2=no, 3=yes, 4=absolutely yes.

Next. The first person we interviewed was Nicola Noviello, the project manager of Next. Such a company provided us with two apps, namely *Bollate* [10] and *Petrella* [38]. The two apps are from the category “Places and travels”, and provide tourist with information on two Italian towns. Before starting the interview, Nicola confirmed that GEMMA might have a high practical applicability even if it would be important to assess which percentage of the overall app energy consumption is actually amenable to its GUI, and thus could be optimized by GEMMA. This is something we plan to explore more in the future. As for the evaluation of the solutions provided by GEMMA for the *Bollate* app, we showed to Nicola eleven design alternatives, asking him if he would adopt them in his app. None of them was considered acceptable (answer “*absolutely not*”) despite the average energy saving around 85%. The reason why Nicola discarded such alternatives is represented by the color of the background. All the alternatives provided by GEMMA had a black background. In the context of Nicola’s experience, “*it is hard to make attractive an interface with a black background*”. Specifically, Nicola claimed that he uses a black background only if it is explicitly required by the customer. Otherwise, he designs apps with more vivid colors even if he recognized that this could negatively impact the energy consumption of the app. This suggests an interesting future direction to explore, *i.e.*,

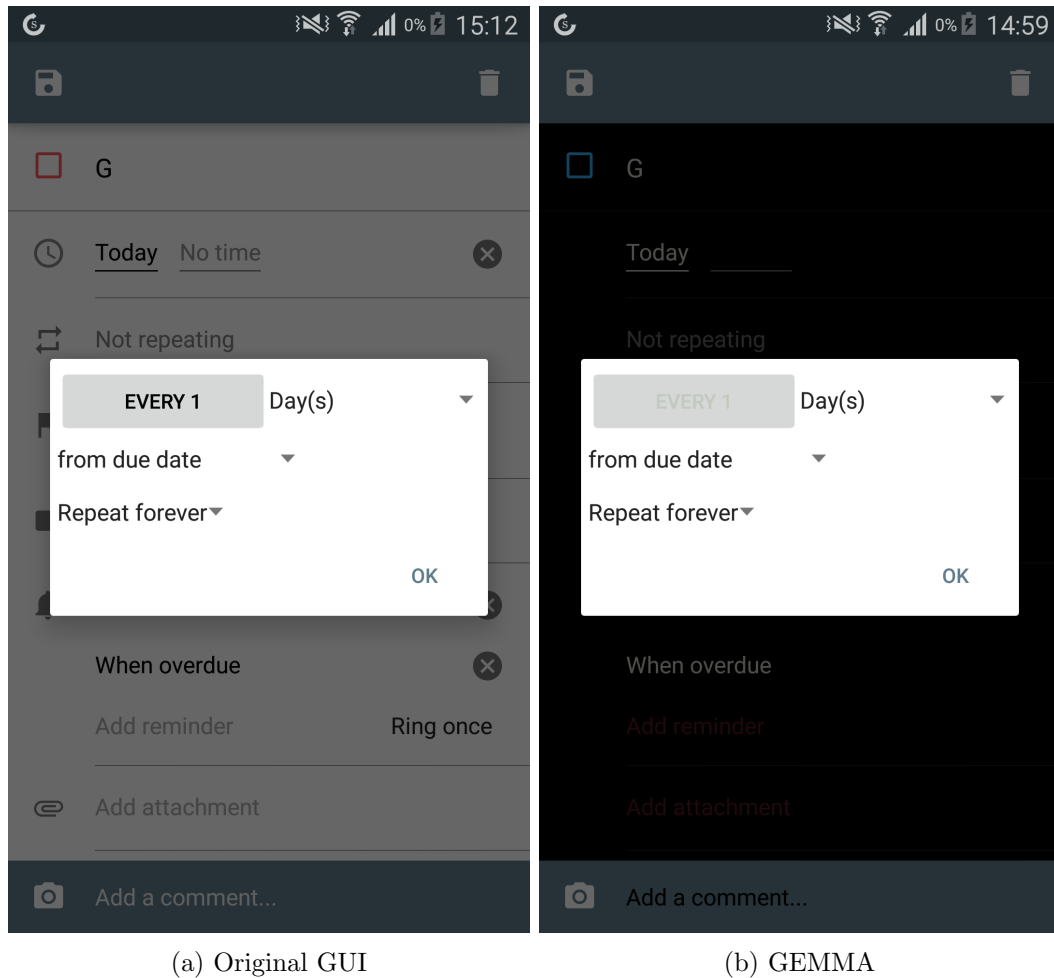


Figure 5.20: Example of GUI without significant improvement in the energy consumption for the Tasks: Astrid To-Do List app.

usage of interactive GAs, where the developer can be integrated in the evolutionary loop and set personalized constraints (*e.g.*, do not use this color for this component) during the generation of solutions. The results were different for the Petrella app. Among the five different alternatives, Nicola considered three as good alternatives to the original design, claiming that he would consider their adoption (answer “*yes*”). It is worth noting that the solution considered the best by Nicola is also the one that provides the highest energy savings (71%). For this app, Nicola particularly appreciated the recommended solutions due to their high similarity with the original design. This characteristic allows obtaining



Figure 5.21: IdeaSoftware app: Original design *vs* GEMMA’s solution (excerpts).

substantial energy savings just by performing small adjustments to the original GUI. In addition, Nicola claimed that *“even if the colors of these alternatives are less vivid than the original design, I believe that the proposed combinations—with some small adjustments—will make the app attractive”*.

Genialapps. The second person we interviewed was Giuseppe Socci, the project manager of Genialapps. The app that they provided was Sing Happy Birthday Songs (HBS) [45], which has around 2,000 ratings on the Google Play market and a number of installations between 100K and 500K. The app is from the category “Music and Songs” and it can be used to send happy birthday wishes with a personalized phone call. Giuseppe was particularly interested in GEMMA. He claimed that such an approach might have an high industrial impact. We provided him with a set of 46 different alternatives. Among them, 29 were discarded as not adoptable as alternative to the original design (answer *“absolutely not”*). The reason why Giuseppe did not like those alternatives is that *“the colors are too dull and in a joyful app, like HBS, colors must be vivid”*. The other 17 alternatives were considered good ones, with three of them gathering an *“absolutely yes”* to the posed question (see Section 5.3.4). One of these three solutions provides an energy savings of 63%. Such an alternative, again with some small adjustments, was considered by Giuseppe an excellent solution for his app. During the interview, Giuseppe also pointed out an important issue that approaches like GEMMA should take into account. GEMMA is particularly useful *“for apps that are used for a rather long period of time (e.g., social network app). Although it is true that many drops make the ocean, and therefore, in our*

case, many apps not optimized used for a short time still consume a lot of energy, it is also true that developers are not very likely to change the GUI aiming at saving battery. The reason is that this non-functional requirement is quite difficult to sell, unless the app is used for rather long time periods and thus the customer has a tangible evidence of the energy saved.” This consideration is perfectly inline with our weighted model for power consumption, in which we give more importance to screens that are used more. As well as Nicola, Giuseppe is also interested in “the actual impact that the colors of the GUI have on the overall energy consumption of the app”. He also pointed out that “in the case of HBS, since it has a lightweight logic (the server is in charge of the application logic) I suppose that the influence of the GUI is considerable”.

IdeaSoftware. The last person we interviewed was Luciano Cutone, co-founder and the project manager of IdeaSoftware. The company requested to anonymize both of the apps that they provided for our study. In the following, we refer to such apps as App1 and App2. Both apps have thousands of installations on the Google Play market. Luciano was particularly interested in GEMMA, that he defined as “a tool that should be integrated in an IDE and used daily”. This confirms the high industrial impact that GEMMA might have. As for App1, we provided to Luciano a set of 15 different alternatives. Among them, 8 were not considered acceptable (answer “*absolutely not*”) by Luciano because “the combination of colors was not exciting”. The other 7 solutions were instead considered as good alternative to the original design (answer “*absolutely yes*”). Luciano particularly liked one solution that is reported, compared to the original design, in Figure 5.21. Due to the need for anonymization, we only report a portion of the original and recommended GUIs. Luciano claimed “I would definitively use this combination of colors in my app. The final result is excellent and I really like the effect of the GUI with a black background. This helps in saving battery and makes the app more elegant. I will propose the new combination of colors for the next release of the app”. Such a consideration highlights different points of view by Luciano and Nicola about black background emphasizing that the choice of colors is subjective and, as mentioned before, an interactive version of GEMMA could

be worthwhile. For the App2, we only provided three different solutions. In addition, the average energy savings for such solutions is around 10%, due to the already energy efficient original design. Indeed, as pointed out by Luciano, when designing the GUI the developers used low bright colors in order to (i) make the app more professional and (ii) save the battery of the phone since the app was designed to be used continuously during the day. Such considerations confirmed that the energy consumption is a problem relevant for industry when the app is used frequently. Nevertheless, among the three solutions, Luciano classified two of them as good alternatives (answer “yes”).

