Empirical Software Engineering (2024) 29:31
https://doi.org/10.1007/s10664-023-10391-y

A large-scale empirical study on mobile performance: energy, run-time and memory

Rui Rua[1,2] · **João Saraiva**[1,2]

Accepted: 6 September 2023 / Published online: 27 December 2023

Abstract

Software performance concerns have been attracting research interest at an increasing rate, especially regarding energy performance in non-wired computing devices. In the context of mobile devices, several research works have been devoted to assessing the performance of software and its underlying code. One important contribution of such research efforts is sets of programming guidelines aiming at identifying efficient and inefficient programming practices, and consequently to steer software developers to write performance-friendly code. Despite recent efforts in this direction, it is still almost unfeasible to obtain universal and up-to-date knowledge regarding software and respective source code performance. Namely regarding energy performance, where there has been growing interest in optimizing software energy consumption due to the power restrictions of such devices. There are still many difficulties reported by the community in measuring performance, namely in large-scale validation and replication. The Android ecosystem is a particular example, where the great fragmentation of the platform, the constant evolution of the hardware, the software platform, the development libraries themselves, and the fact that most of the platform tools are integrated into the IDE's GUI, makes it extremely difficult to perform performance studies based on large sets of data/applications. In this paper, we analyze the execution of a diversified corpus of applications of significant magnitude. We analyze the source-code performance of 1322 versions of 215 different Android applications, dynamically executed with over than 27900 tested scenarios, using state-of-the-art black-box testing frameworks with different combinations of GUI inputs. Our empirical analysis allowed to observe that semantic program changes such as adding functionality and repairing bugfixes are the changes more associated with relevant impact on energy performance. Furthermore, we also demonstrate that several coding practices previously identified as energy-greedy do not replicate such behavior in our execution context and can have distinct impacts across several performance

Communicated by: Bara Buhnova

✉ Rui Rua
rui.a.rua@inesctec.pt

João Saraiva
saraiva@di.uminho.pt

1 HASLab/INESC TEC, Braga, Portugal

2 University of Minho, Braga, Portugal

indicators: runtime, memory and energy consumption. Some of these practices include some performance issues reported by the Android Lint and Android SDK APIs. We also provide evidence that the evaluated performance indicators have little to no correlation with the performance issues' priority detected by Android Lint. Finally, our results allowed us to demonstrate that there are significant differences in terms of performance between the most used libraries suited for implementing common programming tasks, such as HTTP communication, JSON manipulation, image loading/rendering, among others, providing a set of recommendations to select the most efficient library for each performance indicator. Based on the conclusions drawn and in the extension of the developed work, we also synthesized a set of guidelines that can be used by practitioners to replicate energy studies and build more efficient mobile software.

Keywords Performance · Empirical · Mobile · Testing

1 Introduction

In this century, the focus on computer performance is changing: a program's execution time and memory usage are no longer the sole concerns when discussing performance. The fast adoption of non-wired computer devices and the construction of big data centers in recent years is making energy consumption one of the main bottlenecks when building powerful computers (Theis and Wong 2017; Schlachter 2013) and software (Pinto and Castor 2017).

Battery life is known to be one of the major factors influencing the satisfaction of mobile device users (Thorwart and O'Neill 2017). A survey with 1,894 smartphone users in the US placed battery life as the most important factor impacting smartphone purchasing decisions (Richter 2019), with 9 out of 10 users suffering from low battery anxiety (Mickle 2018). Users of mobile devices have also been showing concerns regarding performance, reporting energy-related issues, and avoiding applications (apps for short) that they identify as energy-greedy (Pinto and Castor 2017) in order to increase the device's uptime. Also, a research study (Ma et al. 2013) reported that most of the energy issues detected in mobile devices are caused by apps (more than 47%) rather than by the system itself. Very often, software developers are not aware of possible performance issues (Li et al. 2020) or do not have the knowledge to correct them (Pang et al. 2016; Pinto et al. 2014; Manotas et al. 2016). Furthermore, bad performance results in User eXperience (UX) degradation and consequently less app usage. Such degradation can also indirectly harm developers, especially those who monetize their apps. Thus, mobile software developers have shown a growing interest in optimizing their applications, while also being energy conscious (Pinto and Castor 2017).

As far as mobile platforms are concerned, several studies have emerged, from both academia and industry, with the objective of analyzing different aspects of the energy consumption of software, such as architectural components (Linares-Vásquez et al. 2014; Ortiz et al. 2019; Bangash et al. 2021), programming languages (Couto et al. 2017; Pereira et al. 2017, 2021; Lima et al. 2016), or libraries (Linares-Vásquez et al. 2014; Pathak et al. 2011). In fact, these works allowed developers to evaluate the impact of common programming practices on the energy consumption of the devices. The reported studies, however, have an evident limitation: each study is validated in one specific combination of hardware/software that can limit the generality of the reported results (Vilkomir et al. 2014) and do not consider (at least) several performance indicators. A recent survey (Hort et al. 2022) analyzed 156 publications published between 2008 and 2020 focused on mobile applications' performance

and identified several gaps in the literature that are still unexplored. For instance, this study evidences the need for an evaluation of whether anti-patterns that exist for several performance indicators such as responsiveness, memory or energy usage, exist for other indicators, such as application launch time.

Moreover, each study involving dynamic analysis was conducted over its own (reduced) set of mobile applications, where the median number of apps is around 8 and not more than 100 (Kong et al. 2019). As a consequence, this makes it impossible to properly compare the energy impact of the reported programming practices. Indeed, a large-scale study on the energy consumption of such programming practices, when executed in the same setting, is needed to help both software and compiler developers write/generate energy-efficient code.

In this paper, we present a large-scale study aiming at understanding the real-world energy performance impact of several programming practices reported in the literature. In order to define guidelines to help both software and compiler developers write/generate efficient code, we discuss the following performance indicators: memory, execution time, and energy consumption, with a special emphasis on the latter. In this way, we intend to validate assumptions on the performance impact of several known programming practices, while also presenting new conclusions on practices not yet analyzed until now in terms of all performance components considered in this study. For instance, we evaluate the resultant performance impact of several development/maintenance-related activities performed at source-code level. Despite the fact that these types of changes might not be intrinsically and directly associated with performance, these can affect source code resources and components that have a significant performance impact. Our conclusions suggest that many of the analyzed coding practices previously labeled as inefficient have distinct performance impacts across the considered performance indicators and its classification in terms of performance is not evident. We also concluded that different statically detected project source modifications such as fixing bugs, adding layouts or strings and performing refactors are the changes most associated with relevant changes in app performance.

In order to gather data to synthesize our conclusions, we used model-based GUI testing frameworks in a black box-testing setup in order to evaluate if it is possible to identify relevant changes in an application's energy performance throughout its various releases. After confirming the validity of this approach, and in order to identify the main causes of such performance changes, each change was cataloged and matched with the project source's modifications performed since the last release. Furthermore, we used the data from this analysis to classify and evaluate the performance of statically detected issues by widely-used tools like Android Lint (Google 2021), as well as energetically inefficient programming practices (also known as Red APIs) presented in a previous study (Linares-Vásquez et al. 2014). In addition, we carried out an analysis on similarly competing libraries that perform typical programming tasks, in order to assess which libraries might be most efficient or inefficient for certain programming purposes.

In summary, the results of our study will answer the following questions:

- **RQ1:** Which program changes have more impact on an Android's applications' energy consumption?
- **RQ2:** Do previously identified inefficient programming practices exhibit the same behavior on larger datasets and different contexts?
- **RQ3:** Are there significant performance differences between apps that use different competing libraries implementing typical programming tasks?

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

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

To the best of the authors' knowledge, we present the largest empirical study on mobile software and source-code performance known to date. As far as we acknowledge, there is no equivalent empirical study, in terms of magnitude, that involved real-world app execution built from source in physical Android devices. This study analyzed 1322 versions of 215 different apps, which implies that 1322 different APKs have been installed and executed, through different tests with different inputs, totaling 27900 executed test scenarios. These were gathered after filtering 708 apps across 6071 versions based on certain criteria (Section 3.3), such as if they could be automatically compiled in our workstation with our automatic procedure or executed in the target devices. All analyzed versions were both present in the Google Play Store and have their respective source code available in open-source repositories. The process of gathering and filtering such a number of compilable and executable apps, through dynamic execution, lasted for several weeks, with a total processing time of approximately 28 days just to execute all tests on the final set of filtered apps.

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

- Adding functionalities, fixing bugs, adding layouts or strings and performing refactors are the changes most associated with relevant changes in app performance;
- Many coding practices/patterns previously identified as inefficient did not evidence such behavior in our tests. For instance, a relevant part (40%) of the APIs identified as energy-greedy in Linares et al. (2014) was even more correlated with energy-efficient tests.
- The levels of Severity with which Lint issues are cataloged have little correlation with the correspondent performance impact.
- The evaluation of the performance of apps using different competing libraries that perform similar common programming tasks allowed us to classify the efficiency of each library on each performance indicator. This evaluation suggests that there are libraries that favor one or more performance indicators but end up being the most inefficient in other components. For example, for web communication via HTTP, Volley was the library used in the tests with the best memory and energy efficiency, being the one present in the tests with the worst execution time.

The remainder of the document is as follows: Section 2 introduces several of the most relevant contributions regarding performance in mobile platforms. Section 3 describes the building blocks used to design the processing procedure followed for gathering the collected apps and automatically analyzing their performance. Afterward, we present and discuss the main results from our study in Section 4. In Section 5, we answer the three research questions and present several guidelines for developers building performance-friendly apps

and the resulting dataset. Section 6 presents some of the threats to the validity of our work. Finally, Section 7 presents the conclusions and future research directions that this work can provide.

2 Performance Awareness in Mobile Platforms

This section presents work and results in software performance analysis throughout recent years. As specified in Section 1, the focus is on mobile platforms and run-time, memory, and energy performance, with a greater focus on the latter. Thus, this section presents some of the most important contributions in this area in terms of tools, methodologies, data and knowledge.

The execution of one app depends mainly on 4 major aspects (Pinto and Castor 2017): 1) a given software system under execution; 2) a given hardware combination; 3) a given context; and 4) a given time. In a mobile setup, the number of possible execution scenarios is even larger. According to Android, there are 24 000 different types of devices from nearly 1,300 different brands running Android, and this number continues to grow. Also, Android offers over 30 different platform versions (API levels) for mobile devices and suffers from fragmentation (Park et al. 2013), which is the biggest challenge in testing Android apps according to a recent study (Lin et al. 2020). Context also plays a key role, since the way software is built and used as a critical influence on energy consumption (Ortiz et al. 2019; Pinto and Castor 2017), which also depends on the static and dynamic features of not only the user (personal features), but also the device, and environment (network conditions, temperature, among others) (Pereira et al. 2020). This tremendous number of possible execution scenarios makes it almost impossible to achieve meaningful and universal conclusions, as well as gather representative data regarding the apps and platform development paradigm. A large-scale empirical study performed in multiple devices, platform versions, apps and contexts can be a key factor to better understand the software performance behavior (Pereira et al. 2020).

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

Most studies from the state-of-the-art that analyze Android coding practices only cover JVM source code or bytecode code (Couto et al. 2015; Linares-Vásquez et al. 2014; Cruz et al. 2017; Ribeiro et al. 2021; Couto et al. 2020; Maia et al. 2020), since Java was for many years the reference approach for Android development. With the growing popularity of other programming languages, frameworks and libraries were created for development that allowed not only to develop code using other programming languages, but also to do it cross-platform. Since recent studies show that different languages have different energy footprints (Couto et al. 2017; Pereira et al. 2021), other studies have emerged in the last years aiming to analyze the energy consumption of applications built with different languages or development

frameworks (Biørn-Hansen et al. 2020; Oliveira et al. 2017). These studies showed that although the use of these approaches may lead to decreased performance compared to the native development approach, there are several cases where their use even substantially benefits energy consumption. Peters et al. (2021) also evaluated the run-time performance impact of migrating from Java to Kotlin, by dynamically analyzing 10 open-source apps that conducted such migration. The authors concluded that migrating to Kotlin has a statistically significant impact on CPU and memory usage, while not significantly impacting energy consumption. This study also confirmed that most open-source Android apps migrated to Kotlin.

In terms of tools and techniques to estimate or measure energy consumption, several efforts offer powerful alternatives. For developers to inspect and evaluate performance consumption on individual apps, Android Profiler (Google 2021) offers a set of tools that can be used to evaluate performance in terms of CPU usage, energy, or memory performance. However, this tool is embedded in the Android Studio IDE, aiming to help developers to perform single app analysis with a GUI view. This embedding makes it difficult to port the tools outside this environment and perform bulk app analysis. For this purpose, other tools have emerged that measure, analyze, and compare performance among sets of apps.

Besides hardware-based solutions, such as Monsoon (2021), which are physical external apparatus, require device/battery disassembly, and are usually costly, there are several alternatives (Chowdhury et al. 2019; Hu et al. 2017; Nucci et al. 2017) that also offer accurate results. For instance, GreenScaler (Chowdhury et al. 2019) can be used to estimate energy consumption during app execution, using a model-based approach to estimating energy consumption based on invoked system calls. Its energy model was calibrated using data from previous work by the same authors (Hindle 2013). Having access to the source code, PETrA (Nucci et al. 2017) is an alternative to consider to measure and locate inefficient energy consumption of applications. PETrA approaches this by using APIs available since version 5 of Android.[1] The authors claim that this tool can estimate the energy consumption of an Android application's source code with a low granularity, i.e. method level, providing accurate results (Nucci et al. 2017).

Regarding performance evaluation on mobile devices, several studies and efforts have appeared with a view of assessing the impact of programming practices on both app and operating system performance. From the Android platform, a list of statically detectable source-code performance issues has been compiled. These issues can be automatically detected in an app's source code using Android Lint and were widely studied by the research community (Goaër 2020; Couto et al. 2020; Das et al. 2020; Cruz et al. 2017; Goaër 2020). Android Lint allows the detection of various possible Android issues of different categories, namely performance issues. Although the listed Lint performance issues are accompanied by a categorization of severity, a brief description, and the impact of the issue, their gain in terms of performance is not clear nor in what performance indicators do they affect (run-time, energy consumption, or memory). Many recent studies have assessed the performance impact of software practices at different levels of granularity. From languages (Couto et al. 2017; Pereira et al. 2017, 2021; Lima et al. 2016), development tools or software standards (Pereira et al. 2018; Couto et al. 2015, 2017), to methods and libraries (Pereira et al. 2020, 2016; Linares-Vásquez et al. 2014), several community efforts have proven that it is possible to optimize software at every software development granularity level.

