Noname manuscript No.
(will be inserted by the editor)

Investigating Types and Survivability of Performance Bugs in Mobile Apps

Alejandro Mazuera-Rozo · Catia Trubiani · Mario Linares-Vásquez · Gabriele Bavota

Received: date / Accepted: date

Abstract A recent research showed that mobile apps represent nowadays 75% of the whole usage of mobile devices. This means that the mobile user experience, while tied to many factors (*e.g.,* hardware device, connection speed, etc.), strongly depends on the quality of the apps being used. With "quality" here we do not simply refer to the features offered by the app, but also to its non-functional characteristics, such as security, reliability, and performance. This latter is particularly important considering the limited hardware resources (*e.g.,* memory) mobile apps can exploit. In this paper, we present the largest study at date investigating performance bugs in mobile apps. In particular, we (i) define a taxonomy of the types of performance bugs affecting Android and iOS apps; and (ii) study the survivability of performance bugs (*i.e.,* the number of days between the bug introduction and its fixing). Our findings aim to help researchers and apps developers in building performance-bugs detection tools and focusing their verification and validation activities on the most frequent types of performance bugs.

Keywords Performance bugs · Mobile apps · Empirical study

Alejandro Mazuera-Rozo
Università della Svizzera italiana, Switzerland
E-mail: alejandro.mazuera.rozo@usi.ch

Catia Trubiani
Gran Sasso Science Institute, Italy
E-mail: catia.trubiani@gssi.it

Mario Linares-Vásquez
Universidad de los Andes, Colombia
E-mail: m.linaresv@uniandes.edu.co

Gabriele Bavota
Università della Svizzera italiana, Switzerland
E-mail: gabriele.bavota@usi.ch

1 Introduction

Over the last decades, research has highlighted the importance of integrating quality analysis in the software development process [2,26]. Software quality can be defined as the discipline representing the entire collection of engineering activities used throughout the software development cycle and directed to meet quality requirements [37]. Together with others (*e.g.,* reliability, security, etc.), performance has been recognized as an essential quality attribute of every software system. Indeed, if performance targets are not met, a variety of negative consequences (such as damaged customer relations, business failures, lost income, etc.) can impact on a significant fraction of projects [33,42].

Performance problems have been studied from several decades in literature, and software performance engineering emerged as the discipline focused on fostering the specification of performance-related factors [95,8,94] and reporting experiences related their management [81,41,78,3]. Performance bugs, *i.e.,* suboptimal implementation choices that create significant performance degradation, have been demonstrated to hurt the satisfaction of end-users in the context of desktop applications [67]. These bugs, that are pervasive and difficult to understand, can cause delays, failures on deployment, redesigns, even a new implementation of the system or abandonment of projects, which lead to significant costs [93,58]. As discussed in [97], a good understanding of the impact of different types of non-functional bugs (such as performance, but also security, reliability, etc.) on various project aspects is essential to improve software quality research and practice.

When talking about mobile applications (apps), performance bugs also present some additional requirements that distinguish them from traditional software applications [91,35]. For instance, the design and the development is affected by the different hosting devices, operating systems, and even by different versions of the same operating system [14]. Besides this, an app may inadvertently make extensive use of battery resources, and the excessive power consumption may prevent the actual usage of the app itself. To prevent this, it is fundamental to deeply study the non-functional characteristics of mobile apps [15].

The goal of this paper is to investigate performance bugs occurring in native mobile apps. These specific bugs are an important non-functional type of bugs, since they are strictly related with other types. For instance, the raising of security or reliability requirements usually leads to performance degradation. Indeed, in previous work [79] we experimented that security has a direct overhead on performance. This means that the introduction of security mechanisms inevitably consumes system resources affecting the system performance, even compromising its full operability. Besides this, it has been demonstrated that performance bugs are very commonly occurring in real-world projects [33].