5.5 Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation, and in this work are mainly related to the measurements we performed in our study. One major threat is that the measures of contrast and pleasant design—*i.e.*, the $CF(S)$ and $DF(S)$ objectives—which we use in the optimization process and quantitatively compare with values of the original colors (**RQ₁**) might not represent a proxy of what actually is perceived by users. **RQ₂**, within its generalizability limitations, is intended to mitigate such a threat. Another possible threat is related to the fact that we used power models to estimate energy consumption. So far, such models have been already used in previous studies and considered reliable enough, and our models are consistent with the ones by Dong *et al.* [90, 91] for OLED screens.

Threats to *internal validity* are related to factors, internal to our study, that can influence our results. We have used GA settings that are frequently used in the research literature [88, 94, 99], and calibrated the number of evolutions so that longer evolutions did not produce better results. Of course, we cannot exclude that a better calibration would produce better results. Last, but not least, we accounted for GA randomness by executing GEMMA 30 times [63].

Threats to *conclusion validity* concern the relationship between experimentation and

outcome. For **RQ₁** and **RQ₂**, as described in Section 5.3.5, we have quantitatively assessed our results using appropriate statistical procedures. Instead, **RQ₃** is exploratory in nature, intended to collect some preliminary feedback about the practical applicability of GEMMA in a real development context.

Threats to *external validity* are related to the generalizability of our findings. We are aware that our results have to be interpreted carefully because (i) they may depend on the specific device for which we extracted the color power model; (ii) results of **RQ₂** may depend on people’s preferences, and other people might have different opinion on the chosen colors, and (iii) although we applied GEMMA on 25 different apps, we are aware that GEMMA might produce different results in terms of energy consumption reduction and colorfulness on a different set of apps.

5.6 Related Work

GEMMA is mostly related to previous work on energy optimization of GUIs in mobile apps as well as approaches for detecting energy greedy units in such apps. Other approaches have focused on (i) measuring/estimating energy consumption of Java APIs [126], (ii) measuring the impact of refactorings on energy consumption of Java apps [214], (iii) empirical studies with practitioners regarding practices and perspectives for green software engineering [182], (iv) optimizing energy consumption of test suites [153, 136], and (v) infrastructures/frameworks for energy measurement/green mining [127, 130, 183, 129]. However, because GEMMA was developed in the context of Android apps, in this section we only discuss previous work on energy consumption and optimization of Android apps.

5.6.1 Improving Energy Consumption of GUIs

OLED screens are suitable for optimizing energy consumption of GUIs in apps, because the power consumed by OLED screens depends on the combination of color levels in the screen’s sub-pixels. Therefore, power models of OLED screens estimate the energy by

Table 5.10: Energy Optimization of Android Apps

Approach	Opt. problem	Color palettes
[91]	<i>min</i> energy, <i>s.t.</i> similar contrast than in original GUI	1) Predefined themes, 2) Monochromatic palette, 3) black assigned to most frequent color in original GUI, then subsets of R,G,B assigned to the rest of pixels
[154, 235, 157]	<i>min</i> energy, <i>s.t.</i> keep color distance between neighboring HTML elements	Background replaced with dark colors, randomly selected colors are assigned to the rest of pixels
GEMMA	<i>min</i> energy, <i>max</i> contrast, <i>min</i> distance to original design, <i>s.t.</i> contrast > 40	Palette contains 512 colors: original + black + white + equidistant harmonies and monochromatic palette)

combining individual consumption of the color sub-pixels [135, 240]. These models are particularly useful, since they allow researchers to estimate energy consumption without using expensive power monitors. Thus, power models have been used for energy saving visualization of sequential data [237], design of energy adaptive displays [135], and design of color-adaptive browsers [90].

As for mobile apps, representative works are the ones by Dong and Zhong [91] for Windows mobile apps, Li *et al.*'s approach [154] for mobile web applications, and Wan *et al.*'s work [235] on screenshots of Android apps. The three related approaches and GEMMA are compared in Table 5.10. While we share the goals of minimizing energy consumption through properly choosing colors and using color power models with these approaches, GEMMA introduces novel aspects related to (i) multi-objective optimization, (ii) considering contribution of different screens based on their usage duration, and (iii) ensuring a pleasant and consistent choice of colors.

5.6.2 Detecting Energy Bugs in Mobile Apps

Energy bugs and energy hot spots in Android apps—at different granularities—have been catalogued extensively. For instance, Pathak *et al.* [204, 207] describe a taxonomy of energy bugs that depends on the hardware, software, or external conditions. Kwon and Tilevich [150] focused on cloud offloading energy consumption. Pathak *et al.* [205] and

Linares-Vásquez *et al.* [165] reported energy greedy Android APIs. Hao *et al.* [123] and Li *et al.* [151] identified energy greedy apps, meanwhile Li *et al.* [152] focused on energy measurement at code statement level. Liu *et al.* [173] detected energy bugs related to misuses of wake locks and sensors. Li *et al.* [155, 156] optimize the energy consumed by HTTP requests in Android apps, by automatically detecting and bundling sequential HTTP requests. Finally, Rasmussen *et al.* [210] and Gui *et al.* [118] measured the impact of ads in apps on energy consumption.

In addition to energy bugs and hot spots, energy consumption has been analyzed from the perspective of other factors such as obfuscation [216] and performance-related changes [215]. Sahin *et al.* [216], in a study with 15 apps, showed that obfuscation can impact—with statistical significance—the energy consumption of Android apps; and in a study with 8 apps, Sahin *et al.* [215], found that the impact of 4 performance-related optimizations (use final for static constants, avoid using floating point, avoid internal getters/setters, and avoid accessing array length in loop body.) is negligible in terms of batter savings. Another representative work is Ecodroid [137], which is a code search engine for Android apps that ranks apps in the same category based on energy consumption; Ecodroid combines static and dynamic analysis and estimates the apps energy consumption by aggregating the energy consumed by the API calls.

All the approaches mentioned in this section detect different kinds of energy-greedy units in the source code, yet not specifically related to the screen usage. For this reason, GEMMA can be considered complementary to those approaches: in other words, reducing the energy footprint of mobile apps concerns considering various aspects, including, among the others, screen usage and colors.

5.7 Discussion

We presented GEMMA, a multi-objective approach for generating energy-friendly color schemas for Android app GUIs. The multi-objective optimization balances the energy

reduction objective with other objectives related to contrast and closeness to the original design. GEMMA successfully generated designs for 27 Android apps with significant reduction in the energy consumption. While the empirical evaluation highlighted that solutions with the highest energy savings are usually not preferred by end-users, mainly because of the dark background, GEMMA still generated solutions that achieve a good energy reduction while being acceptable to end-users. Also, by evaluating the solutions on five commercial apps, we confirmed that some project managers and developers are ready to use GEMMA's recommendations in future app releases. In the future work, we are planning on relying on a more precise approach for reverse engineering app GUIs, which would account for a proper choice of colors for GUI elements such as text fields and buttons.

GEMMA is an example of (i) how dynamic analysis is an important task for evolution and maintenance of Android apps and, and (ii) how an underlying infrastructure for dynamic analysis can enable automated tasks that otherwise can not be achieved manually. The relevance of quality attributes such as energy consumption, from the point of view of the users and energy-aware developers, makes tools like GEMMA interesting to the industry. Currently, an industrial prototype of GEMMA is under development as part of a initiative between William and Mary and Huawei aimed at providing developers with free tools that support evolution and maintenance of mobile apps.

5.8 Bibliographical Notes

The papers supporting the content described in this Chapter were done in collaboration with other members of the SEMERU group at William and Mary, and researchers from University of Sannio, University of Bozen-Bolzano, and University of Molise:

- **Linares-Vásquez, M.**, Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D., “Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach” in Proceedings of 10th Joint Meeting of the European

Software Engineering Conference and the 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15), Bergamo, Italy, August 31-September 4, 2015, pp. 143-154 (25.4% acceptance ratio). **ACM Distinguished Paper Award.**

- **Linares-Vásquez, M.**, Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Di Penta, M., and Poshyvanyk, D., “Visual Aesthetics Matter: Multi Objective Optimization of Energy Consumption of GUIs in Android Apps”, under review.