[1] https://developer.android.com/about/versions/android-5.0.html#Power

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

A recent research effort conducted by Das et al. (2020) performed a large-scale study of a large set of Android applications, aiming at analyzing the evolution of statically detectable performance issues. This study analyzed 724 open-source repositories written solely in Java (projects containing NDK or Kotlin code were excluded) of Android apps in order to detect and study the evolution of 9 performance patterns. The goal of the study was to evaluate how much time the energy patterns remained unsolved on these apps, as well as how this kind of patterns tend to appear in the development lifecycle of Android apps. From the analyzed set of apps, 316 had performance issues, while 45% of these didn't solve such issues.

Mazuera-Rozo et al. (2020) conducted a large-scale study that aimed at categorizing performance bugs on mobile platforms. From an analysis of the commit history of 47 Android apps and 31 IOS apps, the authors synthesised a taxonomy of performance bugs by aggregating them into different categories. The authors also analyzed the survivability of these bugs, concluding that on average, performance bugs tend to persist on mobile software for longer than non-performance bugs. Another study (Habchi et al. 2021) that manually analyzed 561 smell-removing commits of 324 apps concluded that the high diffuseness of mobile-specific code smells is not a result of releasing pressure and that app developers do not refactor smelly instances even when they are aware of them.

Several literature studies been also targeted end users alongside developers with their studies focusing on energy consumption. These studies aimed at comparing competing apps for the Android platform and prove that these have different energy footprints and many of them have room for performance improvements. A recent study (Rua et al. 2020) evaluated the performance of several of the most widely used keyboards on the Android platform and concluded that it is possible to save a significant amount of energy just by changing to a more energy-saving keyboard or keyboard configuration. Also, another study (Gonçalves et al. 2022) focused on the most widely used Android browsers also concluded that there are significant differences in terms of energy consumption among these browsers. This study evaluated the efficiency of each browser for different use cases and concluded that Chrome was the most energy-friendly browser for tasks such as playing videos on Vimeo and browsing Facebook.

Linares-Vásquez et al. (2014) empirically measured the energy consumption of method calls in a small set of 55 Android apps to identify energy-greedy APIs (or Red APIs) and usage patterns. This study identified a total of 131 energy-greedy APIs and several guidelines targeting app developers aiming to save energy on their apps, such as avoiding certain software design principles such as Information Hiding. More recently, Li et al. (2020) used 2 different testing frameworks to execute apps in different execution contexts and with different inputs to detect energy-issues in 27 real-world apps. Both Lint issues and these Red APIs will be under

analysis, in terms of their performance impact, with their respective results and conclusions presented in the following sections.

A recent study (Li et al. 2022) proposed a new testing tool to detect energy issues in real-world applications. This study dynamically analyzed the consumption of 36 widely-used applications, using Monsoon (2021) for power monitoring and the Dynodroid testing tool to exercise applications. This study concluded that 62% of the energy issues detected need to be executed with specific inputs or in special contexts to be detected and only less than 20% can be manifested with simple inputs. Finally, they also concluded that almost 90% of the detected issues in their experiment were previously unknown to developers and that energy issues are generally harder to fix than non-energy related issues. 2 different studies also tried to assess the impact of logging on Android applications (Chowdhury et al. 2018; Zeng et al. 2019), having concluded that performing intensive logging has a significant impact on application performance. Chowdhury et al. (2018) also conclude that logging at a limited rate (less than 1 message per second) has no significant impact on performance and that factors such as logging rate, disk flush and message size have a significant impact on energy consumption.

Efforts by the research community have also shown that it is possible to improve the energy performance of applications without making changes to their original source code. Bangash et al. (2022) proposed a technique based on the Redex tool,[2] which aims to change the bytecode of applications in order to improve run-time performance and reduce application size. A preliminary evaluation of this tool carried out over 22 applications allowed to improve the performance of 12 applications. However, this experiment allowed the authors to conclude that the proposed optimizations do not present universal results, since they did not always lead to consumption improvements in some of the analyzed applications.

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

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

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

3 Methodology

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

3.1 Performance Analysis Tooling

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

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

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

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

The former (Monkey for short) has been used in many previous works (Chowdhury et al. 2019; Rua, Couto, and Saraiva 2019; Hu et al. 2017) to detect energy/performance-related issues, being the most widely used tool by the researching community to automatically test applications (Choudhary et al. 2015). It allows the generation of pseudo-random streams of

user events such as clicks, touches, or gestures, as well as several system-level events. It also allows repeating tests and the respective sequence of events, which is essential to replicate the same work over different apps to draw comparisons among them. The latter, App Crawler, is a recent tool provided by Google that allows developers to crawl app GUIs. This tool considers contextual behavior, being able to efficiently locate app UI widgets and properly interact with them.

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

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

3.2 Testing Workflow

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

3.2.1 Instrumentation and Building

For the source code instrumentation task of the collected apps of our corpus, we used the JInst tool (Couto et al. 2015; Rua, Couto, and Saraiva 2019). Source code written in Java and Kotlin is parsed into an AST (Abstract Syntax Tree), which is then traversed to locate tree nodes corresponding to method calls. At method call nodes we insert the code fragment (subtree) needed to perform energy monitoring at run-time. Because in this study we rely on Trepn to monitor energy consumption, the inserted code consists of calls to TrepnLib (Rua, Couto, and Saraiva 2019; Rua, Couto, Pinto et al. 2019). A dependency to TrepnLib is added to the building scripts which are also automatically adapted to the context of our running platform. JInst performs method-level instrumentation and uses both JavaParser[3] and Kastree[4] to instrument Java and Kotlin code, respectively.

As a final step of this phase, the AnaDroid tool compiles and builds the apps's APKs. To guarantee the correct build of the instrumented applications (both their source code and

[3] Java Parser: https://javaparser.org

[4] Kastree: https://github.com/cretz/kastree

Table 1 Auxiliary tools used for Android projects' instrumentation and compilation

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

scripts), we follow a regression test-like approach that detects and corrects a set of typical building errors such as API/dependencies conflicts (Scalabrino et al. 2020), Gradle and Gradle-plugin version mismatching, etc. Finally, we used *Androguard*[5] for detecting the APIs present in the APK built. The set of tools used so far from the application gathering to the compilation stage is presented in Table 1.

Figure 1 shows the flow of work performed in every app of our corpus.

3.2.2 Execution and Analysis

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

An app, however, may require the use of sensors to be fully functional and allow to fully explore its core functionalities. In order to minimize external interference to the monitoring process while not limiting apps' execution, we designed a simple heuristic that enables certain sensors according to the permissions specified by each app. Furthermore, for each app, we inspect the *Android Manifest* file to infer the permissions required by the app and we enable sensors whose usage is frequently associated with some permissions before the test execution. The list of permissions considered to activate sensors and the respective sensors are detailed in Table 2.

Therefore, we guaranteed that at the beginning of each app's execution, the required sensors are turned on, according to the permissions specified by the the app developers in the manifest file. Controlling the status of these sensors is programmatically done via *adb*. This makes our process permission-aware (Sadeghi et al. 2017), while reducing testing effort and increasing code coverage.

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

[5] Androguard: https://github.com/androguard/androguard

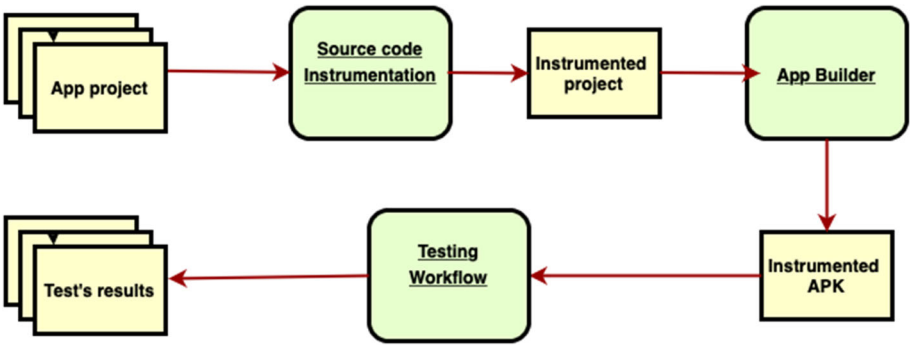

Fig. 1 Main workflow

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

In the *initProfilerService* step, the device cache is cleared, so that no data from previous executions may cause biased results. Afterward, the Trepn service is started, and the app is initialized. The device resource status is collected (number of running processes, CPU and RAM usage, etc) just before starting the monitoring process ('*startProfiler ()*') and the system log is cleaned. This step consists of using Trepn states to temporarily delimit the beginning of the process. The next step consists of starting the testing framework and executing the tests. Finally, the monitoring process is finished and the device status is recorded again. The app is closed, the collected monitoring data is exported to a *csv* file, the system log is extracted, the profiler is turned off, and the data is exported to the workstation.

3.3 Data Collection and Validation

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

In order to obtain our final corpus of 215 apps and respective 1322 versions which were analyzed in our empirical study, we resorted to using open-source software repositories.

Table 2 List of permissions that activate sensors

Permission	Sensor enabled
ACCESS_FINE_LOCATION	GPS
ACCESS_COARSE_LOCATION	GPS
BLUETOOTH	Bluetooth
BLUETOOTH_*	Bluetooth
INTERNET	Wi-fi
NFC	NFC
NFC_TRANSACTION_EVENT	NFC

Algorithm 1 Workflow algorithm.

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

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

One of the main objectives of this work was to focus on quality apps, avoiding projects which were deprecated/sample, or with minimal and unfinished functionalities. Thus, we only selected apps that were present in the Google Play Store. This assured us a minimum level of peer approval, as to be present in the Google Play Store an app must be approved and meet certain quality and security standards. Although the application admission process is not as rigorous as the Apple App Store process, all applications on the Google Play Store undergo an analysis process that assesses their correctness, security, among others.[6] Furthermore, as we also wanted to track the evolution of app source code and have a minimum number of app versions/releases to draw comparisons from, we selected those that had at least 3 versions also present in the repositories.

After extracting the source code of each existing version, it was necessary to guarantee that each app was compilable in our development environment and executable on the mobile devices we had at our disposal. Often times developers customize their development environment, some of the available software in open-source repositories only compile on the machines they were originally developed. Since manually adapting the build scripts for each app/version is impractical, to filter the apps that did not compile/execute on our devices, we followed the automatic procedure described in Section 3.2. Thus, we used AnaDroid (Rua, Couto, and Saraiva 2019) tool to automate the building process and to execute a simple test: building the APK without any transformation performed, installing the app on the device,

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

and testing whether it launched without errors on the device. After executing this automated procedure, we obtained a set of 565 executable apps containing 2629 versions in total.

Our dynamic execution pipeline, shown in Fig. 1, involves an elaborated application analysis process, whose complexity influences the accuracy of the automatic execution process. The execution of the building process outside the typical application development machine/environment can cause failures in the automatic building process. The application instrumentation process changes source code and building scripts, which can cause problems during project building. Including instrumentation libraries can cause incompatibilities between libraries and SDK versions, among other errors. After replicating the previous process of building and installing the APK, but now with the instrumentation process, the percentage of applications fully capable of being processed by our pipeline drops to 270. This reveals that AnaDroid's automated application instrumentation process has a success rate of 47.8%.

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

The execution of the execution pipeline and respective tests allowed us to gather millions of measurements of a large and diversified set of metrics (discussed later in Section 5.3), obtained through both static and dynamic analysis of the application code and execution of the respective executable/APK. In order to draw fair comparisons among the collected apps, versions and respective tests, for the analysis present in this section we only considered tests performed with Monkey, since the number of events and test duration obtained using App Crawler strongly depends on the app UI complexity, being considerably divergent among apps. Although the tests performed with Crawler are not analyzed in this section and considered to answer the research questions we intend to answer, their inclusion in this study and the corresponding dataset is still relevant, as we will illustrate in Section 5.

Each testing framework was invoked via command-line, with a defined set of arguments for each executed application. For Monkey, the seeds used to generate the pseudo-random tests can be found in the online appendix. For each test, 1000 events were generated with a delay of 100 ms between them, avoiding system keys that generate events that can interfere with the test process, invoking functions outside the application (volume buttons, return, home, etc). Also, these commands were executed with the –ignore-security-exceptions switch, to avoid stopping the tests when run-time permissions are asked by the apps under test. Executions of both frameworks were limited by a timeout of 5 minutes (300 seconds), in order to prevent errors that occurred during execution from blocking the execution process. After the defined time, if an error occurred during the execution, the test was discarded. Next, we present both commands:

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

```
$ timeout -s 9 300 <path_to_jar>/crawl_launcher.jar --apk-file
<path_to_installed_APK> --app-package-name <package>
```

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

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

The distribution of the number of apps/releases per year presented in Fig. 2 aims to illustrate the diversity of our dataset in terms of the analyzed versions' age, in order to classify the novelty of our dataset and respective applications. According to this analysis, the most common release years are 2018 and 2019, with 70 and 46 applications, respectively. Our dataset has also at least 20 and 40 app versions from 2020 and 2017, having only 1 version from 2015 and 17 from 2017. This data demonstrate the diversity of our dataset and that our dataset is not composed of legacy or deprecated software.

Figure 3 presents an histogram that illustrates the distribution of the number of versions per app. As can be observed, the most common number of versions per app is 3, with 35 apps having such number of versions. Nearly 85% of the apps executed have 10 or less versions executed and 60.9% of the applications have 5 or less versions executed. This data demonstrates that our global results are relative to a diverse set of apps, which helps to

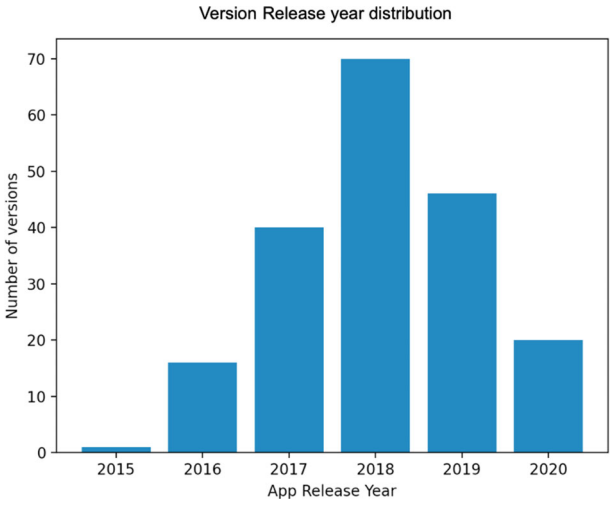

Fig. 2 Distribution of number of app/versions per release year

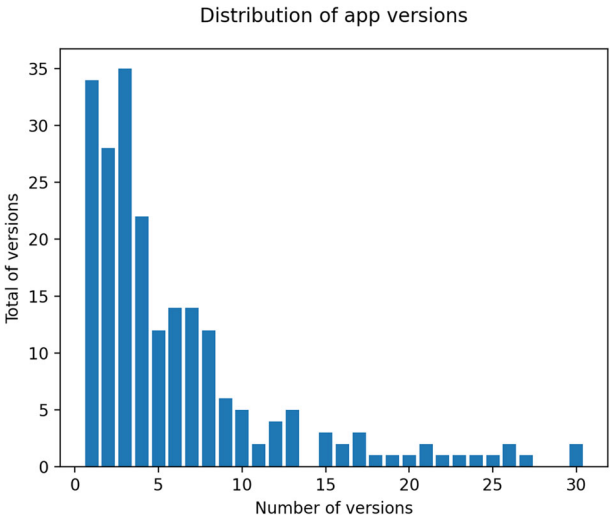

Fig. 3 Distribution of number of versions per app

mitigate the possibility of being influenced by specific apps with a high number of executed versions, since there are very few of such cases.