Additional motivation for investigating performance bugs in mobile apps has also been recently pointed out in the literature [59,55]. In fact, lower-rated apps resulted to show a relationship with bad programming practices and poor

performance, suggesting that code quality can actually have an impact on the success of mobile apps [38]. In [64] future trends in software engineering research for mobile apps are discussed, and performance problems are recognized to span in a wide variety requiring the development of tools and approaches looking more at the client-side [84].

To the best of our knowledge, there are few studies investigating performance bugs in mobile apps [57,27,9,55]. Hecht *et al.* [27] study the impact of code smells on performance metrics, while Cruz and Abreu [9] investigate whether the usage of performance best-practices help in saving battery life. Linares-Vásquez *et al.* [55] surveyed practitioners regarding their practices for detecting and fixing performance bottlenecks in Android apps. Liu *et al.* [57] have been the first to run an empirical study on performance bugs in the specific context of mobile apps. They studied 70 real-world performance bugs mined from the issue tracker of eight Android apps. Starting from this data, they manually classified the 70 bugs into three categories, namely: *GUI lagging* (53 instances), *Energy leak* (10 instances), and *Memory bloat* (8 instances). They also studied (i) the way in which the performance bugs manifest, showing that one third of them require a specific user interaction to manifest, and (ii) the bug-fixing effort spent to close the subject issues, using proxies such as the number of days for which the bug was open, the number of comments posted in the issue discussion, and the patch size. Finally, they identified three common performance bug patterns (*i.e.,* lengthy operations in main threads, wasted computation for invisible GUI, and frequently-invoked heavy-weight callbacks) and implemented a tool to identify two of them.

Our work stems from the seminal paper by Liu *et al.* [57] and aims at expanding the empirical knowledge about performance bugs in mobile apps. Our main contributions, as compared to [57], can be summarized as follows:

1. *We present a larger study on the types of performance bugs affecting not only Android but also iOS apps.* We manually analyze 500 commits (250 related to Android and 250 to iOS apps) aimed at fixing real performance bugs with the goal of categorizing the type of bug being fixed (*e.g.,* memory leak). Based on this classification, we created two taxonomies of performance bugs for Android and iOS apps. As compared to [57] we: (i) analyze a much larger sample of performance bugs (70 *vs* 250+250); (ii) investigate the types of performance bugs affecting both Android and iOS apps; and (iii) while confirming *GUI lagging*, *Energy leak*, and *Memory bloat* as frequent types of performance bugs, we identified additional categories of bugs not covered in [57]. Knowing the types of performance bugs that most likely affect mobile apps can help to guide (i) apps developers, in focusing code inspection/reviewing activities toward the identification of the most frequently reported types of performance bugs, (ii) researchers, in investing in the development of detection tools targeting the most diffused types of performance bugs, and (iii) language/API developers, to design/improve mechanisms for promoting the development of efficient code.

2. *Bug-fixing effort vs survivability of performance bugs.* As previously said, Liu *et al.* [57] analyzed the bug-fixing effort for the 70 issues representing performance bugs. Since our work focuses on commits fixing performance bugs, we have the chance to study the survivability of these bugs (*i.e.,* the time going from their introduction to their fixing). We found that, on average, they survive for at least 90 days in both Android and iOS apps. Moreover, we observed that, when compared to bugs unrelated to performance issues, performance-related bugs tend to survive longer in the apps.

Paper structure. Section 2 presents background information on performance problems and discusses the related work mostly focusing on performance bugs. Section 3 describes the design of the empirical study: data extraction and analysis are illustrated. Results and main findings are discussed in Section 4, while threats to validity are discussed in Section 5. Section 6 concludes the paper and provides future research directions.

2 Background and Related work

In this section, we introduce some foundational concepts related to performance problems and discuss the related literature. Although this paper focuses on performance bugs of Android and iOS apps, works related to other types of bugs and apps are also reviewed to provide a wider overview of current research trends.