Chapter 6

Conclusion

This dissertation makes several research contributions from three different perspectives: empirical studies, technological solutions, and novel practical approaches for supporting evolution and maintenance of Android apps:

- The empirical studies presented in Chapter 2 analyzed key challenges experienced by practitioners and open issues in the mobile development community such as (i) Android API instability, (ii) performance optimizations, (iii) automatic GUI testing, and (iv) energy consumption; consequently, the results from the studies can be used by practitioners and researchers to design new methods, tools and approaches for supporting evolution and maintenance of Android apps (similarly to the work done with MONKEYLAB and GEMMA).
- This dissertation proposes an infrastructure for enabling large-scale execution of Android apps in Chapter 3. This infrastructure was designed with several design drivers in mind to enable extensibility (open-closed principle), horizontal/vertical scalability, and easy implementation with commodity machines and open source software. A taxonomy of issues, solutions, and guidelines are also provided to support researchers interested in enabling large-scale execution of Android apps on their own environments.

- Two novel solutions are proposed to solve highly relevant tasks during evolution and maintenance of Android apps: automated testing and energy optimizations. For automated testing, we proposed an approach that combines GUI and usage models (extracted from the source code and real application usages) to generate actionable (un)natural test cases in Chapter 4. The test sequences are generated by relying on language models that are aware of events history and are validated dynamically to remove unfeasible events. The validation shows that the proposed approach generates (un)natural test cases that were neither exercised by human participants nor by the state-of-the-art approaches. We also proposed an approach for automatic generation of GUI color compositions that reduce screen energy consumption and are visually appealing in Chapter 5; the solution is rooted into our novel combination of power modeling, dynamic analysis, pixel-based engineering, color-theory, and evolutionary multi-objective optimization. The experiments confirmed that the color compositions intended to reduce drastically the energy consumption generate large and significant savings (when compared to the original design); in addition, the compositions are visually attractive and mobile development companies are eager to use them.

In addition, this dissertation establishes a future roadmap of research and development activities. First, the empirical studies highlighted challenges that still require novel solutions; secondly, the foundations we established in this dissertation allow extensions, in particular in the field of automated GUI testing. We briefly summarize some of those activities.

A General Infrastructure for Enabling Testing of Android Apps. The *Record* \rightarrow *Mine* \rightarrow *Generate* \rightarrow *Validate* framework introduced in Chapter 4 is based on a modular design, in which the component in charge of generating the test sequences can be replaced to derive test sequences with different purposes. Figure 6.1 illustrates the case in which the *Generate* phase can be tailored to different purposes while reusing

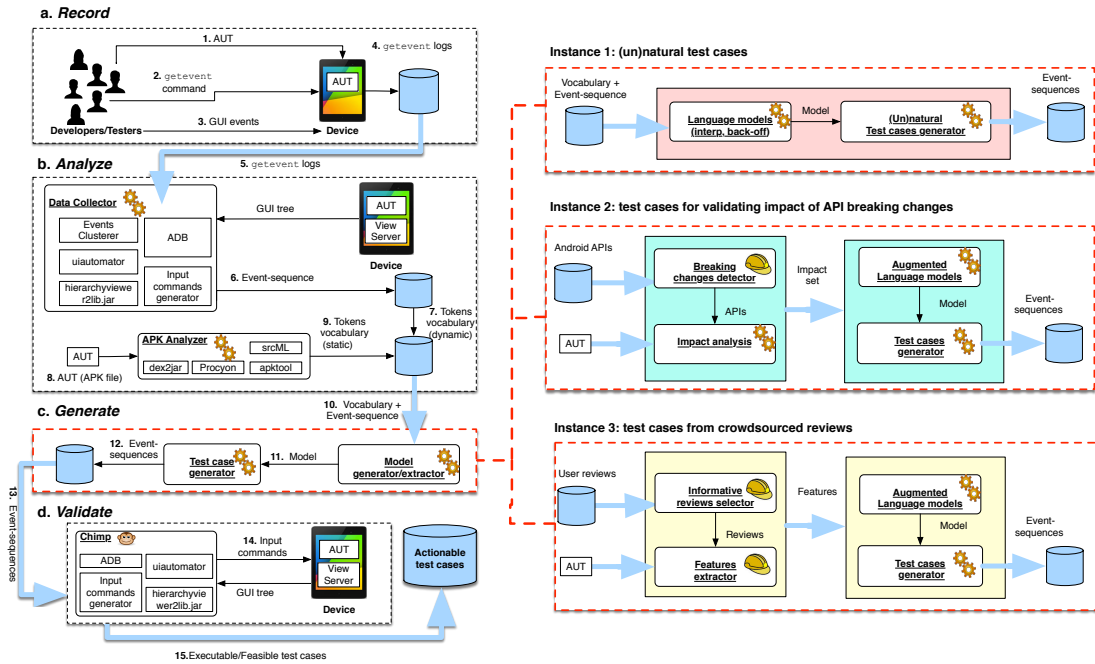


Figure 6.1: A general infrastructure for generating test sequences with different purposes the components and features from the other phases. MONKEYLAB is the first instance described in Chapter 4, which is aimed at generating (un)natural test cases by relying on language models. However, other types of test sequences can be generated using the same infrastructure.

With the empirical studies (Chapter 2), we identified that API instability and fault-proneness are the major threats to the success of Android apps. When a new API is released, developers should conduct regression testing on the apps to verify that the new API does not inject bugs in the client code (i.e., apps). Therefore, by mining the changes to the APIs and performing static analysis on the source code of the apps, it is possible to drive the test sequence generations towards executing features that are likely to be impacted by API-breaking changes. In addition, informative reviews reporting bugs/crashes could be analyzed to generate test cases aiming at testing features/GUI components involved in the bug/crashes reported by users. Both scenarios can be implemented by augmenting/extending the *Generate* phase in the proposed infrastructure.

Combining Multiple Models For Test Cases Generation. Requirements of a software system are often modeled as a combination of three essential models in Object-Oriented Software Engineering approaches [57, 58]: (i) a *usage model* that describes how users will work with the system (e.g, use cases or user stories); (ii) a *domain model* that describes the business entities, relationships, and data constraints (e.g., class diagram or database model); and (iii) a *GUI model* depicting the user interface. Therefore, those three models are the main sources for designing test cases. In fact, deriving test cases from the usage model is a well-promoted practice in Object-Oriented Software Engineering approaches as a systematic way to validate requirements [71, 85, 138]. In the case of mobile apps, there is a fourth model (*contextual model*) that describes [195] the events/states related to contextual events (e.g., GPS and WiFi) and adversarial conditions.

Few approaches for automated testing of mobile apps combined two of the models for requirements specifications (*i.e.*, usage, GUI, domain, contextual). For instance, MONKEYLAB combines usage and GUI models, and SIGDROID [188] combines GUI and domain models. CrashScope by Moran *et al.* [195] is a recent publication of the SEMERU group!¹ that combines the GUI and contextual models. As of today, there is no complete solution/approach combining the four models (*i.e.*, GUI, usage, domain, and contextual); therefore, future work should be devoted to building solutions for automated testing that combine all four models. One potential solution is to reuse the *Record* \rightarrow *Mine* \rightarrow *Generate* \rightarrow *Validate* framework, and to design graphical probabilistic models that are able to consolidate the information from the GUI, usage, domain, and contextual models.

Supporting Other Evolution and Maintenance Tasks. Some of the empirical studies described in Chapter 2 suggest that developers lack the tools for (i) automated detection of features in the apps that might be affected by breaking changes, and (ii) automatic detection of performance bottlenecks following observation-based strategies (e.g., profiling and debugging). Therefore, future work related to evolution and maintenance of Android

¹CrashScope is supported by the SEMERU framework and infrastructure designed as part of this dissertation (Chapter 3)

apps should be aimed at devising approaches and solutions for these two aforementioned issues.

Bibliography

- [1] Amazon ec2 cloud computing service <https://aws.amazon.com/ec2/>.
- [2] Android Monkey Recorder. <http://code.google.com/p/android-monkeyrunner-enhanced/>.
- [3] Android x86 project. <http://www.android-x86.org>.
- [4] Aosp compilation instructions. <https://source.android.com/source/initializing.html>.
- [5] Aosp v4.4.2. https://android.googlesource.com/platform/build/+android-4.4.2_r1.
- [6] Apache couch db. <http://couchdb.apache.org>.
- [7] Apktool. <https://code.google.com/p/android-apktool/>.
- [8] At&t application resource optimizer. <http://developer.att.com/application-resource-optimizer>.
- [9] Best practices for performance. <http://developer.android.com/training/best-performance.html>.
- [10] Bollate. <https://play.google.com/store/apps/details?id=com.comunicazione360.bollate>.
- [11] Changing default port (i.e. 5037) on which adb server runs. <http://goo.gl/RAEdY5>.