In order to evaluate if our final set of applications is representative of the set of applications present in the Play Store, we used Yamane sample size method (Yamane 1973). As a reference value, we considered the total number of apps present in the App Store in December of 2021 (the time of writing of this document), 2605000 apps. If we consider this value as the total number of our population, the Yamane method tells us that we would need a sample of approximately 400 applications to obtain a margin of error of 5%, at a confidence level of 95%. Our set of 215 different applications gives a margin of error of approximately 6.68%. However, if we consider the number of versions as distinct applications, assuming that each application corresponds to a unique software artifact and distinct from the rest of the population, according to Yamane's method our sample would already be representative (see Table 3).

Nonetheless, this (still) large set of apps requires an execution time of more than 28 days to just execute every app with the considered Monkey tests. This excludes the time of building and the installation time, setup, warm-up and cool-down times, exporting results, among others inherent to the proper execution of the automatic procedure described in Section 3.2.

Table 3 Analyzed app categories

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

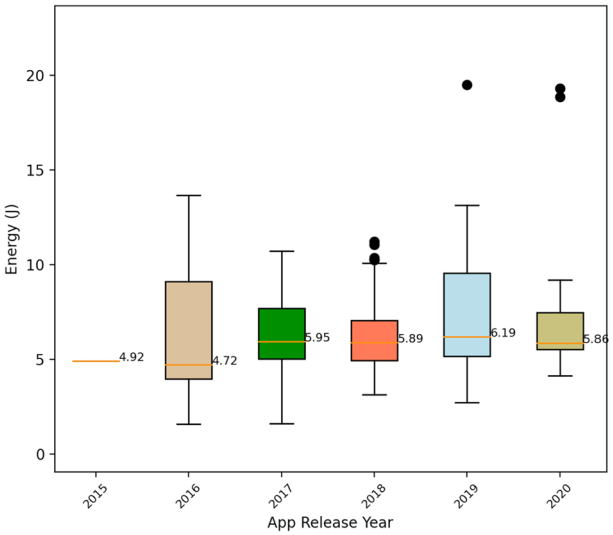

Fig. 4 Average Energy consumption across apps versions' release years

4 Large-Scale Android Performance Analysis

This section presents the results and analysis of the execution pipeline presented in Section 3.2. We performed an empirical study using real devices, in which 2 testing frameworks were executed on a large set of applications. We present the results of executing a set of 215 different apps from the original set of 708, whose filtering process was described in Section 3.3.

Figures 4 and 5 illustrate the distribution of the average test values for the evaluated performance indicators across the app versions' creation year. The date presented in these Figures allows to observe that there is no clear trend in the energy consumption of the executed apps. When using as reference the median values of the boxplots, as well as the lower/upper quartiles and whisker values, our results suggest that the release year does not hint that the performance has been changing throughout the years. Although there is a noticeable increase

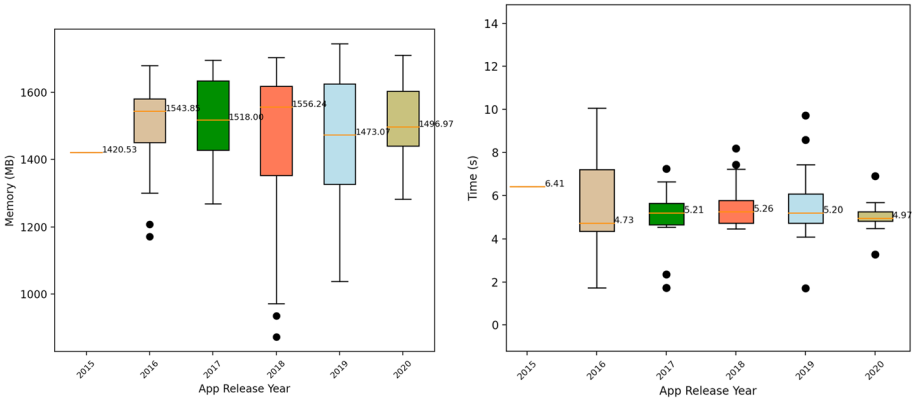

Fig. 5 Memory and run-time across apps versions' release years

since 2017 when comparing the medians, the values from the quartiles and whiskers values do not follow the same trend. In terms of memory and run-time, we also observed the same behavior.

As shown in Fig. 16 there is a clear decrease in energy consumption when using the *Kotlin coroutines* mechanism. *GreenRobot* also shows gains when compared to both *Executor* and *RxJava*. The most inefficient energy consumption to come from *RxJava*. *Executor* has a substantial gain over *RxJava*, obtaining similar results to *GreenRobot*.

After filtering all the invalid data, they were grouped in order to answer the research questions presented in Section 1. The next sections present the results and analyze the data gathered in order to answer the research questions, which answer is later presented in Section 5.

4.1 Performance Behavior Results

In this section, we present the results and analysis performed to answer **RQ1:** *Which program changes have more impact on an Android's applications energy consumption?*. In order to detect and categorize significant changes in the energy behavior of the applications as required to answer this question, we have executed Monkey tests across the different versions of the applications. To select applications where the impact on energy may be significant we defined the following set of criteria:

- **1)** Tests executed with Monkey framework. Tests performed with this tool allow the establishment of fairer comparisons between versions since it ran the same set of tests and events for each version and app.
- **2)** Apps with average energy consumption above 5.2 Joules. We considered apps with consumption above the lower percentile of app average consumption;
- **3)** Apps with average method coverage above 5%. In order to ensure that the tests exercised a minimum amount of code from the application.
- **4)** Apps with a variation of at least 20% between consecutive versions concerning the application's maximum consumption. For this purpose, the versions were sorted according to their version number according to the semantic versioning format.[7] This criterion helps to ensure that a certain version has an abnormal performance behavior compared to the adjacent versions and compared to the expected app behavior;
- **5)** Versions with more than 20 app method invocations, in order to avoid considering simple demo apps or tests with little interaction that are not representative of real-world apps or real user interaction.

These criteria were not chosen in order to reduce the domain of analysis but to ensure that only comparisons are made in versions of applications whose consumption is relevant and in which there was a guarantee that the tests mostly capture the performance of the applications and not of the device's system. Since the performance measurements obtained are captured at the system level, in order to impute the measurements with app execution we imposed the criteria to assure that the executions that were considered in our analysis are relevant. Thus, we consider only executions with relevant consumption (2) whose metrics were less susceptible to being affected by system events, that executed a relevant portion of the application (3,5). Furthermore, we also ensure that we only draw comparisons between versions with very significant divergence in terms of consumption and with irregular behavior as expected in the application itself(4).

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

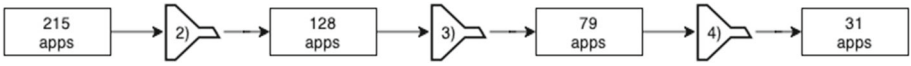

Fig. 6 Filtering process for answering **RQ1**

By considering these criteria, we obtain a set of 31 applications and 46 versions where we detect suspicious energy variations. In fact, we detect such suspicious behavior in 14.2% of the executed applications. When we exclude the applications with minimal energy and coverage, which is observed in 52 apps (24.9%), then we observe that 1 in 4 applications has one of its versions with an abnormal energy performance (when compared to the other versions). Figure 6 illustrates the filtering process performed by the previous sequence. Filters **1)** and **5)** from the previous criteria are not illustrated, as they did not remove any application from the set to which they were applied.

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

To perform this manual inspection we analyzed the information available in the open-source repositories of those applications. We discarded two app versions since they are not available on GitHub anymore, leaving us with a final set of 44 transitions between versions to analyze. All other applications were analyzed via the GitHub website GUI that reports the changes made between project commits and releases. Thus, we classified every change occurring in the version with anomalous consumption in relation to the predecessor version. We structured the changes between versions in four types, namely *performance*, *semantic changes*, *project code/resources*, and *project building*, as presented in Table 4.

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

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

Project Code/Resources We considered changes to *Activities*, *Fragments* and *Permissions*. The first two are the most used classes to build the various "pages"/"screens" that each application can assume during its execution. Thus, a change in activities/fragments may impact the app's performance. Besides changes at the source code level, we also considered changes that affect view components at resource levels, namely layouts/menus and drawables and mipmaps, colors, values and strings. In terms of data to be consumed by the application, we also looked for changes in .db, JSON or text files. Furthermore, we considered the permissions requested by the new versions, so that we could analyze the impact on app performance of these changes. Finally, in a more macroscopic view, we also looked for the number of files changed and commits performed between versions, to correlate such changes with performance measurements. It is important to notice that for these kinds of changes we

Table 4 Criteria considered for the manual analysis

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

analyzed when the project suffered relevant changes, namely, by adding or removing files or by updating existing ones. We did not consider minor changes such as renaming elements, changes in indentation, or fixing typos in strings.

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

Having manually extracted this information for every application, we wish now to understand which criteria are affecting the performance of the mobile apps. Because we need to compare two data samples with dichotomous/categorical values, the comparison is presented in tables containing the proportion of agreement between criteria as a contingency table. The tables show the total number and the percentage of times that a criterion simultaneously occurred when a significant change in the application's performance is detected. In these

Table 5 Relation between raises and drops of the 3 performance indicators

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

tables, a bolded value in raise columns,[8] signifies that the performance value raised in at least 50% of the app versions (implying a potential degradation of performance), while a bold value in drop columns (TODO) that the performance value was improved in at least 50% (implying a potential performance improvement) of the analyzed app versions.

In Table 5 we present a comparison between the up/down pairs of performance indicators detected in our 44 applications. Each cell in the table shows the total and percentage of times that a rise or fall in the value of a performance indicator occurred simultaneously with a rise/fall of another performance indicator.

As we can see in Table 5 the greatest relationship between changes occurs between energy and time, either between rises in consumption of both components (79.16%) and between reductions (75% and 100%). This correlation between time and energy was already observed in several studies (Pereira et al. 2021, 2017; Chowdhury et al. 2019). Although these two performance indicators tend to be highly correlated, we observe that a rise or drop in one of the components does not mean a change of the same magnitude in the other. It is also possible to see that it is much more common to observe a decrease/increase of all performance indicators at the same time, rather than in a disjoint way. However, a rise in one performance indicator does not always lead to the rise of another component. In fact, they are not perfectly correlated. Regarding energy consumption, its rise was only accompanied by a decrease in run-time or memory in 20.84% and 37.5% of the time, respectively.

Table 6 compares the occurrence of the Project Resources, Project Source and Building Code type criteria with the occurrence of energy rises/drops. The percentage (%) reported for each change leading to a rise and drop is relative to the total of energy rise and drops presented in Table 5 (24 and 20, respectively).

As Table 6 shows the criterion that has the strongest impact on the rise of energy consumption is the addition of functionalities. Indeed, in 70.8% of the apps' versions where new functionality was added the energy consumption increased (at least 20%), while in only 30% of them the consumption decreased. On the positive side, the fixing of bugs shows a

[8] We assume the reader has access to an electronic or colored version of this document.

Table 6 Impact of changes in energy consumption

Change	Energy raise		Energy drop	
	Total	%	Total	%
Add functionalities	**17**	**70,8**	6	30,0
Add layout	**14**	**58,3**	5	25,0
Add strings	**3**	**54,2**	8	40,0
Refactor	**13**	**54,2**	*10*	*50,0*
Gradle update	**12**	*50,0*	5	25,0
Update dependencies	8	33,3	6	30,0
BugFix	7	29,2	*12*	*60,0*
New dependencies	7	29,2	7	35,0
Add colors/values	7	29,2	7	35,0
Add drawables	7	29,2	7	35,0
SDK update	6	25,0	4	20,0
New activities	4	16,6	3	15,0
Add mipmap	3	12,5	4	20,0
Update permissions	2	8,3	5	25,0

reduction of energy consumption in 60% of the applications, and only in 29.2% the energy raised.

To better understand why the addition of *functionalities* and *bugfixes* have a strong impact on energy consumption, let us analyze in detail which other changes occur simultaneously with these two criteria. We start by analyzing the impact on the energy, time, and memory consumption in all apps where new functionality has been added. This is shown in Table 7.

As we can see in Table 7 the changes that more often occur when adding new functionality are *Layout* and *String*, *Refactor* and *Update to Gradle Dependencies*. *Layout* does contribute to the rise in energy in 66.7% of the apps. This change also shows an impact on run-time and memory consumption. Such impact, however, is in opposite directions since it is (almost) equally divided between time/memory rises and drops.

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

Let us analyze now the occurrence of *bugfixing* changes as shown in Table 8.

There are a few criteria that occurred concurrently with a significant performance impact. This can be justified by the fact that most *bugfixes* are quick changes that do not involve major changes to Android code and building resources. All occurrences with a significant impact had a negative impact on performance, with the Layout changes having a negative impact on all performance indicators and the addition of functionalities with a negative impact only on energy consumption. These changes were already the changes with the most negative impact on energy consumption, as presented in Table 6.

Table 7 Impact of changes in performance on versions with increased functionality

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

Table 8 Impact of changes in performance on versions with *bugfixes*

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

4.2 Android Programming Practices

This section presents the procedure and results obtained to answer **RQ2:** *Do the previously identified inefficient programming practices behave the same with larger data sets and different contexts?*. To answer this question, we analyzed 2 programming practices widely studied as energy inefficient in the literature: The energy greedy API (Linares-Vásquez et al. 2014) and then Android performance problems Lint (Google 2021). An analysis of these practices is presented in Sections 4.2.1 and 4.2.2, respectively.