2.1 Background: Performance-related concepts

Hereafter we provide the definitions of the most common performance issues that may arise when evaluating software systems. These specifications are later used to describe and categorize the performance bugs identified in the mobile apps subject of our study (see Section 4.1).

The most common performance indicators are listed in the following:

- *Service latency*, defined as the time interval between a user request of a service and the response of the software system. It can be measured in microseconds, seconds, minutes, etc. Usually, upper bounds are defined as "business" requirements by the end users to denote the maximum time frame users are supposed to wait before aborting the request [32].
- *System throughput*, defined as the rate at which requests can be handled by a software system and measured in requests per unit of time. Throughput requirements can be both "business" and "system" requirements, depending on the target domain. This means that such requirements can represent either an upper or a lower bound, *e.g.,* a hardware machine should serve at least 10 requests/sec but a web service should not send more than 100 requests/sec otherwise the system becomes congested [32].

- *Resource utilization*, defined as the ratio of busy time of a resource (*e.g.,* the RAM) and the total elapsed time of the measurement period. Usually, upper bounds are defined in "system" requirements by system engineers on the basis of their experience, scalability issues, or constraints introduced by other concurrent software systems sharing the same resource [32].
- *Energy consumption*, defined as the amount of energy or power used to accomplish some tasks; it can be measured as watts per unit of time. This strongly depends on the types of operations that are executed on a machine. Similarly to resource utilization, upper bounds are defined to limit the types of operations and avoid a high energy consumption [19].
- *Communication and bandwidth*, defined as the communication bit rate and indicating the maximum throughput of a logical or physical communication path in a digital communication system. Bandwidth tests are aimed to measure the computer network by reporting on the maximum rate of data transfer, typically measured in bits per second (bit/s). Similarly to energy consumption, upper bounds are defined to report on the channel capacity and avoid loss of information in the communication [19].
- *Compression ratio*, defined as the power of reducing the size of data; it is a measurement of the relative decrease in size of data representation produced by a compression algorithm. It is typically expressed as the division of uncompressed size by compressed size. The compression mechanism supports the efficiency of communications since data itself is not reduced, there is a diminution on its size only [19].
- *Scalability*, defined as the property of a system to handle a growing amount of load by adding resources to the system, so that no performance variation is perceived by the end-users. To this end, vertical and horizontal scaling are used: The former varies the resource sharing on each machine, whereas the latter aims to employ multiple physical/virtual machines. The combination of these two techniques efficiently manages the load distribution and resulted to be beneficial in the process of continuously fulfilling performance goals [31].
- *Availability*, defined as the probability that a software system works when required during a certain time frame. The availability of a hardware/software module can be obtained as the Mean Time Between Failures (MTBF) over the same metric plus the Mean Time To Repair (MTTR), *a.k.a.,* Mean Down Time (MDT). Usually, lower bounds are defined in "system" requirements by system engineers on the basis of their experience, to denote the expectations of end-users in completing their requests when required [86].

Note that all of the above-defined performance indicators are relevant for mobile apps either on the client or on the server side. Indeed, several mobile apps rely on server-side services (*e.g.,* for online gaming) that are subject to all classic performance issues of software systems. In the context of this work, we define performance bugs as all suboptimal implementation choices that can negatively impact one of the above listed performance indicators. When discussing the taxonomy of performance bugs we identified in Android and iOS

apps, we will explicitly link the different types of bugs to the corresponding performance indicators they impact.

In the next section we discuss the related work dealing with performance bugs. To better compare the different studies in the literature, we also specify the performance indicators they are related to (see the last column in Tables 1 and 2).

2.2 Related work on Performance Bugs

Performance bugs have been widely investigated in empirical studies, not only for desktop applications, but also for mobile apps. In [66] the detection of performance bugs has been performed by looking at non-intrusive fixes only, *i.e.,* when a condition becomes true during loop execution, then just break out of the loop. This technique has been applied to Java and C/C++ applications and managed to detect previously unreported performance bugs.