- [12] Class file that implements the `mapprtransitionstate` property. <https://goo.gl/Wf50PD>.
- [13] Contrast ratio (luminance). http://www.w3.org/WAI/GL/wiki/Contrast_ratio.
- [14] Couchdb wiki - attachments. https://wiki.apache.org/couchdb/HTTP_Document_API#Attachments.
- [15] Description of `mapprtransitionstate` properties <https://modusoperandi4android.wordpress.com>.
- [16] `dex2jar`. <https://code.google.com/p/dex2jar/>.
- [17] Eclipse memory analyzer (`mat`). <https://eclipse.org/mat/>.
- [18] Emma: a free java code coverage tool. <http://emma.sourceforge.net/>.
- [19] Espresso-loopmainthreaduntilidle <https://goo.gl/jKXIoz>.
- [20] Experience report online appendix <http://www.research-appendix.com/ase16-android-testing>.
- [21] F-droid. <https://f-droid.org/>.
- [22] Genial apps website. <http://www.genialapps.eu/portale/>.
- [23] Genymotion <https://www.genymotion.com/>.
- [24] Google play. <https://play.google.com/store>.
- [25] How developers fix performance bottlenecks in android apps – online appendix. <http://www.cs.wm.edu/semeru/data/ICSME15-Android-bottlenecks>.
- [26] How developers micro-optimize android apps – online appendix. <http://www.cs.wm.edu/semeru/data/EMSE-Android-opt>.

- [27] How expensive findViewById ?? https://groups.google.com/forum/?fromgroups=#!topic/android-developers/_22Z90dshoM.
- [28] Ideasoftware website. <http://lnx.space-service.it>.
- [29] Intel parallel studio. <https://software.intel.com/en-us/intel-parallel-studio-xe>.
- [30] Junit. <http://junit.org>.
- [31] Linode cloud computing service <https://www.linode.com>.
- [32] Lint. <http://developer.android.com/tools/help/lint.html>.
- [33] Littleeye. <http://www.littleeye.co/>.
- [34] Mobile gaming & graphics (adreno) tools and resources. <https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno/tools-and-resources>.
- [35] Next website. <http://www.nextopenspace.it/>.
- [36] Official google android emulator <http://developer.android.com/tools/help/emulator.html>.
- [37] Openesb. <http://www.open-esb.net>.
- [38] Petrella. <https://play.google.com/store/apps/details?id=it.nextlabs.platform.petrella>.
- [39] Pmd. <http://pmd.sourceforge.net>.
- [40] Proguard. <http://proguard.sourceforge.net/>.
- [41] Rabbitmq. <https://www.rabbitmq.com>.
- [42] Ravello systems cloud emulator solution <https://goo.gl/udyZMn>.

- [43] Robotium. <https://code.google.com/p/robotium/>.
- [44] Sending keyboard input via adb to your android device. <http://blog.rungeek.com/post/42456936947/sending-keyboard-input-via-adb-to-your-android>.
- [45] Sing happy birthday songs. <http://happybirthdayshow.net/en/>.
- [46] Solution for dynamic inferral of gui idle states <http://goo.gl/NVWLFt>.
- [47] Solution for enabling concurrent execution of avds <http://goo.gl/apZ1MC>.
- [48] Solution to effectively clean app data during testing <http://goo.gl/uw7yRq>.
- [49] Solutions for the uiautomator android tool <http://goo.gl/679zsu>.
- [50] Srcml. <http://www.sdml.info/projects/srcml/>.
- [51] Valgrind. <http://valgrind.org/>.
- [52] Virtualbox vm platform <https://www.virtualbox.org>.
- [53] S. ABOLFAZLI, Z. SANAEI, E. AHMED, A. GANI, AND R. BUYYA. Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges. *CST*, 16(1):337–368, 2014.
- [54] P. AHO, M. SUAREZ, T. KANSTREN, AND A.M. MEMON. Murphy tools: Utilizing extracted gui models for industrial software testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 343–348, 2014.
- [55] D. AMALFITANO, A.R. FASOLINO, P. TRAMONTANA, S. DE CARMINE, AND A.M. MEMON. Using gui ripping for automated testing of android applications. In *International Conference on Automated Software Engineering (ASE'12)*, 258–261, editor, 2012.

- [56] DOMENICO AMALFITANO, ANNA RITA FASOLINO, PORFIRIO TRAMONTANA, BRYAN DZUNG TA, AND ATIF MEMON. Mobiguitar - a tool for automated model-based testing of mobile apps. *IEEE Software*, page to appear, 2014.
- [57] SCOTT AMBLER. *Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process*. John Wiley & Sons, Inc., New York, NY, USA, 2002.
- [58] SCOTT W. AMBLER. *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, New York, NY, USA, 2004.
- [59] S. ANAND, M. NAIK, M. J. HARROLD, AND H. YANG. Automated concolic testing of smartphone apps. In *FSE'12*, 2012.
- [60] ANDROID DEVELOPERS. Android performance patterns. <https://www.youtube.com/playlist?list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE>.
- [61] G. ANTONIOL, G. CANFORA, G. CASAZZA, A. DE LUCIA, AND E. MERLO. Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, 28(10):970–983, 2002.
- [62] APPLE. ios performance tips. <https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/PerformanceTips/PerformanceTips.html>.
- [63] ANDREA ARCURI AND LIONEL BRIAND. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 1–10, New York, NY, USA, 2011. ACM.
- [64] ANNE AUGER, JOHANNES BADER, DIMO BROCKHOFF, AND ECKART ZITZLER. Theory of the hypervolume indicator: optimal ϵ -distributions and the choice of the reference point. In *Proceedings of the tenth ACM SIGEVO workshop on Foundations of genetic algorithms, FOGA '09*, pages 87–102. ACM, 2009.

- [65] VITALII AVDIENKO, KONSTANTIN KUZNETSOV, ALESSANDRA GORLA, ANDREAS ZELLER, STEVEN ARZT, SIEGFRIED RASTHOFER, AND ERIC BODDEN. Mining apps for abnormal usage of sensitive data. In *ICSE'15*.
- [66] N. AYEWAH, D. HOVEMEYER, J.D. MORGENTHALER, J. PENIX, AND WILLIAM PUGH. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
- [67] NATHANIEL AYEWAH AND WILLIAM PUGH. The google findbugs fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10*, pages 241–252, New York, NY, USA, 2010. ACM.
- [68] T. AZIM AND I. NEAMTIU. Targeted and depth-first exploration for systematic testing of android apps. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, pages 641–660, 2013.
- [69] G. BAVOTA, M. LINARES-VÁSQUEZ, C. E. BERNAL-CÁRDENAS, M. D. PENTA, R. OLIVETO, AND D. POSHYVANYK. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering*, 41(4):384–407, April 2015.
- [70] MICHAEL BIERMA, ERIC GUSTAFSON, JEREMY ERICKSON, DAVID FRITZ, AND YUNG RYN CHOE. Andlantis: Large-scale android dynamic analysis. In *MoST'14*, 2014.
- [71] ROBERT V. BINDER. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [72] JR. BROOKS, F.P. No silver bullet essence and accidents of software engineering. *Computer*, 20(4), April 1987.
- [73] FRED. P. BROOKS, JR. *The Mythical Man-Month: Essays on Software Engineering*. 1995, Addison-Wesley.

- [74] IKER BURGUERA, URKO ZURUTUZA, AND SIMIN NADJM-TEHRANI. Crowdroid: Behavior-based malware detection system for android. In *SPSM'11*, pages 15–26, 2011.
- [75] C. CHEN K. KHOR C. YEH, H. LU AND S. HUANG. Craxdroid: Automatic android system testing by selective symbolic execution. In *IEEE Eighth International Conference on Software Security and Reliability (SERE-C)*, pages 140–148, 2014.
- [76] L. V. GALVIS CARRENO AND K. WINBLADH. Analysis of user comments: An approach for software requirements evolution. In *35th International Conference on Software Engineering (ICSE'13)*, pages 582–591, 2013.
- [77] A. CARROLL AND G. HEISER. An analysis of power consumption in a smartphone. In *USENIX ATC'10*, 2010.
- [78] N. CHEN, J. LIN, S. HOI, X. XIAO, AND B. ZHANG. AR-Miner: Mining informative reviews for developers from mobile app marketplace. In *36th International Conference on Software Engineering (ICSE'14)*, pages 767–778, 2014.
- [79] STANLEY F. CHEN AND JOSHUA GOODMAN. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics, ACL '96*, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.
- [80] W. CHOI, G. NECULA, AND K. SEN. Guided gui testing of android apps with minimal restart and approximate learning. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, pages 623–640, 2013.
- [81] SHAUVIK ROY CHOUDHARY, ALESSANDRA GORLA, AND ALESSANDRO ORSO. Automated test input generation for android:are we there yet? In *ASE'15*, page to appear, 2015.