4.2.1 Energy Greedy APIs

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

In this section, we analyze in detail Linares et al. energy-greedy APIs, the so-called red APIs. Although the original study reported a set of 129 APIs, in our repository we detected the use of 60 different red APIs only. The APIs that each method invokes are collected with *Androguard* from the Dalvik bytecode contained in the apps' APK. In order to determine if a certain test invoked a certain red API, we verify if each app method invoked during a test contains the red API in its code. Table 9 summarizes the 60 red APIs found in the analyzed

Table 9 Red APIs invoked in Monkey tests

Red API	#Apps	#Versions	#Tests	#Methods
Global	161	852	27674	35802
android.widget.TextView.setText	117	589	4239	15414
android.widget.Toast.show	74	250	1996	3414
android.widget.Toast.makeText	72	262	2071	3492
android.util.Log.e	54	260	2455	14404
android.database.sqlite.SQLiteDatabase.execSQL	27	123	846	1973
android.util.Log.i	24	85	471	1804
android.widget.ImageView.setImageResource	22	93	524	1878
android.content.Intent.getIntExtra	20	104	375	767
android.database.sqlite.SQLiteDatabase.rawQuery	19	76	442	1226
android.database.sqlite.SQLiteDatabase.query	18	68	388	447
android.view.ViewGroup.addView	17	62	416	424
android.app.Activity.findViewById	14	69	419	734
android.graphics.Bitmap.createBitmap	14	85	277	309
android.database.sqlite.SQLiteDatabase.insert	10	47	503	845
android.view.View.setEnabled	10	26	216	426
android.content.res.Resources.openRawResource	9	50	323	326
android.graphics.Canvas.drawText	7	51	470	974
android.widget.EditText.setSelection	7	15	53	53

Table 9 continued

Red API	#Apps	#Versions	#Tests	#Methods
android.webkit.WebView.loadUrl	7	23	68	87
android.view.View.startAnimation	6	20	191	409
android.database.sqlite.SQLiteDatabase.delete	6	39	306	462
android.view.View.setLayoutParams	6	14	209	1080
android.content.ContentResolver.query	6	20	231	475
android.database.sqlite.SQLiteQueryBuilder.query	6	20	231	251
android.widget.ArrayAdapter.notifyDataSetChanged	5	11	30	30
android.webkit.WebView.loadData	4	8	0	0
android.widget.ProgressBar.setMax	4	35	182	211
android.database.sqlite.SQLiteDatabase.update	4	27	195	196
android.webkit.WebView.loadDataWithBaseURL	4	8	57	57
android.app.NotificationManager.cancelAll	3	10	24	24
android.view.ViewGroup.removeAllViews	3	8	7	7
android.app.Dialog.show	3	6	26	26
android.database.sqlite.SQLiteDatabase.openDatabase	3	4	3	3
android.app.Activity.finish	3	18	271	481
android.util.Log.getStackTraceString	3	15	201	358
android.app.Dialog.dismiss	3	8	4	4
android.view.View.setClickable	2	4	1	4
android.database.sqlite.SQLiteDatabase.insertOrThrow	2	2	26	26
android.text.format.DateFormat.getDateFormat	2	3	1	1
android.text.format.DateFormat.getTimeFormat	2	11	112	114
android.database.sqlite.SQLiteDatabase.endTransaction	2	15	133	236
android.widget.ArrayAdapter.clear	2	8	150	150
android.app.Service.onStartCommand	2	22	214	214
android.database.sqlite.SQLiteOpenHelper.getWritableDatabase	1	8	0	0
android.app.Activity.startActivityForResult	1	12	207	216
android.view.Window.findViewById	1	4	7	14
android.location.LocationManager.getGpsStatus	1	2	2	2
android.view.View.performClick	1	2	23	23
android.database.sqlite.SQLiteDatabase.openOrCreateDatabase	1	1	1	1
android.os.Handler.dispatchMessage	1	12	12	12
android.graphics.Bitmap.getPixel	1	6	71	71
android.graphics.BitmapFactory.decodeStream	1	2	2	2
android.content.ContentResolver.update	1	2	20	23
android.app.Activity.setContentView	1	4	4	4
android.database.sqlite.SQLiteStatement.executeInsert	1	2	50	50
android.database.sqlite.SQLiteOpenHelper.getReadableDatabase	1	8	0	0
android.database.sqlite.SQLiteDatabase.getVersion	1	1	1	1
android.telephony.TelephonyManager.getPhoneType	1	1	1	1
android.graphics.Paint.getTextBounds	1	4	4	5
android.view.ViewConfiguration.getLongPressTimeout	1	8	69	69

apps. For each API we associate the number of unique apps, versions, tests and methods where it is called.

Next, we present results regarding energy, memory, and run-time associated with the invocation of several red APIs. Our results include the APIs that were executed by more than 100 test cases, across at least 25 different versions of 10 or more different apps. Figure 7 shows a comparison in terms of energy consumption (in Joules, Y axis) of such red APIs. As we can immediately see, several APIs appear to have a strong negative impact on energy consumption, such as *findViewById* and *addView*.

In order to assess the impact of directly invoking these red APIs on app energy consumption, we present in Fig. 8 the energy consumption of the tests (of the same apps) that did not exercise those APIs. As we can see, the most notable energy footprint is shown by the same *findViewById* and *addView* APIs. By looking at the medians, the value of *findViewById* for example more than doubles most of the medians of all APIs illustrated in the figure, except for *addView* and *Log.e*. Comparing *findViewById* with the API with lower consumption, *drawText*, its consumption in terms of median is 2.58 times higher.

If we look in more detail at Figs. 7 and 8, we see that there are red APIs whose impact on energy consumption is negligible, or more surprisingly the energy consumption decreases. For instance, just by looking at the medians, tests now not containing *drawText*, *createBitmap* and *rawQuery* have increased their energy consumption. More particularly, tests not containing *drawText* increased their energy consumption by nearly 96.5% when comparing the medians. To clarify these issues, we perform a statistical test on all pairs with and without red APIs to verify if there are statistically significant differences in our results. We begin by evaluating whether the values of the tests for each API followed a parametric distribution. After rejecting

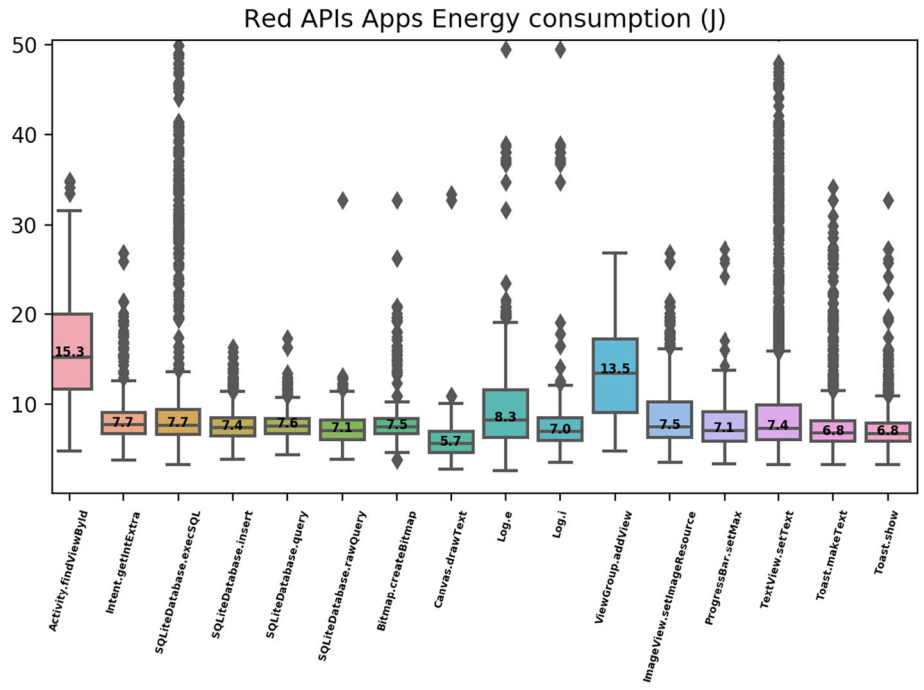

Fig. 7 Energy consumption of tests with Red APIs

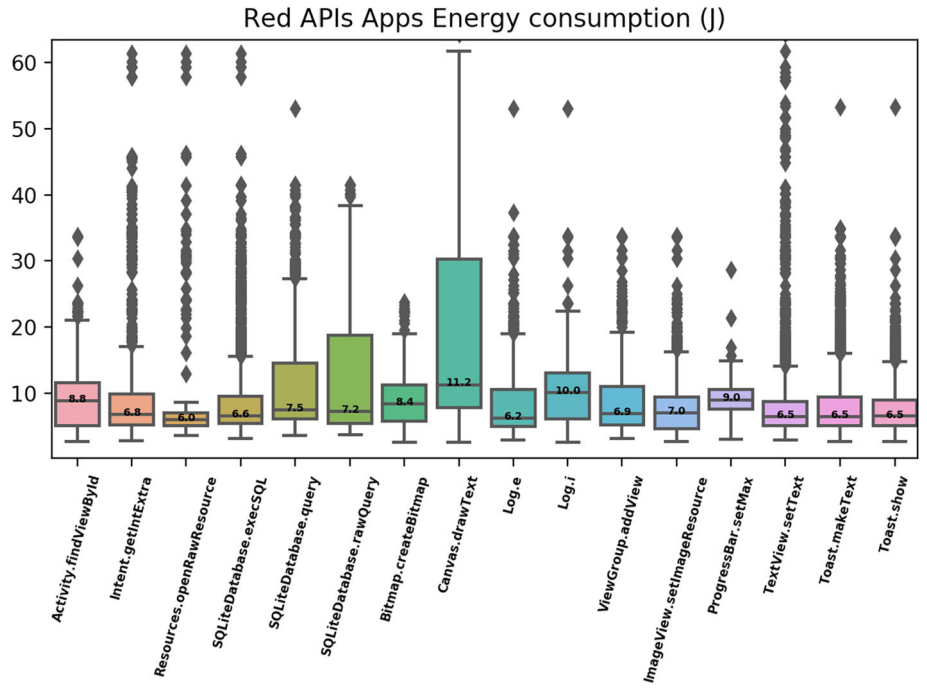

Fig. 8 Energy consumption of tests without Red APIs

this hypothesis for all the APIs, through both Shapiro and Wilk (1965) and D'Agostino (1971), we apply the Mann and Whitney (1947) U test to compare pairs with/without the invocation of each Red API. In order to evaluate if there are significant and statistically supported differences between the pairs, we evaluate the following hypothesis for each performance indicator under analysis: $\boldsymbol{H}_0$: The values obtained for tests with or without Red APIs invocations follow the same distribution; $\boldsymbol{H}_1$: The values obtained for tests with or without Red APIs invocations do not follow the same distribution.

The results of these hypothesis tests are shown in Table 10 (left columns), as well as a classification of each API as *Red* or not, obtained by comparing the medians of the pairs with and without Red API. If the tests using the API were less efficient than the ones not using it, it was classified as *Red* (marked by ✓). Otherwise, the APIs are marked with ✗ and didn't evidence performance-greedy behavior in our context. In regards to energy, the Mann-Whitney U test rejected H0 in practically all pairs, except for the *rawQuery* and *createBitmap* methods. This represents the case where we could not draw any conclusions regarding its performance (marked by =). This shows, with statistical support, that there are indeed differences in energy consumption when using, or not using, these APIs. However, based on the median and average values, the tests using the *drawText* and *Log.i* APIs show that such differences might sometimes even be positive within apps. In fact, we see that calling these two APIs does reduce energy consumption, thus, making their classification as red API invalid in our context.

In Fig. 9 we present results of the memory consumption of the same APIs. Once again, we see considerable differences in memory consumption in these APIs. The API methods

Table 10 P-values and resultant Hypothesis of assessing if there were differences in test performance using Red APIs

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

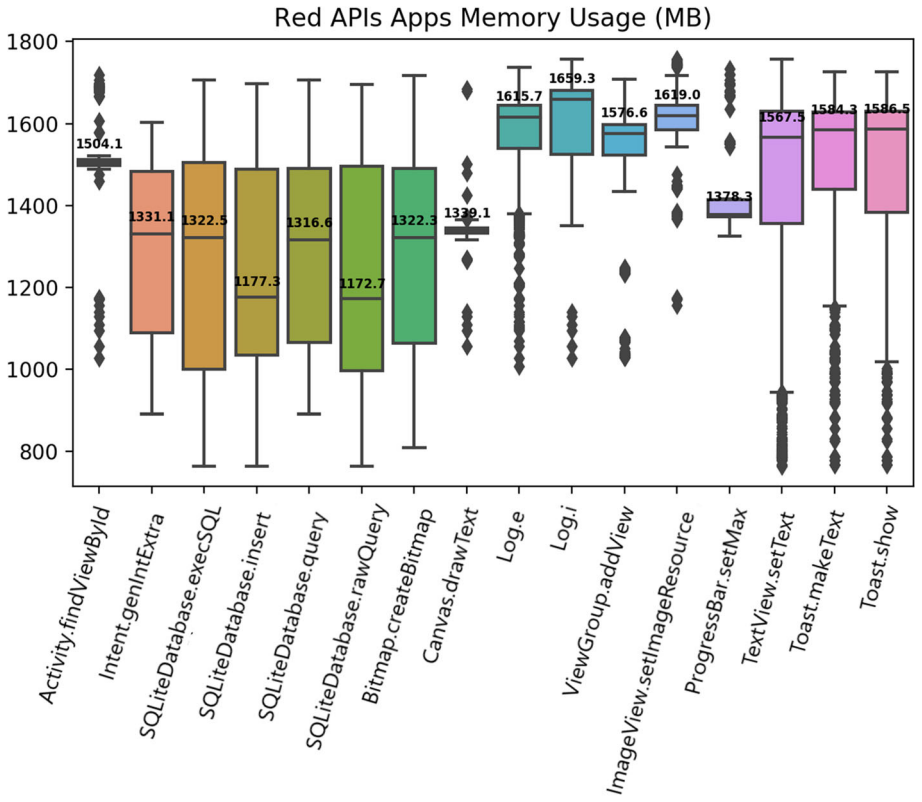

Fig. 9 Memory used by tests with Red APIs

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

Figure 10 presents the results of our third performance indicator: run-time. The *findViewById* and *addView* APIs stand out negatively from the other red APIs considered. The former presents tests with median values twice above that of many other red APIs (15.3 Joules), with the latter standing out from the others, but not in the same magnitude (values between 30 to 40% higher in terms of median). Execution time is the measured performance indicator in which the Red APIs demonstrate less impact during our performed tests. The results in Table 10 (rightmost columns), for the evaluation of H0 and H1, show that there are 5 Red APIs that do not show statistical differences in terms of execution time with their invocation.