Jin *et al.* [33] provide a starting point for understanding real-world performance bugs, in particular roughly one hundred bugs are collected from five software repositories. This study is aimed to highlight the most promising research directions in this context, that are: (i) guidance for performance testing; (ii) support for bug detection and avoidance; (iii) comparison with functional bugs. However, the categorization of root causes of performance bugs is limited to three types, *i.e.,* uncoordinated and skippable functions, synchronization issues.

Another example of an empirical study on performance bugs is the work by Zaman *et al.*[98]; in this study performance bugs found in Mozilla Firefox and Google Chrome were analyzed to learn how project members collaborate to detect and fix these bugs. Four main points are outlined: (i) techniques should be developed to improve the reproducibility of bugs; (ii) more optimized means to identify the root cause of performance bugs should be developed; (iii) collaborative root cause analysis process should be better supported; and (iv) the impact of changes on performance should be analyzed, *e.g.,* by linking automated performance test results to commits, thus tracing software changes and performance characteristics.

Nistor *et al.* [67] present an empirical study on three popular code bases (Eclipse JDT, Eclipse SWT, and Mozilla) with the goal of investigating how performance and non-performance bugs are discovered, reported and fixed by developers. Three main findings are outlined: (i) fixing performance bugs may introduce new functional bugs, similarly to fixing non-performance bugs; (ii) fixing performance bugs is more difficult than fixing non-performance bugs; (iii) unlike non-performance bugs, many performance bugs are found by code reasoning and profiling, not through direct observation of the bug's negative effects. Olivo *et al.* [70] also investigate performance bugs, specifically traversal bugs that arise if a program fragment repeatedly iterates over a data structure, such as an array or list, that has not been modified between successive traversals. Such performance bugs are typically easy to fix and often only require the

Table 1 Summary of related work - Performance bugs.

Bugs type	*Approach*	*Pros*	*Scope*	*Performance factors*
Performance	Nistor et al. [66]	Detection of performance problems in Java and C/C++ applications	Limited to non-intrusive fixes on conditional loops	service latency
	G. Jin et al. [33]	Diagnosis of performance bugs and their relationship with functional correctness	Limited categorization of root causes of performance bugs	communication and bandwidth
	Zaman et al. [98]	Analysis of the collaboration among project members to detect and fix performance bugs	Limited to browsers (i.e., Mozilla Firefox and Google Chrome) performance issues	service latency
	Nistor et al. [67]	Performance bugs are demonstrated to be more difficult than functional bugs	Bugs are found through code reasoning, not by the direct observation of profiling data	system throughput
	Liu et al. [57]	Empirical study of performance bugs from smartphone applications	Limited to Android applications	service latency, system throughput, resource utilization
	Hecht et al. [27]	Empirical study on the impact of code smells for performance metrics	Limited to Android applications	service latency, system throughput
	Cruz et al. [9]	Empirical study on the impact of performance best practices on the energy consuption	Limited to Android applications	energy consumption
	Olivo et al. [70]	Static detection of performance bugs in collection of redundant traversals	Limited to data structures wrongly used	compression ratio
	Jovic et al. [36]	Look for causes of long latency performance bugs	Limited to Java applications	service latency
	Killian et al. [39]	Detection of performance bugs in distributed systems	Network delays are simulated and can hide some software specific bugs	communication and bandwidth
	Parsons et al. [72]	Detection of performance antipatterns in real enterprise applications	Limited to parse the run-time design model whose accuracy relies on the monitoring data	system throughput
	Grechanik et al. [18]	Anticipate performance bottlenecks by observing execution traces and deriving load testing scripts	Limited to running applications where it is possible to retrieve execution traces	service latency, system throughput
	Wert et al. [92]	Detection of performance problems by means of systematic experiments	Limited to specific heuristics that target three tier enterprise applications only	resource utilization
	Yang et al. [96]	Empirical investigation on performance bugs and fixes related to GPUs	Limited to a specific hardware setting	scalability
	Selakovic et al. [82]	Empirical investigation on performance issues arising in JavaScript applications	Limited to JavaScript bugs	service latency
	Linares-Vásquez et al. [55]	Empirical study on how developers deal with performance bugs in mobile applications	Limited to Android apps and no tool support for detecting performance bottlenecks	energy consumption, resource utilization, system throughput, communications and bandwith, service latency

Table 2 Summary of related work - Performance and other non-functional bugs.