- [82] Y.-F. CHUNG, CH-Y.LIN, AND CH.-T. KING. Aneprof: Energy profiling for android java virtual machine and applications. In *ICPADS'11*, pages 372–379, 2011.
- [83] KENNETH CHURCH AND WILLIAM GALE. A comparison of the enhanced good-turing and deleted estimation methods for estimating probabilities of english bigrams. *Computer Speech and Language*, 5:19–54, 1991.
- [84] W. J. CONOVER. *Practical Nonparametric Statistics*. Wiley, 3rd edition edition, 1998.
- [85] LISA CRISPIN AND JANET GREGORY. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Addison-Wesley Professional, 1 edition, 2009.
- [86] V. DALLMEIER, N. KNOPP, C. MALLON, G. FRASER, S. HACK, AND A. ZELLER. Automatically generating test cases for specification mining. *Software Engineering, IEEE Transactions on*, 38(2):243–257, 2012.
- [87] K. DEB AND H. JAIN. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints. *IEEE Transactions on Evolutionary Computation*, 18(4):577–601, Aug 2014.
- [88] K. DEB, A. PRATAP, S. AGARWAL, AND T. MEYARIVAN. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182 – 197, 2002.
- [89] A.C. DIAS-NETO, R. SUBRAMANYAN, M. VIEIRA, AND G.H. TRAVASSOS. A survey on model-based testing approaches: a systematic review. In *International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 31–36, 2007.
- [90] M. DONG AND L. ZHONG. Chameleon: A color-adaptive web browser for mobile OLED displays. *IEEE Transactions on Mobile Computing*, 11(5):724–738, May 2012.

- [91] M. DONG AND L. ZHONG. Power modeling and optimization for OLED displays. *IEEE Transaction on Mobile Computing*, 11(9):September, 2012.
- [92] JOSHUA J. DRAKE, ZACH LANIER, COLLIN MULLINER, PAU OLIVA FORA, STEPHEN A. RIDLEY, AND GEORG WICHESKI. *Android Hacker's Handbook*. John Wiley and Sons, 2014.
- [93] J.J. DURILLO, A.J. NEBRO, CARLOS A. COELLO COELLO, J. GARCIA-NIETO, F. LUNA, AND E. ALBA. A study of multiobjective metaheuristics when solving parameter scalable problems. *Evolutionary Computation, IEEE Transactions on*, 14(4):618–635, 2010.
- [94] JUAN J. DURILLO AND ANTONIO J. NEBRO. jMetal: A Java framework for multi-objective optimization. *Advances in Engineering Software*, 42:760–771, 2011.
- [95] SEBASTIAN ELBAUM, GREGG ROTHERMEL, SRIKANTH KARRE, AND MARC FISHER II. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, March 2005.
- [96] J. FLINN AND M. SATYANARAYANAN. Powerscope: A tool for profiling the energy usage of mobile applications. In *WMCSA '99*, pages 1–9, 1999.
- [97] B. FU, J. LIN, L. LI, C. FALOUTSOS, J. HONG, AND N. SADEH. Why people hate your app: Making sense of user feedback in a mobile app store. In *19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1276–1284, 2013.
- [98] BARNEY G. GLASER AND ANSELM L. STRAUSS. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Aldine de Gruyter, New York, NY, 1967.
- [99] DAVID E. GOLDBERG. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1989.

- [100] L. GOMEZ, I. NEAMTIU, T. AZIM, AND T. MILLSTEIN. Reran: Timing- and touch-sensitive record and replay for android. In *International Conference on Software Engineering (ICSE'13)*, pages 72–81, 2013.
- [101] I.J. GOOD. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3/4):237–264, 1953.
- [102] GOOGLE. Android debugger bridge. <http://developer.android.com/tools/help/adb.html>.
- [103] GOOGLE. Android storage options. <http://developer.android.com/guide/topics/data/data-storage.html>.
- [104] GOOGLE. Device monitor. <http://developer.android.com/tools/help/monitor.html>.
- [105] GOOGLE. Espresso pickeractions. <http://developer.android.com/reference/android/support/test/espresso/contrib/PickerActions.html>.
- [106] GOOGLE. Google mobile testing blog. <http://googletesting.blogspot.com/search/label/Mobile>.
- [107] GOOGLE. Nexus 7. <http://www.google.com/nexus/7/>.
- [108] GOOGLE. Notifications. <http://developer.android.com/guide/topics/ui/notifiers/notifications.html>.
- [109] GOOGLE. Profiling with traceview and dmtracedump. <http://developer.android.com/tools/debugging/debugging-tracing.html>.
- [110] GOOGLE. Storage options. <http://developer.android.com/guide/topics/data/data-storage.html>.
- [111] GOOGLE. Systrace. <http://developer.android.com/tools/help/systrace.html>.

- [112] GOOGLE. Testing ui for a single app. <http://developer.android.com/training/testing/ui-testing/espresso-testing.html>.
- [113] GOOGLE. Ui/application exerciser monkey. <http://developer.android.com/tools/help/monkey.html>.
- [114] GOOGLE. Using ddms. <http://developer.android.com/tools/debugging/ddms.html>.
- [115] ALESSANDRA GORLA, ILARIA TAVECCHIA, FLORIAN GROSS, AND ANDREAS ZELLER. Checking app behavior against app descriptions. In *ICSE'14*, pages 1025–1035, 2014.
- [116] M. GRECHANIK, QING XIE, AND CHEN FU. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 9–18, Sept 2009.
- [117] ROBERT J. GRISSOM AND JOHN J. KIM. *Effect sizes for research: A broad practical approach*. Lawrence Earlbaum Associates, 2nd edition edition, 2005.
- [118] JIAPING GUI, STUART MCILROY, MEIYAPPAN NAGAPPAN, AND WILLIAM G. J. HALFOND. Truth in advertising: The hidden cost of mobile ads for software developers. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE '15*, pages 100–110, 2015.
- [119] CHAORONG GUO, JIAN ZHANG, JUN YAN, ZHIQIANG ZHANG, AND YANLI ZHANG. Characterizing and detecting resource leaks in android applications. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 389–398, 2013.
- [120] E. GUZMAN AND W. MAALEJ. How do users like this feature? a fine grained sentiment analysis of app reviews. In *RE'14*, pages 153–162, 2014.

- [121] D. HAN, C. ZHANG, X. FAN, A. HINDLE, K. WONG, AND W. STROUILA. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *19th Working Conference on Reverse Engineering*, pages 83–92, 2012.
- [122] S. HAO, D. LI, W. G. J. HALFOND, AND R. GOVINDAN. Estimating Android applications’ CPU energy usage via Bytecode profiling. In *GREENS’12*, pages 1–7, 2012.
- [123] S. HAO, D. LI, W. G. J. HALFOND, AND R. GOVINDAN. Estimating mobile application energy consumption using program analysis. In *ICSE’13*, pages 92–101, 2013.
- [124] S. HAO, B. LIU, S. NATH, W.G.J. HALFOND, AND R. GOVINDAN. Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *12th annual international conference on Mobile systems, applications, and services (MobiSys’14)*, pages 204–217, 2014.
- [125] MARK. HARMAN, YUE. JIA, AND YUANYUAN ZHANG. App store mining and analysis: Msr for app stores. In *9th IEEE Working Conference on Mining Software Repositories (MSR’12)*, pages 108–112, 2012.
- [126] S. HASAN, Z. KING, M. HAFIZ, M. SAYAGH, B. ADAMS, AND A. HINDLE. Energy profiles of java collections classes. In *38th International Conference on Software Engineering (ICSE’16)*, page to appear, 2016.
- [127] A. HINDLE. Green mining: A methodology of relating software change to power consumption. In *MSR’12*, pages 78–87, 2012.
- [128] A. HINDLE, E.T. BARR, Z. SU, M. GABEL, AND P. DEVANBU. On the naturalness of software. In *International Conference on Software Engineering (ICSE’12)*, pages 837–847, 2012.