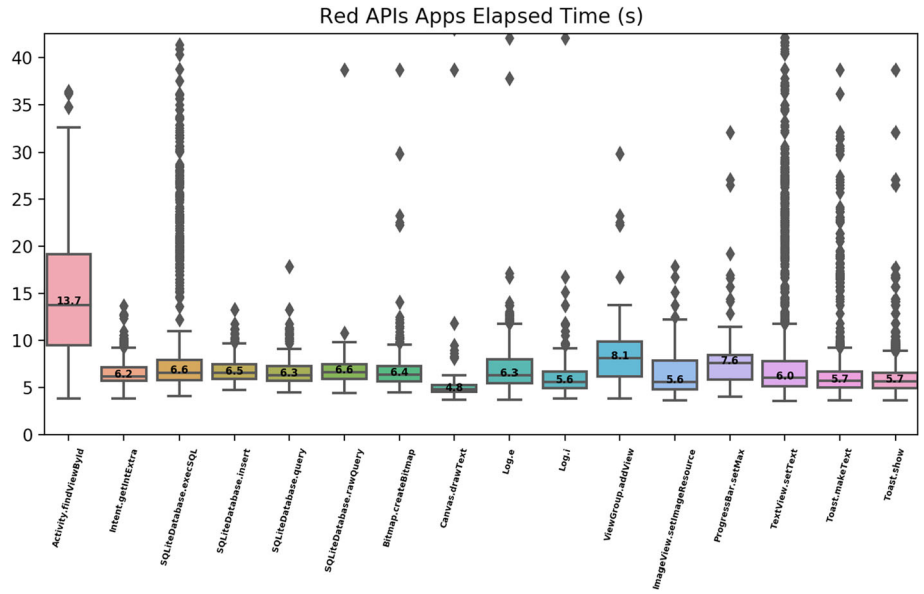

Fig. 10 Elapsed time of tests with Red APIs

4.2.2 Android Lint Issues

Android Lint is a static code analysis tool that checks your Android project source files for potential bugs and performance issues. Such performance issues have been used in several research studies which mainly focus on memory/run-time performance. In this section, we analyze and compare Lint performance issues in terms of run-time, memory and energy consumption. As of this date, there are 36 performance issues reported by Lint. In the context of our application repository, we identify 18 of such issues which are present in at least 25 versions of more than 10 different applications.

Figures 11, 12, 13 present boxplots for the three performance indicators for each one of the 18 evaluated issue. The presented values are in Joules, MBytes, and seconds, respectively. In the middle of each boxplot, we show the median value. In cases where no performance issues were detected in an app under test, it is shown as *None*.

In terms of energy consumption, as we can see in Fig. 11, the *DisableBaselineAlignment*, *ObsoleteLayoutParam*, and *UselessLeaf* Lint issues stand out negatively, with higher whiskers, 1st quartile and median values. On the other side, the *VectorPath* and *Recycle* issues are more energy efficient, but do not have consistently lower consumption values when compared to the others.

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

Finally, when it comes to execution time, the *DisableBaselineAlignment* and *UselessLeaf* Lint issues stand out in the negative, just as they did in terms of energy. Both display higher whiskers, 1st quartile, and median values over all others. *UseValueOf* and *Recycle* have a

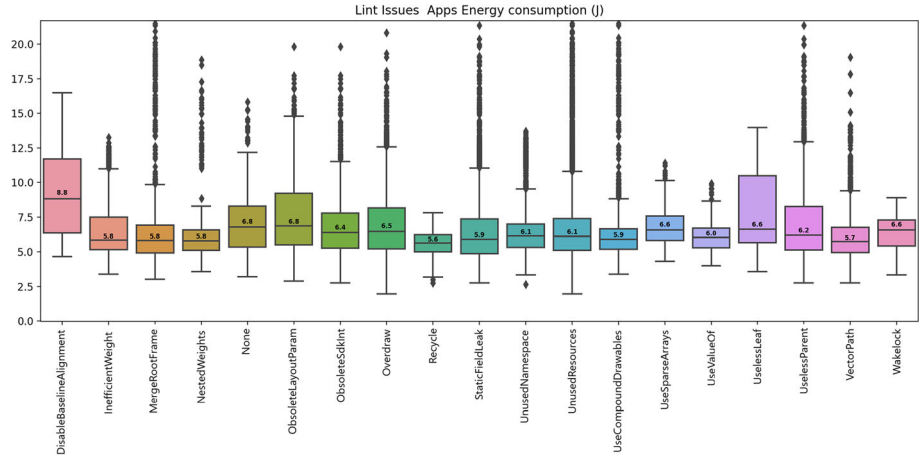

Fig. 11 Lint issues energy consumption

consistently lower run-time than the others. Additionally, most of the issues also have a lower run-time than *None*. This means that in our study, these issues did not have a noticeable impact on app run-time performance.

To assess whether there are statistically significant differences in behavior, in terms of performance across the 3 analyzed components, we evaluated the following hypothesis for each performance indicator: $\boldsymbol{H}_0$: The values obtained for apps with Lint issues follow the same distribution; $\boldsymbol{H}_1$: The values obtained for apps with Lint issues do not follow the same distribution.

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

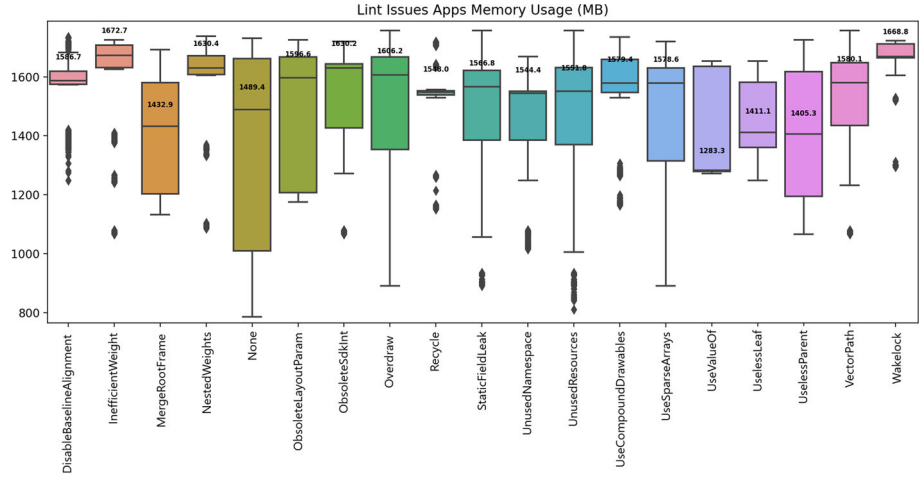

Fig. 12 Lint issues memory consumption

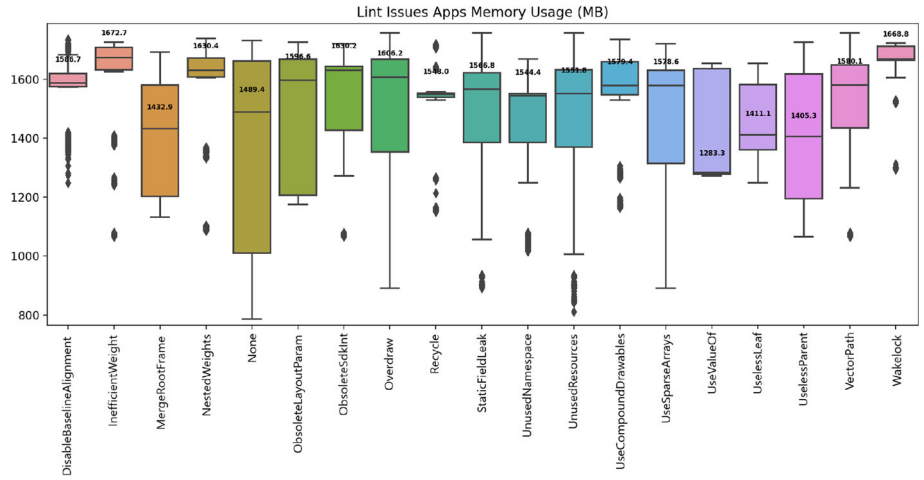

Fig. 13 Lint Issues run-time

Since the Lint issues are accompanied by different levels of *Severity* and *Priority*, we conceived the possibility of measuring the correlation between these levels and each of the performance indicators under analysis. As all the issues we evaluated in this section are of *Severity: Warning*, we assessed the correlation between *Priority* levels (scale from 1 to 10) and each of the components. For this purpose, the Spearman's Rank Correlation Coefficient (Spearman 1904) of each pair was calculated (*Component*,*Priority*). Using the obtained ρ, we applied the Rea et al. (2016) rule to obtain a classification of the level of correlation between the pairs. The results of this analysis are shown in Table 11, showing that the performance indicators have little correlation with the *Priority* of these issues.

4.3 Alternative Libraries

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

Given the diversity of the analyzed apps in this study, we present a comparative study of alternative libraries for common Android programming tasks. In order to answer **RQ3:** *Are there significant performance differences between apps that use different competing libraries implementing typical programming tasks?*, we selected a set of 7 programming tasks typically

Table 11 Correlation results between performance indicators and Lint issues priority

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

performed by Android apps. These libraries are listed in Table 12, organized in networking over HTTP, Collections, Image rendering/loading, JSON manipulation, I/O, Logging and Threading.

Taking into account the extent of the analysis carried out, we needed to select a limited set of tasks to be considered. Furthermore, we considered tasks with at least 2 alternatives/libraries,

Table 12 Evaluated libraries

Library	#Apps	Versions	Tests
HTTP			
Volley	6	25	79
OkHttp	16	87	447
HttpURLConnection	7	57	232
Retrofit	16	85	399
Picasso	9	34	164
Collections			
Sparse*Array	9	38	20
List	159	865	244
Set	129	692	147
*Map	71	325	127
Apache Collections	1	1	1
JSON			
Logan	0	0	0
Moshi	1	5	22
GSON	22	133	133
Jackson	1	3	3
org.JSON	22	96	405
I/0			
java.IO	89	400	1771
java.NIO	6	34	96
Okio	0	0	0
Apache IO	1	3	3
Logging			
util.Log	90	489	1807
Slf4j	1	3	3
Timber	6	21	158
Threading			
GreenRobot	6	14	86
RxJava	3	13	102
Executor	5	13	152
Kotin Coroutines	9	29	223
Image rendering/loading			
Volley	6	25	25
Glide	3	23	16
Picasso	9	34	164

which have methods to be invoked in at least 5 different apps, across 10 different versions, and with 25 tests different test scenarios. The libraries under analysis are some of the most used libraries for Android for each category, according to AppBrain (2020).

In the following sections, we present results of evaluating the performance of HTTP, JSON and Threading libraries. We selected these three tasks because they were the tasks with the largest number of alternative libraries to meet our selection criteria when excluding Image rendering/loading and Collections. We did not present a deep analysis of the Image rendering/loading task since its libraries are also contained in the set of the HTTP libraries and we also didn't consider Collections since these were already analyzed in terms of energy efficiency on the Android platform (Oliveira et al. 2019). The results obtained for the remaining libraries present are available in the online appendix.

4.3.1 HTTP Libraries

HTTP requests are the main technique for transferring data in web apps, a key aspect of a software ecosystem targeting mobile devices. In this section we analyse the most widely-used HTTP libraries to perform communication via HTTP: *HTTPURLConnection*, *Volley*, *Okhttp*, *Retrofit*, and *Picasso*. *HTTPURLConnection* was the first of these libraries to appear in Android development. *Volley* was released in 2013 as an alternative to *Okhttp*, offering more options such as multiple connections, request scheduling and cancellation APIs, working over *HTTPURLConnection*. *Retrofit* is a type-safe HTTP Client, more oriented to REST API access offering conversion to many common formats, while *Picasso* is optimized for downloading and caching media formats. Figures 14, and 15 show a comparison of the performance values for 5 different libraries:

As shown in Fig. 14, *Volley* is the most energy-efficient library, while *Picasso* is the most inefficient library in our study. The median values show that *Okhttp* consumes 43% more energy than *Volley*. *HTTPURLConnection* also has significant energy gains over the 3 most ineffective ones, resulting in an 18% energy reduction when compared to *Okhttp*. By looking at the medians of the boxplots, *Volley* is the most energy-efficient alternative with 7.5 Joules, with Picasso being the most inefficient with a consumption 12% higher.

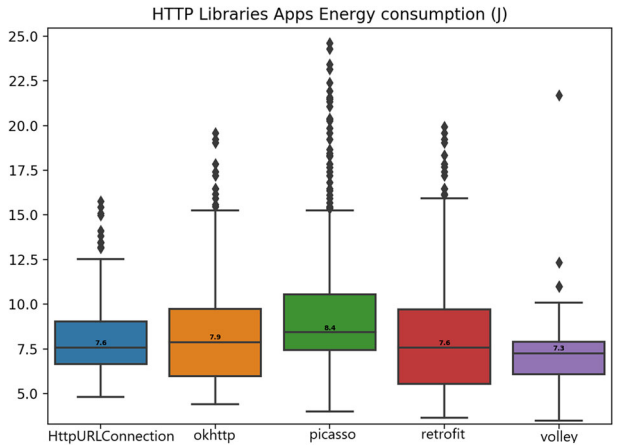

Fig. 14 HTTP Libraries energy consumption

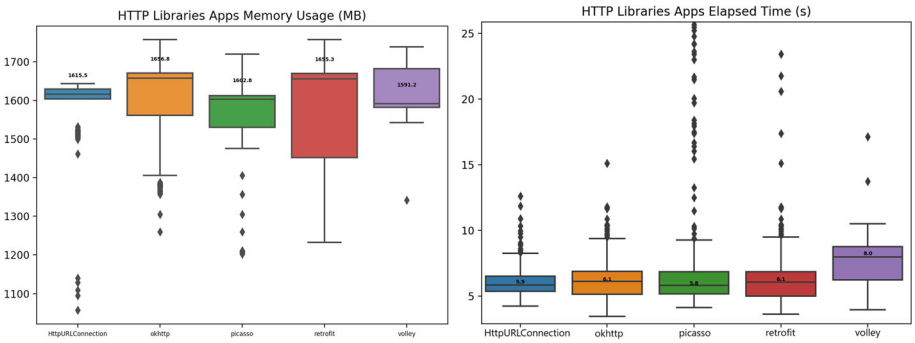

Fig. 15 HTTP Libraries memory consumption (left) and run-time (right)

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

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

4.3.2 Threading Libraries

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

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

In Figs. 16 and 17 we show results of three threading libraries and the (re)use of Kotlin coroutines.

As shown in Fig. 16 there is a clear decrease in energy consumption when using the *Kotlin coroutines* mechanism. *GreenRobot* also shows gains when compared to both *Executor* and

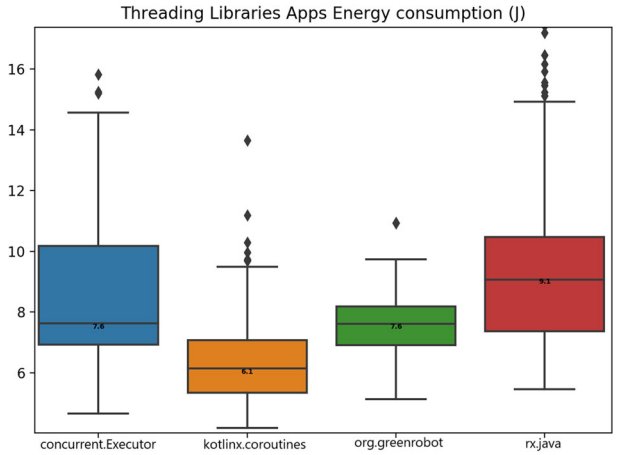

Fig. 16 Threading Libraries energy consumption

RxJava. The most inefficient energy consumption behavior comes from *RxJava*. *Executor* has a substantial gain over *RxJava*, obtaining similar results to *GreenRobot*.

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

When we consider run-time, the results are similar to those from the energy consumption analysis. *Kotlin coroutines* presents a significant gain in relation to the other libraries, presenting a run-time efficiency gain of approximately 32% when compared to *RxJava*, which is the worst evaluated alternative. Just below *Kotlin coroutines, GreenRobot* is once again distinguished from *Executor*, as the second most efficient library in terms of execution time.

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

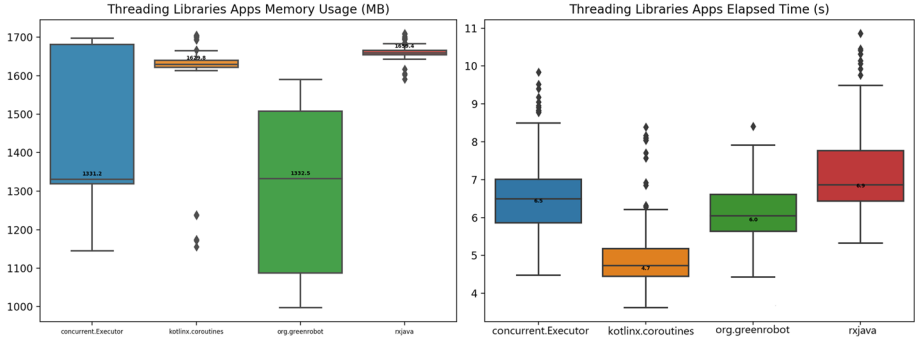

Fig. 17 Threading Libraries memory and run-time

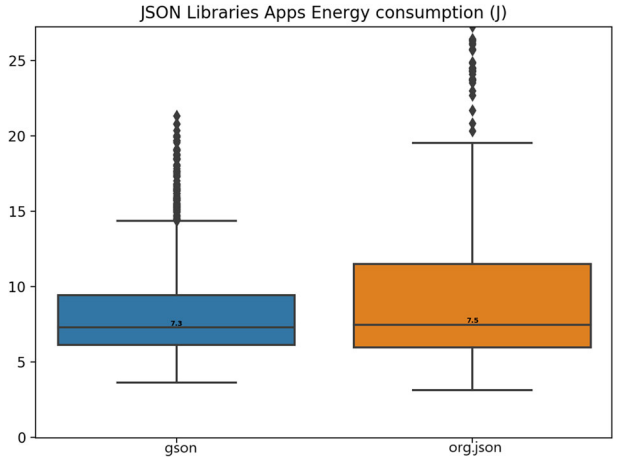

Fig. 18 JSON Libraries energy consumption

4.3.3 JSON Libraries

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

The results of our study are presented in Figs. 18 and 19.

Figure 18 shows that *GSON* has a marginal gain of 2.6% in terms of energy efficiency (considering the medians for comparison) when compared to *org.JSON*. *GSON* is also the fastest library: it is 12.6% faster than *org.JSON* as shown in Fig. 19. In terms of memory consumption, however, *GSON* shows a slightly higher memory usage (0.94% considering the medians).

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

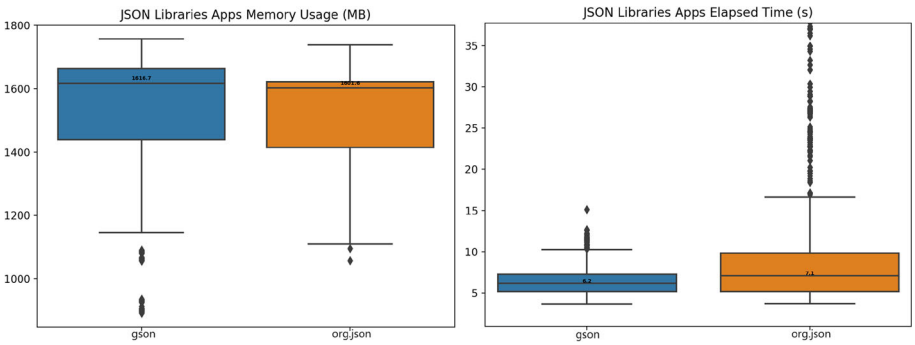

Fig. 19 JSON Libraries memory and run-time

5 Contributions

In this section, we discuss the results whose analysis allowed us to answer the research questions presented in Section 1. We provide answers to these questions and also elaborate on the reasons for some of the observed results. Moreover, we define a set of programming guidelines based on the results extracted from the process of conducting this study. Finally, as a result of our study we compiled a resume of some of the information contained in our dataset, which contains all the data produced when conducting the study. This large dataset is also presented as a contribution and it is openly available to the scientific community for further analysis.

5.1 Research Answers and Related Findings

5.1.1 Research Question 1

In Section 4.1 a set of changes of various types that could occur in the behavior and in the source code of the applications were presented. These changes were evaluated in a set of 44 application version changes with relevant energetic behavior change (greater or greater than 20%). After analyzing the 44 application version changes, we synthesized the conclusions that allow us to answer the first research question (**ARQ1**). We concluded that changes more associated with relevant impact on the app's energy performance are changes that add *functionality* and *bugfixes* in applications, followed by adding *layout* and/or *strings* and *refactors*. Furthermore, we can conclude that semantic changes are the type of changes (according to the criteria presented in Table 4) more associated with a significant impact on energy consumption, while changes such as *Project Building* reveal an unclear impact. A negative impact on performance was expected for changes such as the addition of *functionalities*, as it typically results in an increase in computation and resources to the application which translates into an expected loss of performance. Also, since the testing procedure with the Monkey framework only interacted with the application for a few seconds, the tests mostly capture the app startup and initial setup workflow. Thus, since these changes are translated into more class variables to be loaded, they have a negative impact on app performance. Since a greater use of resources and code to be loaded by the virtual machine incurs additional effort, even if the code of the new functionality was not directly executed, additional functionality can impact negatively the apps' performance. An evaluation of which types of bugfixes or addition of functionalities have more (evident) impact on performance was not evaluated. This was due to the fact that we concluded that the amount of data that we collected to categorize different types of functionalities or bugfixes was not sufficient the derive general conclusions regarding its impact.

> ***[ARQ1]: Which program changes have more impact on an Androids applications energy consumption?*** The changes more associated with relevant impact on the app's energy performance are changes that add *functionality* and *bugfixes* in applications, followed by adding *layout* and/or *strings* and *refactors*.

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

The impact of *bugfixes* can also be explained by the fact that its prevention avoids unnecessary computations to report the exception (e.g. logging the exception) and any system interruptions to stop and kill the application process, among others. The performance loss could be expected when it comes to significant changes in layout and strings since the emergence of new entities of this type results in static fields generated in the Android R.class Java class. This is an auto-generated class by AAPT (Android Asset Packaging Tool) that contains resource IDs for all the resources of the *res/* directory. When a component is declared in the XML file, aid for the corresponding component is automatically created in this class. However, this generation occurs during the compilation process and to translate into a significant loss of performance, it would have to be due to the creation of a large number of components of this type. In the case of adding new layout components (often associated with increased *functionality*), their creation and use translate into an expected loss of performance, since it is necessary at run-time to inflate an XML file in a View class, which is a heavy operation and has to be performed on single-thread.[9] The impact of including new views on energy consumption was also pointed out in Linares-Vásquez et al. (2014).

For changes at Project Building level (according to Table 4), the impact is unclear and lacks a deeper analysis with a more extensive dataset. For example, the observed behavior for the impact of updating dependencies on the project is inconsistent, since it was observed that changes at this level have a significant positive impact on energy in 30% of cases and negative in 33% of cases. The addition of new dependencies reveals a similar behavior, with updates at the Android SDK level being associated with a negligible energy impact in 55% of the time (**F1**).

5.1.2 Research Question 2

In Section 4.2, we present results for two distinct types of widely-studied practices in the literature labeled inefficient ones: Linares et al. Red APIs and Android Lint Issues. These practices were evaluated in our execution context, with different applications and interactions. Based on these results, both the analysis of the Red APIs and the Android Lint Issues, we were able to provide our answer to RQ2 (**ARQ2**).

[ARQ2]: Do previously identified inefficient programming practices exhibit the same behavior with larger datasets and different contexts? No, previously identified inefficient programming practices do not always exhibit the same behavior with larger datasets and different contexts. For instance, many of the Red APIs did not show abnormal performance behavior in our execution context.

In our executed apps and environment, some of the mentioned programming practices did not show abnormal performance behavior. In the case of Red APIs, some of the methods classified as energetically inefficient showed little difference in terms of energy consumption. The Man-Whitney test was used to assess whether the performance of apps invoking these APIs was different from the same apps that did not invoke such APIs. This test revealed

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

statistically significant differences between these pairs but does not indicate whether such differences translate into a gain or loss in performance. Looking at the boxplots and respective Red classifications in Table 10, it is clear that these practices have obvious impacts, of different magnitude, on performance, and that several APIs (40%) are more often associated with energy efficient executions in these apps, such as *setMax* or *Log.i*. In the case of the evaluated Lint issues, while their presence does impact performance, it is not always a clear gain or loss in performance. Additionally, the *Priority* levels have very little correlation with actual performance measurement values (**F2**). For example, in terms of energy consumption and run-time, the vast majority of the tested apps, with identified Android Lint issues, have lower measurements than apps that do not have any of these issues. The performance indicator with the most apparent impact on these practices is memory usage, which displayed much higher median values when compared with tested apps with no identified issues. The reason for obtaining different results than expected regarding these practices seen as energy inefficient is probably because they were evaluated in a context of execution different from the context in which they were initially identified. As described by Pinto and Castor (2017), the execution of an application depends on 4 different factors, where 3 of them (software system under execution, hardware combination, and given time) in our execution context are different from the previous ones. In the particular case of APIs identified by Linares-Vásquez et al. (2014), some of these APIs have already been analyzed from a performance point of view, having shown that their consumption depends on several factors inherent to the way they are used. For instance, in the case of logging APIs, prior studies concluded that Chowdhury et al. (2018); Zeng et al. (2019) logging frequency is a factor with a significant impact on the performance of these APIs. These studies allowed us to conclude that logging at a limited rate has no significant impact on performance and factors such as logging rate, disk flush and message size have a significant impact on energy consumption.

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

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

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

5.1.3 Research Question 3

Section 4.3 presents the performance comparison between apps that used competing, widely-used libraries that are used to implement three typical programming tasks in mobile applications: JSON, HTTP and Threading libraries. The analysis of the results of these libraries allows us to answer RQ3 ((**ARQ3**)).

[ARQ3]: Are there significant performance differences between apps that use different competing libraries implementing typical programming tasks? There are, in fact, significant and macroscopic differences in the performance of real-world Android apps that use alternative libraries for common Android programming tasks. This evidence suggests that these libraries have significantly different performance impacts and the choice of the library can impact significantly the performance of the implemented task.

In the case of libraries for HTTP, apps that use *Volley* tend to have more energy-efficient tests, suggesting that this API might be the most energy-efficient API, despite being the worst in terms of run-time when considering the median and top whisker of the box plots of the Figs. 14, and 15. In our experiment, *Picasso* seems to be the most memory and run-time efficient, being the worst in terms of energy. These results are different from the ones from another research study (Lachgar et al. 2018) that analyzed the run-time performance of *Volley* and *Retrofit*, having concluded that *Volley* is faster and offers more features than *Retrofit*. This divergence can be again justified by the fact that our setup consists of a different execution context, considering different real-world applications and many execution contexts, while the experimental setup used in Lachgar et al. (2018) considered only one application connected to a custom server, aiming at analyzing and measuring only specific methods of these APIs. However, in Fig. 15, we can see that despite *Volley* obtained a higher median, tests using *Retrofit* obtained less consistent results, where many tests had high-consuming tests above the upper percentile when comparing to the values of the tests that used the *Volley* library.

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

5.2 Programming Guidelines and Other Findings

This section presents several guidelines that were defined based on the collected results and their analysis. Such guidelines aim to steer software developers and the scientific community in developing greener Android software. This section also presents several findings that were derived from the analysis of the data presented in Section 4.

The data analyzed the Section 4, which allowed us to answer the second and third research questions, aims to advise developers to **always test their apps and evaluate their app performance individually** (**G1**), especially on devices and environments similar to where they will be executed. For instance, As can be observed by the Figs. 7 and 9, there are many outliers in the indicators measured in the tests that invoke these APIs. This suggests that the classification of these APIs as energy-greedy or energy-efficient hardly depends on the context (Pinto and Castor 2017; Li et al. 2020). For instance, some of these APIs are Logging APIs, whose energy performance was analyzed in prior publications (Zeng et al. 2019; Chowdhury et al. 2018). For instance, Chowdhury et al. (2018) demonstrated that factors such as logging rate, disk flush and message size have a significant impact on energy consumption. Nevertheless, the Android ecosystem, its APIs and its development environment are constantly changing and most of the contributions and performance analyses made by the community relate to specific execution contexts and scenarios that may be out of date and far from more realistic and updated ones. In the case of APIs or patterns identified as inefficient, they can present such behavior only in certain contexts (size and type of arguments, system load, signal quality), sometimes exhibiting behaviors with negligible impact on the app's performance in most of the cases.

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

In addition to developers having to evaluate the resource consumption of their apps individually, they must also **identify the main bottlenecks of their apps and target devices**(**G2**). In the case of the evaluated libraries, they showed different behaviors in terms of performance, with no single one being considered the overall best for all the evaluated components. Thus, there are APIs that, for example, are recommended whenever energy consumption is a concern, while they should be avoided when memory usage is a concern. In this sense, based on the analysis performed to evaluate the performance footprint of the considered Android libraries, we defined Table 13 (**G3**). This table represents a guideline in order to select the most efficient/inefficient libraries for each evaluated programming task, based on the median values of the app tests of our large-scale analysis. Using this information, practitioners can have a better understanding of which libraries they should choose or avoid, considering each performance indicator. For example, for HTTP libraries, *Volley* seems the best choice if both energy consumption and memory usage is of concern, but was the worst performing in terms of run-time efficiency.