- [129] ABRAM HINDLE. Green mining: a methodology of relating software change and configuration to power consumption. *Empirical Software Engineering*, 20(2):374–409, 2015.
- [130] ABRAM HINDLE, ALEX WILSON, KENT RASMUSSEN, E. JED BARLOW, JOSHUA CHARLES CAMPBELL, AND STEPHEN ROMANSKY. Greenminer: A hardware based mining software repositories software energy consumption framework. In *11th Working Conference on Mining Software Repositories (MSR'14)*, pages 12–21, 2014.
- [131] S. HOLM. A simple sequentially rejective Bonferroni test procedure. *Scandinavian Journal on Statistics*, 6:65–70, 1979.
- [132] CUIXIONG HU AND IULIAN NEAMTIU. Automating gui testing for android applications. In *AST'11*, pages 77–83, 2011.
- [133] G. HU, X. YUAN, Y. TANG, AND J. YANG. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Ninth European Conference on Computer Systems (EuroSys'14)*, page Article No.18, 2014.
- [134] C. IACOB AND R. HARRISON. Retrieving and analyzing mobile apps feature requests from online reviews. In *10th Working Conference on Mining Software Repositories (MSR'13)*, pages 41–44, 2013.
- [135] S. IYER, L. LUO, R. MAYO, AND P. RANGANATHAN. Energy-adaptive display system designs for future mobile environments. In *International Conference on Mobile Systems, Applications, and Services (MobiSys'03)*, 2003.
- [136] REYHANEH JABBARVAND, ALIREZA SADEGHI, HAMID BAGHERI, AND SAM MALEK. Energy-aware test-suite minimization for android apps. In *International Symposium on Software Analysis and Testing (ISSTA)*, page to appear, 2016.

- [137] REYHANEH JABBARVAND, ALIREZA SADEGHI, JOSHUA GARCIA, SAM MALEK, AND PAUL AMMANN. Ecodroid: An approach for energy-based ranking of android apps. In *Proceedings of the Fourth International Workshop on Green and Sustainable Software*, GREENS '15, pages 8–14, Piscataway, NJ, USA, 2015. IEEE Press.
- [138] IVAR JACOBSON. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.
- [139] H. JAIN AND K. DEB. An evolutionary many-objective optimization algorithm using reference-point based nondominated sorting approach, part ii: Handling constraints and extending to an adaptive approach. *Evolutionary Computation, IEEE Transactions on*, 18(4):602–622, Aug 2014.
- [140] FREDERICK JELINEK AND ROBERT L. MERCER. Interpolated estimation of Markov source parameters from sparse data. In *In Proceedings of the Workshop on Pattern Recognition in Practice*, pages 381–397, Amsterdam, The Netherlands: North-Holland, May 1980.
- [141] C. S. JENSEN, M. R. PRASAD, AND A. MOLLER. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis (ISSTA'13)*, pages 67–77, 2013.
- [142] N. JONES. Seven best practices for optimizing mobile testing efforts. Technical Report G00248240, Gartner, 2013.
- [143] M. ERFANI JOORABCHI, A. MESBAH, AND P. KRUCHTEN. Real challenges in mobile apps. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, pages 15–24, 2013.
- [144] K. KAPETANAKIS AND S. PANAGIOTAKIS. Efficient energy consumption's measurement on Android devices. In *PCI'12*, pages 351–356, 2012.

- [145] SLAVA M. KATZ. Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, pages 400–401, 1987.
- [146] H. KHALID, M. NAGAPPAN, E. SHIHAB, AND A. HASSAN. Prioritizing the devices to test your app on: A case study of android game apps. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.
- [147] HAMMAD KHALID, EMAD SHIHAB, MEIYAPPAN NAGAPPAN, AND AHMED HASSAN. What do mobile app users complain about? a study on free ios apps. *IEEE Software*, 32(3):70–77, 2015.
- [148] J. KNOWLES AND D. CORNE. The pareto archived evolution strategy: a new baseline algorithm for pareto multiobjective optimisation. In *Proceedings of the 1999 Congress on Evolutionary Computation (CEC'99)*, volume 1, page 105 Vol. 1, 1999.
- [149] PAVNEET SINGH KOCHHAR, FERDIAN THUNG, NACHIAPPAN NAGAPPAN, THOMAS ZIMMERMANN, AND DAVID LO. Understanding the test automation culture of app developers. In *ICST'15*, pages 1–10, 2015.
- [150] Y.W. K.WON AND E. TILEVICH. Reducing the energy consumption of mobile applications behind the scenes. In *ICSM'13*, pages 170–179, 2013.
- [151] D. LI, S. HAO, J. GUI, AND W.G.J. HALFOND. An empirical study of the energy consumption of Android applications. In *International Conference on Software Maintenance and Evolution (ICSME'14)*, page to appear, 2014.
- [152] D. LI, S. HAO, W. G. J. HALFOND, AND R. GOVINDAN. Calculating source line level energy information for android applications. In *ISSTA '13*, pages 78–89, 2013.

- [153] D. LI, Y. JIN, C. SAHIN, J. CLAUSE, AND W.G.J. HALFOND. Integrated energy-directed test suite optimization. In *International Symposium on Software Testing and Analysis (ISSTA'14)*, pages 339–350, 2014.
- [154] D. LI, A. H. TRAN, AND W.G.J. HALFOND. Making web applications more energy efficient for OLED smartphones. In *ICSE'14*, pages 573–538, 2014.
- [155] DING LI AND WILLIAM G. J. HALFOND. Optimizing energy of http requests in android applications. In *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, DeMobile 2015*, pages 25–28, 2015.
- [156] DING LI, YINGJUN LYU, JIAPING GUI, AND WILLIAM G.J. HALFOND. Automated energy optimization of http requests for mobile applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, page To appear, 2016.
- [157] DING LI, ANGELICA HUYEN TRAN, AND WILLIAM G. J. HALFOND. Nyx: A display energy optimizer for mobile web apps. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 958–961, 2015.
- [158] YING-DAR LIN, E.T.-H. CHU, SHANG-CHE YU, AND YUAN-CHENG LAI. Improving the accuracy of automated gui testing for embedded systems. *IEEE Software*, 31(1), Jan 2014.
- [159] YING-DAR LIN, JOSE ROJAS, EDWARD CHU, AND YUANG-CHENG LAI. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering*, Preprint, 2014.
- [160] M. LINARES-VÁSQUEZ, G. BAVOTA, C. BERNAL-CÁRDENAS, M. DI PENTA, R. OLIVETO, AND D. POSHYVANYK. Api change and fault proneness: A threat to the success of android apps. In *ESEC/FSE'13*, pages 477–487, 2013.

- [161] M. LINARES-VÁSQUEZ, C. VENDOME, QI LUO, AND D. POSHYVANYK. How developers detect and fix performance bottlenecks in android apps. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 352–361, Sept 2015.
- [162] MARIO LINARES-VÁSQUEZ. Enabling testing of android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15*, pages 763–765, Piscataway, NJ, USA, 2015. IEEE Press.
- [163] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CARLOS BERNAL-CÁRDENAS, MASSIMILIANO DI PENTA, ROCCO OLIVETO, AND DENYS POSHYVANYK. Api change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 477–487, New York, NY, USA, 2013. ACM.
- [164] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CARLOS BERNAL-CÁRDENAS, ROCCO OLIVETO, MASSIMILIANO DI PENTA, AND DENYS POSHYVANYK. Replication package. <http://www.cs.wm.edu/semeru/data/GEMMA/>.
- [165] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CARLOS BERNAL-CÁRDENAS, ROCCO OLIVETO, MASSIMILIANO DI PENTA, AND DENYS POSHYVANYK. Mining energy-greedy api usage patterns in android apps: An empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014*, pages 2–11, New York, NY, USA, 2014. ACM.
- [166] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CARLOS BERNAL-CÁRDENAS, ROCCO OLIVETO, MASSIMILIANO DI PENTA, AND DENYS POSHYVANYK. Mining energy-greedy api usage patterns in android apps: An empirical study. In *MSR'14*, pages 2–11, 2014.
- [167] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CARLOS EDUARDO BERNAL CÁRDENAS, ROCCO OLIVETO, MASSIMILIANO DI PENTA, AND DENYS POSHY-

- VANYK. 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*, ESEC/FSE 2015, pages 143–154, New York, NY, USA, 2015. ACM.
- [168] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, MASSIMILIANO DI PENTA, ROCCO OLIVETO, AND DENYS POSHYVANYK. How do api changes trigger stack overflow discussions? a study on the android sdk. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 83–94, New York, NY, USA, 2014. ACM.
- [169] MARIO LINARES-VÁSQUEZ, CHRISTOPHER VENDOME, QI LUO, AND DENYS POSHYVANYK. How developers detect and fix performance bottlenecks in android apps. In *ICSME'15*, pages 352–361, 2015.
- [170] MARIO LINARES-VÁSQUEZ, MARTIN WHITE, CARLOS BERNAL-CÁRDENAS, KEVIN MORAN, AND DENYS POSHYVANYK. Mining actionable scenarios for android apps using language models - online appendix. <http://www.cs.wm.edu/semeru/data/MSR15-MonkeyLab/>.
- [171] MARIO LINARES-VÁSQUEZ, MARTIN WHITE, CARLOS BERNAL-CÁRDENAS, KEVIN MORAN, AND DENYS POSHYVANYK. Mining android app usages for generating actionable gui-based execution scenarios. In *MSR 15*, page 111/122, 2015.
- [172] Y. LIU, CH. XU, AND S. C. CHEUNG. Where has my battery gone? finding sensor related energy black holes in smartphone applications. In *PerCom'13*, pages 2–10, 2013.
- [173] YEPANG LIU, CHANG XU, S.C. CHEUNG, AND JIAN LU. GreenDroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, Preprint, 2014.