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

Table 13 Most efficient/inefficient libraries **[G3]**

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

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

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

The long process of extracting and executing the applications, as well as the process of analyzing the results, also allowed us to understand what can be improved in our execution process and what we can recommend to researchers who intend to replicate or conduct similar studies. The first recommendation is that they **use the AnaDroid execution pipeline**(**G4**), used by us to conduct this study. Besides the features referred to in Section 3, such as source code instrumentation, automatic building, and run-time error detection, among others, during the writing and publication of this document, this execution pipeline is being redesigned[10] and continuously improved and extended to support new energy profilers and testing tools. Furthermore, it is completely open-source and extensible, also allowing the easy replication and validation of later studies to be elaborated with this tool. It can be used to benchmark Android applications' performance, from its source code or even from an already-built APK. Furthermore, it supports a new energy profiler (Rua et al. 2022), which can be used on most of the latest Android devices to analyze the energy consumption of your software.

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

[10] Redesigned version of AnaDroid: https://github.com/greensoftwarelab/PyAnaDroid

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

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

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

The results presented in Section 4 aiming to present a taxonomy of the considered applications in our study also allowed us to observe other interesting findings that characterize the current Android software development paradigm. One of the observations that the analysis performed allowed us to provide is that **App's age does not hint expected performance**(**F4**). We analyzed the data in Fig. 4, which shows the relation between the energy consumption of the apps and their release year. Our analysis suggests that besides a noticeable rise since 2017, there is no clear tendency for to apps increase energy consumption over time, even when the energy is being measured in the same device. We also observed the same in terms of memory and run-time.

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

5.3 Dataset

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

The data contained in this dataset presents results from more than one month of continuous test execution. Given the extent of the work presented, from the process of extracting and filtering applications to the agglomeration and obtaining of its results, it is pertinent to share it with the community so it can be used in further studies aimed at analyzing the performance of applications or devices, comparisons of Android testing frameworks, evaluation of software evaluation aspects, between others. In this way, it is intended to contribute to the increase of knowledge regarding the factors that influence the performance of Android applications

and their respective weight. Some examples of other research questions that are examples of further research directions (**RD**) that might be answered by analyzing the data contained in this dataset or possible expansions of it are:

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

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

These data contain a large set of information regarding open-source applications and respective code, as well as static and dynamic metrics regarding their execution and execution environment. These metrics were collected at different levels of granularity, and their value/result can be attributed to an application, a test performed on an application, or a block of code belonging to the application. In Table 14 are the metrics collected during the entire application execution process. Each metric collected is presented in the table accompanied by a description and the level at which it was obtained:

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

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

The collection of all these metrics has led to it being possible to collect more than 1,200,000 measurements of dynamic metrics related to executions on real devices and more than 420, 000 measurements of static metrics related to applications and their source code (see Table 15).

All these metrics result from the analysis of the files obtained from the execution of tests on applications with instrumented source code. These were derived from the analysis of the Trepn Profiler log files collected during the monitoring process and from the method traces collected during the application's execution. In addition to these files also being available in

Table 14 Metrics gathered during tests/apps execution

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

Table 15 Resume of the different metrics obtained

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

the dataset, the system logs (through the *logcat* tool) generated during the monitoring process were also collected. These logs contain dumps of system and app messages, including stack traces of exceptions occurred, which may also be useful for further studies aimed at evaluating the relationship between system events and application or device performance.

In addition to all this information that was collected for all running applications, data relating to the manual analysis of versions with anomalous energy performance (as described in Section 4.1) is also available in the online appendix. The analysis presents tables with the changes registered between each version change, which can be reused by the community to deepen the impact of the evaluated changes, either on performance or on other functional and non-functional aspects of the application's operation.

6 Threats to Validity

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

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

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

The set of collected apps has software pieces developed in a relatively diverse range, which varies between 2015 and 2020 (between 2010 and 2020 if we consider the first release year of each application). The fact that many of them presented APIS and recent languages like Kotlin, shows that they may be apps that represent the state of the art of Android development and can help to characterize the state in which the development is. The apps are of different categories and the versions collected are diversified and from different development stages.

However, the data we collect regarding the downloads of these in the Play Store reveals that, except for some executions, the open-source apps collected are not very popular with Play Store users, according to the download count reported by them. Nevertheless, many of these apps are not downloaded from the Play Store, but rather downloaded from 3rd party stores such as F-Droid or Aptoid, or their official page or repository, encouraging the movement of open-source software.

Our corpus of applications contained only open-source applications extracted from open-source repositories. Despite studying open-source apps being a widely-used procedure by the scientific community (Couto et al. 2015; Rua, Couto, and Saraiva 2019; Das et al. 2020; Li et al. 2020; Ribeiro et al. 2021), considering closed-source apps could help to obtain more universal and representative conclusions from the Android platform. However, analyzing source code from closed-open-source apps is a technically more elaborate process, since it involves the tasks of decompile or unobfuscate code. The automatic execution process used in this work already involved an application processing pipeline that caused a significant number of applications to be excluded from the study due to problems encountered during the building process, instrumentation or automatic execution. Adding even more complex steps to the process to consider (eventually more popular or even robust) closed-source apps would likely result in a reduced set of applications to analyze.

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

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

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

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

7 Conclusions and Future Work

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

In our execution context, we manually analyzed a set of 31 apps and 44 versions with anomalous energy variations to identify software changes with a significant impact on app performance. From our results, we concluded that from the 15 software changes considered in our analysis, only the addition of functionalities, bug fixes, refactors, adding layouts, and strings have a noticeable impact on app performance. By re-evaluating the applicability of

conclusions gathered in previous performance studies, such as Linares et al.and the Android Lint Issues, we concluded that there are significant differences in the performance of practices already identified as harmful in this sense. Regarding the study of Linares et al., we observed that only 40% of the evaluated energy-greedy APIs had a noticeable impact on app performance, with the remaining having negligible impact. Furthermore, regarding Lint Issues, we observed that many of these issues did not reflect their supposed negative impact on app performance. Additionally, we concluded that the severity level that accompanied issues, labeled as performance issues, has very little correlation with the correspondent performance.

Finally, it has been shown that alternative libraries used to perform common programming tasks in Android apps had different impacts on the apps' energy consumption and that there might be significant gains in performance just by migrating to a more efficient library. This contribution was made in a macroscopic way, being able to distinguish versions of these libraries in future works, as well as the respective methods they provide. In this sense, in Section 5 we present a table containing the most efficient and inefficient libraries for each task and performance indicator evaluated in our study. Also in Section 5 we present other guidelines that result from the analysis of our data.

All the collected data is available for the community to analyze and reuse. There are already ongoing works that aim to use this data to carry out analysis regarding the evolution of apps and their code from a semantic point of view, since the results contain code and respective executions of a large set of versions of the same apps. This study aims to understand how Android apps and their consumption evolve throughout their development cycle, since they tend to extend core functionalities and improve UI over time (Jha et al. 2019).

The collected and analyzed apps were not chosen specifically for the presented research questions in mind. The results are just an example of the type of analysis that can be done with the collected data and through dynamic performance analysis. There are still many other possibilities to increase knowledge in this area from this collected data, or from new data that can be obtained with this same procedure, i.e. the evaluation of different testing frameworks for different objectives, exploring code coverage, predicting and evaluating app performance, detecting and identifying errors and patterns, exploring GUI's, etc., among others. Thus, since this work considered several versions of each app, the collected data is also suitable for carrying out studies related to the evolution of mobile software.

Acknowledgements This work is partially financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project UIDP/50014/2020, by COST Action 19135: "CERICIRAS - Connecting Education and Research Communities for an Innovative Resource Aware Society", and by Erasmus+ project No. 2020-1-PT01-KA203-078646: "SusTrainable - Promoting Sustainability as a Fundamental Driver in Software Development Training and Education". The first author is also financed by FCT grant SFRH/BD/146624/2019. The information and views set out in this paper are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

Funding Open access funding provided by FCT|FCCN (b-on).

Data availability The datasets generated during and/or analysed during the current study are available in its online appendix (Rua 2022).

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

AppBrain (2020) App brain libraries statistics. https://www.appbrain.com/stats/libraries. Accessed 10 Feb 2021

Bangash AA, Ali K, Hindle A (2022) A black box technique to reduce energy consumption of android apps. In: Proceedings of the ACM/IEEE 44th international conference on software engineering: new ideas and emerging results, ICSE-NIER '22. Association for Computing Machinery, New York, NY, USA, pp 1–5. https://doi.org/10.1145/3510455.3512795

Bangash AA, Tiganov D, Ali K, Hindle A (2021) Energy efficient guidelines for IOS core location framework. In: 2021 IEEE international conference on software maintenance and evolution (ICSME), pp 320–331. https://doi.org/10.1109/ICSME52107.2021.00035

Biørn-Hansen A, Rieger C, Grønli TM, Majchrzak TA, Ghinea G (2020) An empirical investigation of performance overhead in cross-platform mobile development frameworks. Empir Softw Eng. https://doi.org/10.1007/s10664-020-09827-6

Cañete A, Horcas JM, Ayala I, Fuentes L (2020) Energy efficient adaptation engines for android applications. Inf Softw Technol 118:106220. https://doi.org/10.1016/j.infsof.2019.106220. https://www.sciencedirect.com/science/article/pii/S0950584919302307

Choudhary SR, Gorla A, Orso A (2015) Automated test input generation for android: are we there yet? (e). In: Proceedings of the 2015 30th IEEE/ACM international conference on automated software engineering (ASE), ASE '15. IEEE Computer Society, Washington, DC, USA, pp 429–440. https://doi.org/10.1109/ASE.2015.89

Chowdhury S, Di Nardo S, Hindle A, Jiang ZM (2018) An exploratory study on assessing the energy impact of logging on android applications. Empirical Softw Engg 23(3):1422–1456. https://doi.org/10.1007/s10664-017-9545-x

Chowdhury S, Borle S, Romansky S, Hindle A (2019) Greenscaler: training software energy models with automatic test generation. Empirical Softw Eng 24(4):1649–1692. https://doi.org/10.1007/s10664-018-9640-7

Couto M, Borba P, Cunha J, Fernandes JP, Pereira R, Saraiva J (2017) Products go green: worst-case energy consumption in software product lines. In: Proceedings of the 21st international systems and software product line conference, vol A, pp 84–93

Couto M, Cunha J, Fernandes JP, Pereira R, Saraiva J (2015) Greendroid: a tool for analysing power consumption in the android ecosystem. In: 2015 IEEE 13th international scientific conference on informatics. IEEE, pp 73–78

Couto M, Cunha J, Fernandes JP, Pereira R, Saraiva J (2015) Greendroid: a tool for analysing power consumption in the android ecosystem. In: 2015 IEEE 13th international scientific conference on informatics, pp 73–78. https://doi.org/10.1109/Informatics.2015.7377811

Couto M, Pereira R, Ribeiro F, Rua R, Saraiva JA (2017) Towards a green ranking for programming languages. In: Proceedings of the 21st Brazilian symposium on programming languages, SBLP 2017. ACM, New York, NY, USA, pp 7:1–7:8. https://doi.org/10.1145/3125374.3125382

Couto M, Saraiva J, Fernandes JP (2020) Energy refactorings for android in the large and in the wild. In: 2020 IEEE 27th international conference on software analysis, evolution and reengineering (SANER), pp 217–228. https://doi.org/10.1109/SANER48275.2020.9054858

Cruz L, Abreu R (2017) Performance-based guidelines for energy efficient mobile applications. In: 2017 IEEE/ACM 4th international conference on mobile software engineering and systems (MOBILESoft), pp 46–57. https://doi.org/10.1109/MOBILESoft.2017.19

Cruz L, Abreu R, Rouvignac J (2017) Leafactor: improving energy efficiency of android apps via automatic refactoring. In: 2017 IEEE/ACM 4th international conference on mobile software engineering and systems (MOBILESoft), pp 205–206. https://doi.org/10.1109/MOBILESoft.2017.21

Cruz L, Abreu R, Rouvignac JN (2017) Leafactor: improving energy efficiency of android apps via automatic refactoring. In: 2017 IEEE/ACM 4th international conference on mobile software engineering and systems (MOBILESoft), pp 205–206. https://doi.org/10.1109/MOBILESoft.2017.21

D'Agostino RB (1971) An omnibus test of normality for moderate and large size samples

Das T, Penta MD, Malavolta I (2020) Characterizing the evolution of statically-detectable performance issues of android apps. Empir Softw Eng 25(4):2748–2808. https://doi.org/10.1007/s10664-019-09798-3

Goaër OL (2020) Enforcing green code with android lint. In: 2020 35th IEEE/ACM international conference on automated software engineering workshops (ASEW), pp 85–90. https://doi.org/10.1145/3417113.3422188

Goaër OL (2020) Enforcing green code with android lint. In: 2020 35th IEEE/ACM international conference on automated software engineering workshops (ASEW), pp 85–90. https://doi.org/10.1145/3417113.3422188

Gomez L, Neamtiu I, Azim T, Millstein T (2013) Reran: timing- and touch-sensitive record and replay for android. In: 2013 35th international conference on software engineering (ICSE), pp 72–81. https://doi.org/10.1109/ICSE.2013.6606553

Gonçalves N, Rua R, Cunha J, Pereira R, de Sousa Saraiva J (2022) Energy efficiency of web browsers in the android ecosystem. ArXiv abs/2205.11399

Google (2016) Trepn profiler android app. https://play.google.com/store/apps/details?id=com.quicinc.trepn. Accessed 10 Feb 2021

Google (2021) Android lint checks. http://tools.android.com/tips/lint-checks. Accessed 10 Feb 2021

Google (2021) Android profiler. https://developer.android.com/studio/profile/android-profiler. Accessed 10 Feb 2021

Google (2021) App crawler. https://developer.android.com/training/testing/crawler?hl=en. Accessed 10 Feb 2021

Google (2021) UI/application exerciser monkey. https://developer.android.com/studio/test/monkey. Accessed 10 Feb 2021

Habchi S, Moha N, Rouvoy R (2021) Android code smells: From introduction to refactoring. J Syst Softw 177:110964. https://doi.org/10.1016/j.jss.2021.110964. https://www.sciencedirect.com/science/article/pii/S0164121221000613

Hindle A (2012) Green mining: a methodology of relating software change to power consumption. In: 2012 9th IEEE working conference on mining software repositories (MSR), pp 78–87. https://doi.org/10.1109/MSR.2012.6224303