- [174] YEPANG LIU, CHANG XU, AND SHING-CHI CHEUNG. Characterizing and detecting performance bugs for smartphone applications. In *36th International Conference on Software Engineering (ICSE'14)*, pages 1013–1024, 2014.
- [175] A. MACHIRY, R. TAHILIANI, AND M. NAIK. Dynodroid: An input generation system for android apps. In *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pages 224–234, 2013.
- [176] FEDERICO MAGGI, ANDREA VALDI, AND STEFANO ZANERO. Andrototal: A flexible, scalable toolbox and service for testing mobile malware detectors. In *SPSM'13*, pages 49–54, 2013.
- [177] RIYADH MAHMOOD, NAEEM ESFAHANI, THABET KACEM, NARIMAN MIRZAEI, SAM MALEK, AND ANGELOS STAVROU. A whitebox approach for automated security testing of android applications on the cloud. In *AST'12*, pages 22–28, 2012.
- [178] RIYADH MAHMOOD, NARIMAN MIRZAEI, AND SAM MALEK. Evodroid: Segmented evolutionary testing of android apps. In *FSE'14*, page to appear, 2014.
- [179] S. MALEK, N. ESFAHANI, T. KACEM, R. MAHMOOD, N. MIRZAEI, AND A. STAVROU. A framework for automated security testing of android applications on the cloud. In *SERE-C'12*, pages 35–36, 2012.
- [180] SAM MALEK, HAMID BAGHERI, AND ALIREZA SADEGHI. Automated detection and mitigation of inter-application security vulnerabilities in android (invited talk). In *DeMobile'14*, pages 17–18, 2014.
- [181] CHRISTOPHER D. MANNING AND HINRICH SCHÜTZE. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.
- [182] IRENE MANOTAS, CHRISTIAN BIRD, RUI ZHANG, DAVID SHEPHERD, CIERA JAPAN, CAITLIN SADOWSKI, LORI POLLOCK, AND JAMES CLAUSE. An empirical

- study of practitioners' perspectives on green software engineering. In *International Conference on Software Engineering (ICSE)*, page to appear, 2016.
- [183] IRENE LIZETH MANOTAS-GUTIÉRREZ, LORI L. POLLOCK, AND JAMES CLAUSE. SEEDS: a software engineer's energy-optimization decision support framework. In *ICSE'14*, pages 503–514, 2014.
- [184] A. MARCUS AND J. I. MALETIC. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of 25th International Conference on Software Engineering*, pages 125–135, Portland, Oregon, USA, 2003.
- [185] T. McDONNELL, B. RAY, AND MIRYUNG KIM. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79, Sept 2013.
- [186] ZHANSHUAI MENG, YANYAN JIANG, AND CHANG XU. Facilitating reusable and scalable automated testing and analysis for android apps. In *Internetware'15*, 2015.
- [187] N. MIRZAEI, S. MALEK, C. S. PASAREANU, N. ESFAHANI, AND R. MAHMOOD. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, 2012.
- [188] NARIMAN MIRZAEI, HAMID BAGHERI, RIYADH MAHMOOD, AND SAM MALEK. Sig-droid: Automated system input generation for android applications. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 461–471, Nov 2015.
- [189] ISRAEL MOJICA, BRAM ADMS, MEIYAPPAN NAGAPPAN, STEFFEN DIENST, THORSTEN BERGER, AND AHMED HASSAN. A large scale empirical study on software reuse in mobile apps. *IEEE Software Special Issue on Next Generation Mobile Computing*, 2013.

- [190] ISRAEL MOJICA, M. NAGAPPAN, B. ADAMS, T. BERGER, S. DIENST, AND A. HASSAN. On the relationship between the number of Ad Libraries in an Android App and its rating. *IEEE Software*, 2014.
- [191] I.J. MOJICA RUIZ, M. NAGAPPAN, B. ADAMS, T. BERGER, S. DIENST, AND A.E. HASSAN. Impact of ad libraries on ratings of android mobile apps. *Software, IEEE*, 31(6):86–92, Nov 2014.
- [192] I.J. MOJICA RUIZ, M. NAGAPPAN, B. ADAMS, AND A.E. HASSAN. Understanding reuse in the Android market. In *20th IEEE International Conference on Program Comprehension (ICPC'12)*, pages 113–122, 2012.
- [193] MOONSOON-SOLUTIONS. Power monitor. <http://www.msoon.com/LabEquipment/PowerMonitor/>.
- [194] KEVIN MORAN, MARIO LINARES-VÁSQUEZ, CARLOS BERNAL-CÁRDENAS, AND DENYS POSHYVANYK. Auto-completing bug reports for android applications. In *FSE'15*, page to appear, 2015.
- [195] KEVIN MORAN, MARIO LINARES-VÁSQUEZ, CARLOS BERNAL-CÁRDENAS, CHRISTOPHER VENDOME, AND DENYS POSHYVANYK. Automatically discovering, reporting and reproducing android application crashes. In *ICST'16*, page to appear, 2016.
- [196] MORTEN MOSHAGEN AND MEINAL T. THIELSCH. Facets of visual aesthetics. *Human-Computer Studies*, 68:689–709, 2010.
- [197] SIMONE MUTTI, YANICK FRATANONIO, ANTONIO BIANCHI, LUCA INVERNIZZI, JACOPO CORBETTA, DHILUNG KIRAT, CHRISTOPHER KRUEGEL, AND GIOVANNI VIGNA. Baredroid: Large-scale analysis of android apps on real devices. In *AC-SAC'15*, pages 71–80, 2015.

- [198] BAO NGUYEN AND ATIF MEMON. An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces. *IEEE Transactions on Software Engineering*, 99(Preprints), 2014.
- [199] ADRIAN NISTOR AND LENIN RAVINDRANATH. Suncat: Helping developers understand and predict performance problems in smartphone applications. In *ISSTA'14*, pages 282–292, 2014.
- [200] D. PAGANO AND W. MAALEJ. User feedback in the appstore: An empirical study. In *21st IEEE International Requirements Engineering Conference*, pages 125–134, 2013.
- [201] F. PALOMBA, M. LINARES-VASQUEZ, G. BAVOTA, R. OLIVETO, M. DI PENTA, D. POSHYVANYK, AND A. DE LUCIA. User reviews matter! tracking crowdsourced reviews to support evolution of successful apps. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 291–300, Sept 2015.
- [202] FABIO PALOMBA, MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, ROCCO OLIVETO, MASSIMILIANO DI PENTA, DENYS POSHYVANYK, AND ANDREA DE LUCIA. Online appendix of: Crowdsourcing user reviews to support the evolution of mobile apps. Technical report. <http://www.cs.wm.edu/semeru/data/TSE-cristal>.
- [203] SEBASTIANO PANICHELLA, ANDREA DI SORBO, EMITZA GUZMAN, CORRADO AARON VISAGGIO, GERARDO CANFORA, AND HARALD GALL. How can i improve my app? classifying user reviews for software maintenance and evolution. In *31st IEEE International Conference on Software Maintenance and Evolution (IC-SME 2015)*, pages 281–290. IEEE, 2015.
- [204] A. PATHAK, Y. HU, AND M. ZHANG. Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In *Hotnets'11*, page Article No 5, 2011.