Hindle A (2013) Green mining: a methodology of relating software change and configuration to power consumption. Empir Softw Eng 20. https://doi.org/10.1007/s10664-013-9276-6

Hoque M, Siekkinen M, Khan K, Xiao Y, Tarkoma S (2015) Modeling, profiling, and debugging the energy consumption of mobile devices. ACM Comput Surv 48:40. https://doi.org/10.1145/2840723

Hort M, Kechagia M, Sarro F, Harman M (2022) A survey of performance optimization for mobile applications. IEEE Trans Software Eng 48(8):2879–2904. https://doi.org/10.1109/TSE.2021.3071193

Hu Y, Yan J, Yan D, Lu Q, Yan J (2017) Lightweight energy consumption analysis and prediction for android applications. Sci Comput Program 162. https://doi.org/10.1016/j.scico.2017.05.002

Jabbarvand R, Malek S (2017) ţdroid: an energy-aware mutation testing framework for android. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017. Association for Computing Machinery, New York, NY, USA, pp 208–219. https://doi.org/10.1145/3106237.3106244

Jha A, Lee S, Lee W (2019) An empirical study of configuration changes and adoption in android apps. J Syst Softw 156. https://doi.org/10.1016/j.jss.2019.06.095

Keong KC, Tieng Wei K, Abd. Ghani AA, Sharif KY (2015) Toward using software metrics as indicator to measure power consumption of mobile application: a case study. In: 2015 9th Malaysian software engineering conference (MySEC), pp 172–177. https://doi.org/10.1109/MySEC.2015.7475216

Kong P, Li L, Gao J, Liu K, Bissyandé TF, Klein J (2019) Automated testing of android apps: a systematic literature review. IEEE Trans Reliab 68(1):45–66. https://doi.org/10.1109/TR.2018.2865733

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621. https://doi.org/10.1080/01621459.1952.10483441. https://www.tandfonline.com/doi/abs/10.1080/01621459.1952.10483441

Lachgar M, Benouda H, Elfirdoussi S (2018) Android rest APIS: volley vs retrofit. In: 2018 international symposium on advanced electrical and communication technologies (ISAECT), pp 1–6. https://doi.org/10.1109/ISAECT.2018.8618824

Li X, Chen J, Liu Y, Wu K, Gallagher JP (2022) Combatting energy issues for mobile applications. ACM Trans Softw Eng Methodol. https://doi.org/10.1145/3527851. Just Accepted

Lima LG, Soares-Neto F, Lieuthier P, Castor F, Melfe G, Fernandes JP (2016) Haskell in green land: analyzing the energy behavior of a purely functional language. In: 2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER), vol 1. IEEE, pp 517–528

Lin JW, Salehnamadi N, Malek S (2020) Test automation in open-source android apps: a large-scale empirical study. In: 2020 35th IEEE/ACM international conference on automated software engineering (ASE), pp 1078–1089

Linares-Vásquez M, Bavota G, Bernal-Cárdenas C, Oliveto R, Di Penta M, Poshyvanyk D (2014) 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. ACM, New York, NY, USA, pp 2–11. https://doi.org/10.1145/2597073.2597085

Liu P, Li L, Zhao Y, Sun X, Grundy J (2020) Androzooopen: collecting large-scale open source android apps for the research community. MSR '20. Association for Computing Machinery, New York, NY, USA, pp 548–552. https://doi.org/10.1145/3379597.3387503

Liu J, Wu T, Deng X, Yan J, Zhang J (2017) Insdal: a safe and extensible instrumentation tool on Dalvik byte-code for android applications. In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), pp 502–506. https://doi.org/10.1109/SANER.2017.7884662

Li X, Yang Y, Liu Y, Gallagher JP, Wu K (2020) Detecting and diagnosing energy issues for mobile applications. In: Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, ISSTA 2020. Association for Computing Machinery, New York, NY, USA, pp 115–127. https://doi.org/10.1145/3395363.3397350

Machiry A, Tahiliani R, Naik M (2013) Dynodroid: an input generation system for android apps. In: Proceedings of the 2013 9th joint meeting on foundations of software engineering, ESEC/FSE 2013. Association for Computing Machinery, New York, NY, USA, pp 224–234. https://doi.org/10.1145/2491411.2491450

Ma X, Huang P, Jin X, Wang P, Park S, Shen D, Zhou Y, Saul LK, Voelker GM (2013) eDoctor: automatically diagnosing abnormal battery drain issues on smartphones. In: 10th USENIX symposium on networked systems design and implementation (NSDI 13). USENIX Association, Lombard, IL, pp 57–70. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ma

Maia D, Couto M, Saraiva J, Pereira R (2020) E-debitum: managing software energy debt. In: 2020 35th IEEE/ACM international conference on automated software engineering workshops (ASEW), pp 170–177. https://doi.org/10.1145/3417113.3422999

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other. Ann Math Statist 18(1):50–60. https://doi.org/10.1214/aoms/1177730491

Manotas I, Bird C, Zhang R, Shepherd D, Jaspan C, Sadowski C, Pollock L, Clause J (2016) An empirical study of practitioners' perspectives on green software engineering. In: International conference on software engineering (ICSE), IEEE/ACM 38th. IEEE, pp 237–248

Mazuera-Rozo A, Trubiani C, Linares-Vásquez M, Bavota G (2020) Investigating types and survivability of performance bugs in mobile apps. Empirical Softw Eng 25(3):1644–1686. https://doi.org/10.1007/s10664-019-09795-6

McCabe TJ (1976) A complexity measure. In: Proceedings of the 2Nd international conference on software engineering, ICSE '76. IEEE Computer Society Press, Los Alamitos, CA, USA, pp 407. http://dl.acm.org/citation.cfm?id=800253.807712

Mickle T (2018) Your phone is almost out of battery. Remain calm. Call a doctor. https://www.wsj.com/articles/your-phone-is-almost-out-of-battery-remain-calm-call-a-doctor-1525449283. Accessed 10 Feb 2021

Monsoon (2021) Monsoon power monitor. https://www.msoon.com. Accessed 10 Feb 2021

Nucci DD, Palomba F, Prota A, Panichella A, Zaidman A, Lucia AD (2017) Petra: a software-based tool for estimating the energy profile of android applications. In: 2017 IEEE/ACM 39th international conference on software engineering companion (ICSE-C), pp 3–6. https://doi.org/10.1109/ICSE-C.2017.18

Nucci DD, Palomba F, Prota A, Panichella A, Zaidman A, Lucia AD (2017) Software-based energy profiling of android apps: simple, efficient and reliable? In: 2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER), pp 103–114. https://doi.org/10.1109/SANER.2017.7884613

Oliveira W, Oliveira R, Castor F (2017) A study on the energy consumption of android app development approaches. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR), pp 42–52. https://doi.org/10.1109/MSR.2017.66

Oliveira W, Oliveira R, Castor F (2017) A study on the energy consumption of android app development approaches. In: 2017 IEEE/ACM 14th international conference on mining software repositories (MSR), pp 42–52. https://doi.org/10.1109/MSR.2017.66

Oliveira W, Oliveira R, Castor F, Fernandes B, Pinto G (2019) Recommending energy-efficient Java collections. In: Proceedings of the 16th international conference on mining software repositories, MSR '19. IEEE Press, Piscataway, NJ, USA, pp 160–170. https://doi.org/10.1109/MSR.2019.00033

Ortiz G, García-De-Prado A, Berrocal J, Hernández J (2019) Improving resource consumption in context-aware mobile applications through alternative architectural styles. IEEE Access 7:65228–65250. https://doi.org/10.1109/ACCESS.2019.2918239

Pang C, Hindle A, Adams B, Hassan AE (2016) What do programmers know about software energy consumption? IEEE Softw 33(3):83–89

Park J, Park YB, Ham HK (2013) Fragmentation problem in android. In: 2013 international conference on information science and applications (ICISA), pp 1–2. https://doi.org/10.1109/ICISA.2013.6579465

Pathak A, Hu YC, Zhang M, Bahl P, Wang YM (2011) Fine-grained power modeling for smartphones using system call tracing. In: Proceedings of the sixth conference on computer systems, EuroSys '11. ACM, New York, NY, USA, pp 153–168. https://doi.org/10.1145/1966445.1966460

Pereira R, Carção T, Couto M, Cunha J, Fernandes JP, Saraiva J (2020) Spelling out energy leaks: aiding developers locate energy inefficient code. J Syst Softw 161:110463

Pereira R, Couto M, Cunha J, Fernandes JP, Saraiva J (2016) The influence of the java collection framework on overall energy consumption. In: 2016 IEEE/ACM 5th international workshop on green and sustainable software (GREENS). IEEE, pp 15–21

Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JAP, Saraiva JA (2017) Energy efficiency across programming languages: how do energy, time, and memory relate? In: Proceedings of the 10th ACM SIGPLAN international conference on software language engineering, SLE 2017. ACM, New York, NY, USA, pp 256–267. https://doi.org/10.1145/3136014.3136031

Pereira R, Couto M, Ribeiro F, Rua R, Cunha J, Fernandes JP, Saraiva J (2021) Ranking programming languages by energy efficiency. Sci Comput Program 102609

Pereira R, Matalonga H, Couto M, Castor F, Cabral B, Carvalho P, de Sousa SM, Fernandes JA (2020) Greenhub: a large-scale collaborative dataset to battery consumption analysis of android devices. Empir Softw Eng

Pereira R, Simão P, Cunha J, Saraiva JA (2018) jstanley: placing a green thumb on Java collections. In: Proceedings of the 33rd ACM/IEEE international conference on automated software engineering, ASE 2018. ACM, New York, NY, USA, pp 856–859. https://doi.org/10.1145/3238147.3240473

Peters M, Scoccia GL, Malavolta I (2021) How does migrating to Kotlin impact the run-time efficiency of android apps? In: 2021 IEEE 21st international working conference on source code analysis and manipulation (SCAM), pp 36–46. https://doi.org/10.1109/SCAM52516.2021.00014

Pinto G, Castor F (2017) Energy efficiency: a new concern for application software developers. Commun ACM 60(12):68–75

Pinto G, Castor F, Liu YD (2014) Mining questions about software energy consumption. In: Proceedings of the 11th working conference on mining software repositories. ACM, pp 22–31

Qualcomm (2014) Qualcomm forum - which mobile devices report accurate system power consumption? https://developer.qualcomm.com/forum/qdn-forums/software/trepn-power-profiler/28349. Accessed 10 Feb 2021

Rea L, Parker RA, Allen R (2016) Designing and conducting survey research. Jossey-Bass Publishers

Ribeiro A, Ferreira JF, Mendes A (2021) Ecoandroid: an android studio plugin for developing energy-efficient Java mobile applications. In: 2021 IEEE 21st international conference on software quality, reliability and security (QRS), pp 62–69. https://doi.org/10.1109/QRS54544.2021.00017

Richter F (2019) The most wanted smartphone features. https://www.statista.com/chart/5995/the-most-wanted-smartphone-features. Accessed 10 Feb 2021

Rua R (2022) Online appendix. https://sites.google.com/view/perf-guidelining-appendix. Accessed 22 Nov 2022

Rua R, Couto M, Pinto A, Cunha J, Saraiva J (2019) Towards using memoization for saving energy in android. In: Marín B, Brito IS, Mora MK, Malucelli A, Serral E, Giachetti G, Araújo J, Goulão M, Ayala CP, Genero M, Souza VS (eds) Proceedings of the XXII Iberoamerican conference on software engineering, CIbSE 2019, La Habana, Cuba, April 22–26, 2019, pp 279–292. Curran Associates

Rua R, Couto M, Saraiva J (2019) Greensource: a large-scale collection of android code, tests and energy metrics. In: 2019 IEEE/ACM 16th international conference on mining software repositories (MSR), pp 176–180. https://doi.org/10.1109/MSR.2019.00035

Rua R, Fraga T, Couto M, Saraiva JA (2020) Greenspecting android virtual keyboards. In: Proceedings of the IEEE/ACM 7th international conference on mobile software engineering and systems, MOBILESoft '20. Association for Computing Machinery, New York, NY, USA, pp 98–108. https://doi.org/10.1145/3387905.3388600

Rua R, Saraiva JA (2022) E-manafa: energy monitoring and analysis tool for android. ASE22. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3551349.3561342

Sadeghi A, Jabbarvand R, Malek S (2017) Patdroid: permission-aware GUI testing of android. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, ESEC/FSE 2017. Association for Computing Machinery, New York, NY, USA, pp 220–232. https://doi.org/10.1145/3106237.3106250

Sahin C, Tornquist P, Mckenna R, Pearson Z, Clause J (2014) How does code obfuscation impact energy usage? In: 2014 IEEE international conference on software maintenance and evolution, pp 131–140. https://doi.org/10.1109/ICSME.2014.35

Scalabrino S, Bavota G, Linares-Vásquez M, Piantadosi V, Lanza M, Oliveto R (2020) API compatibility issues in android: causes and effectiveness of data-driven detection techniques. Empir Softw Eng 25:5006–5046. https://doi.org/10.1007/s10664-020-09877-w

Schlachter F (2013) No Moore's law for batteries. Proc Natl Acad Sci 110(14):5273–5273. https://doi.org/10.1073/pnas.1302988110. https://www.pnas.org/content/110/14/5273

Shapiro SS, Wilk MB (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3–4):591–611. https://doi.org/10.1093/biomet/52.3-4.591

Spearman C (1904) The proof and measurement of association between two things. Am J Psychol 15(1):72–101. http://www.jstor.org/stable/1412159

Theis TN, Wong HP (2017) The end of Moore's law: a new beginning for information technology. Comput Sci Eng 19(2):41–50. https://doi.org/10.1109/MCSE.2017.29

Thorwart A, O'Neill D (2017) Camera and battery features continue to drive consumer satisfaction of smartphones in US. https://www.prnewswire.com/news-releases/camera-and-battery-features-continue-to-drive-consumer-satisfaction-of-smartphones-in-us-300466220.html. Accessed 10 Feb 2021

Vilkomir S, Amstutz B (2014) Using combinatorial approaches for testing mobile applications. In: 2014 IEEE seventh international conference on software testing, verification and validation workshops, pp 78–83. https://doi.org/10.1109/ICSTW.2014.9

Yamane T (1973) Statistics. an introductory analysis, 3rd edn. Harper International edition. Harper & Row. https://books.google.co.jp/books?id=sl75MgEACAAJ

Zeng Y, Chen J, Shang W, Chen THP (2019) Studying the characteristics of logging practices in mobile apps: a case study on f-droid. Empir Softw Eng 24. https://doi.org/10.1007/s10664-019-09687-9