- [205] A. PATHAK, Y. HU, AND M. ZHANG. Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In *EuroSys'12*, pages 29–42, 2012.
- [206] A. PATHAK, Y. HU, M. ZHANG, P. BAHL, AND Y. M. WANG. Fine-grained power modeling for smartphones using system call tracing. In *Sixth Conference on Computer Systems (EuroSys'11)*, pages 153–168, 2011.
- [207] A. PATHAK, A. JINDAL, Y. HU, AND S. P. MIDKIFF. What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In *MobiSys'12*, pages 267–280, 2012.
- [208] D. POSHYVANYK. Using information retrieval to support software maintenance tasks. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 453–456, Sept 2009.
- [209] D. POSHYVANYK AND D. MARCUS. Combining formal concept analysis with information retrieval for concept location in source code. In *Proc. of 15th IEEE ICPC*, pages 37–48, Banff, Alberta, Canada, 2007. IEEE CS Press.
- [210] K. RASMUSSEN, A. WILSON, AND A. HINDLE. Green mining: energy consumption of advertisement blocking methods. In *GREENS'14*, pages 38–45, 2014.
- [211] L. RAVINDRANATH, S. NATH, J. PADHYE, AND H. BALAKRISHNAN. Automatic and scalable fault detection for mobile applications. In *12th annual international conference on Mobile systems, applications, and services (MobiSys'14)*, pages 190–203, 2014.
- [212] A. ROUNTEV AND Y. DACONG. Static reference analysis for gui objects in android software. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

- [213] ALIREZA SADEGHI, HAMID BAGHERI, AND SAM MALEK. Analysis of android inter-app security vulnerabilities using covert. In *ICSE'15*, pages 725–728, 2015.
- [214] CAGRI SAHIN, LORI POLLOCK, AND JAMES CLAUSE. How do code refactorings affect energy usage? In *8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, 2014.
- [215] CAGRI SAHIN, LORI POLLOCK, AND JAMES CLAUSE. From benchmarks to real apps: Exploring the energy impacts of performance-directed changes. *Journal of Systems and Software*, 117:307 – 316, 2016.
- [216] CAGRI SAHIN, MIAN WAN, PHILIP TORNUST, RYAN MCKENNA, ZACHARY PEARSON, WILLIAM G. J. HALFOND, AND JAMES CLAUSE. How does code obfuscation impact energy usage? *Journal of Software: Evolution and Process*, pages n/a–n/a, 2016.
- [217] GAURAV SHARMA. *Digital Color Imaging Handbook*. CRC Press, Inc., Boca Raton, FL, USA, 2002.
- [218] TY SMITH. Mastering proguard for building lightweight android code. <http://www.crashlytics.com/blog/mastering-proguard-for-building-lightweight-android-code/>.
- [219] STACKOVERFLOW. How often to call notifydatasetchanged() when changing arrayadapter. <http://stackoverflow.com/questions/15990849/>.
- [220] OLEKSII STAROV AND SERGIY VILKOMIR. Integrated taas platform for mobile development: Architecture solutions. In *AST'13*, pages 1–7, 2013.
- [221] MIKE STROBEL. Procyon. <https://bitbucket.org/mstrobels/procyon>.
- [222] D.M. SYER, B. ADAMS, Y. ZOU, AND A.E. HASSAN. Exploring the development of micro-apps: A case study on the blackberry and Android platforms. In *11th*

- IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'11)*, pages 55–64, 2011.
- [223] SEYYED EHSAN SALAMATI TABA, IMAN KEIVANLOO, YING ZOU, JOANNA NG, AND TINNY NG. An exploratory study on the relation between user interface complexity and the perceived quality. In *International Conference on Web Engineering*, Sven Casteleyn, Gustavo Rossi, and Marco Winckler, editors, volume 8541 of *Lecture Notes in Computer Science*, pages 370–379. Springer International Publishing, 2014.
- [224] T. TAKALA, M. KATARA, AND J. HARTY. Experiences of system-level model-based gui testing of an android application. In *Fourth International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 377–386, 2011.
- [225] N. THIAGARAJAN, G. AGGARWAL, A. NICOARA, D. BONEH, AND J. PAL SINGH. Who killed my battery: Analyzing mobile browser energy consumption. In *WWW'12*, pages 41–50, 2012.
- [226] YUAN TIAN, M. NAGAPPAN, D. LO, AND A.E. HASSAN. What are the characteristics of high-rated apps? a case study on free android applications. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 301–310, Sept 2015.
- [227] P. TONELLA, R. TIELLA, AND C.D. NGUYEN. Interpolated n-grams for model based testing. In *International Conference on Software Engineering (ICSE'14)*, 2014.
- [228] J. W. TUKEY. *Exploratory Data Analysis*. Addison-Wesley, 1977.
- [229] P. VEKRIS, R. JHALA, S. LERNER, AND Y. AGARWAL. Towards verifying Android apps for the absence of no-sleep energy bugs. In *HotPower'12*, 2012.

- [230] A. VETRO, M. MORISIO, AND M. TORCHIANO. An empirical validation of findbugs issues related to defects. In *15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011)*, pages 144–153, 2011.
- [231] ANTONIO VETRO, MARCO TORCHIANO, AND MAURIZIO MORISIO. Assessing the precision of findbugs by mining java projects developed at a university. In *7th IEEE Working Conference on Mining Software Repositories (MSR'10)*, pages 110–113, 2010.
- [232] VISIONMOBILE. Developer economics q1 2014: State of the developer nation. Technical report, 2014.
- [233] W3C. Contrast ratio definition. <http://www.w3.org/WAI/ER/WD-AERT/#color-contrast>.
- [234] STEFAN WAGNER, JAN JÜRJENS, CLAUDIA KOLLER, AND PETER TRISCHBERGER. Comparing bug finding tools with reviews and tests. In *Testing of Communicating Systems*, Ferhat Khendek and Rachida Dssouli, editors, volume 3502 of *Lecture Notes in Computer Science*, pages 40–55. Springer Berlin Heidelberg, 2005.
- [235] MIAN WAN, YUCHEN JIN, DING LI, AND WILLIAM G. J. HALFOND. Detecting display energy hotspots in Android apps. In *ICST'15*, 2015.
- [236] CHENG-YAO WANG, WEI-CHEN CHU, HOU-REN CHEN, CHUN-YEN HSU, AND MIKE Y. CHE. Evertutor: Automatically creating interactive guided tutorials on smartphones by user demonstration. In *SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*, pages 4027–4036, 2014.
- [237] JI WANG, XIAO LIN, AND CHRIS NORTH. Greenvis: Energy-saving color schemes for sequential data visualization on oled displays. Technical Report TR-12-09, Department of Computer Science, Virginia Tech, 2012.

- [238] HSIANG-LIN WEN, CHIA-HUI LIN, TZONG-HAN HSIEH, AND CHENG-ZEN YANG. Pats: A parallel gui testing framework for android applications. In *COMPSAC'15*, volume 2, 2015.
- [239] WEN-CHIEH WU AND SHIH-HAO HUNG. Droiddolphin: A dynamic android malware detection framework using big data and machine learning. In *RACS'14*, 2014.
- [240] Y. XIAO, R. BHAUMIK, Z. YANG, M. SIEKKINEN, P. SAVOLAINEN, AND A. YLA-JASSKI. A system-level model for runtime power estimation on mobile devices. In *International Conference on Green Computing and Communications*, pages 27–34, 2010.
- [241] F. XU, Y. LIU, Q. LI, AND Y. ZHANG. V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics. In *NSDI'13*, pages 43–56, 2013.
- [242] W. YANG, M.R. PRASAD, AND T. XIE. A grey-box approach for automated gui-model generation of mobile applications. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*, pages 250–265, 2013.
- [243] RAZIEH NOKHBEH ZAEEM, MUKUL R. PRASAD, AND SARFRAZ KHURSHID. Automated generation of oracles for testing user-interaction features of mobile apps. In *ISCT'14*.
- [244] JACK ZANG, AYEMI MUSA, AND WEI LE. A comparison of energy bugs for smartphone platforms. In *MOBS'13*, 2013.
- [245] L. ZHANG, M. S. GORDON, R. P. DICK, Z. MORLEY, P. DINDA, AND L. YANG. Adel: An automatic detector of energy leaks for smartphone applications. In *CODES+ISSS'12*, pages 363–372, 2012.

- [246] L. ZHANG, B. TIWANA, Z. QIAN, Z. WANG, R. P. DICK, Z. MORLEY, AND L. YANG. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *CODES+ISSS'10*, pages 105–114, 2010.
- [247] J. ZHENG, L. WILLIAMS, N. NAGAPPAN, W. SNIPES, J.P. HUDEPOHL, AND M.A. VOUK. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, April 2006.
- [248] ECKART ZITZLER, MARCO LAUMANN, AND LOTHAR THIELE. Spea2: Improving the strength pareto evolutionary algorithm. Technical report, 2001.
- [249] ECKART ZITZLER AND LOTHAR THIELE. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Trans. Evolutionary Computation*, 3(4):257–271, 1999.