Bugs type	*Approach*	*Pros*	*Scope*	*Performance factors*
Performance and other Non-functional	Carroll et al. [4]	Analysis of energy usage and battery lifetime of the device's processor for different patterns	Limited to a relatively dated mobile phone and its specific system architecture	energy consumption
	Linares-Vásquez et al. [56]	Analysis of the impact of micro-optimizations on resources consumption in Android apps	Limited to micro-optimizations practices in Android apps	system throughput, resource utilization
	Cruz et al. [10]	Catalog of the design patterns related to the energy efficiency of mobile apps	Impact of applying the design patterns is neither automated or assessed	energy consumption
	Zaman et al. [97]	Empirical investigation on the Firefox project and its performance and security bugs	The comparison does not include the correlations between these two types of bugs	service latency, system throughput
	Linares-Vásquez et al. [52]	Empirical investigation for the energy consumption of mobile applications	Developers typically do not select the best energy-efficient options	energy consumption
	Gegick et al. [17]	Analysis of security bugs by means of a statistical model recognizing the actual bugs	Limited to bugs that are expressed as part of the training for the statistical model	—
	Zhou et al. [99]	Identification of security issues from commit messages and bug reports	Affected by the training dataset since the approach builds upon ML techniques	—
	Mazuera-Rozo et al. [60]	Empirical investigation on vulnerabilities of android applications	No support for the detection of security vulnerabilities	—
	Oliveira et al. [69]	A study on the energy consumption of Android app development approaches	Limited support on the combination of the different approaches for energy savings	energy consumption
	Near et al. [65]	A catalog of access control patterns is provided to detect security bugs	The detection of previously unknown bugs depends from the matching of patterns	—

addition of a parameter to a method, the addition of a field to an object, or the use of a slightly different data structure.

Jovic *et al.* [36] recognize performance bugs to be essential in post-deployment detection approaches since they are particularly sensitive to the context, *i.e.,* they may escape detection in a testing lab. A tool is also developed to find the causes of long latency by means of call stack sampling. This work focuses on Java applications.

Other types of applications have been also studied. Yang *et al.* [96] present an empirical study on general purpose graphics processing units (GPGPU) programs. This investigation pointed out various performance bugs. For some

Table 3 Summary of related work - Generic bugs.

Bugs type	***Approach***	***Pros***	***Scope***
Generic	Gao et al. [16]	Prevention of JavaScript bugs through Flow and TypeScript	Limited to JavaScript projects
	Wang et al. [90]	Analysis and characterization of concurrency bugs in Node.js	Limited to bugs occurring on Node.js, no portability is demonstrated for other server setting
	Di Franco et al. [12]	Study of numerical bug characteristics from well known scientific libraries	The fixing of numerical bugs can be partially automated through program analysis techniques
	Panichella et al. [71]	Empirical evaluation on the automatic derivation of test summaries in JUnit tests	The perceived test comprehensibility relies on perception of participants
	Wan et al. [89]	Empirical study on bug characteristics of blockchain systems	Limited to classify the bugs, no support to detect and fix bugs
	Lee et al. [43]	Bug triage system based on deep learning and word embedding techniques	limited to retrieve the appropriate developer for a bug report, no characterization of bugs
	Linares-Vásquez et al. [56]	Analysis of the impact of micro-optimizations on resources consumption in Android apps	Limited to micro-optimizations practices in Android apps
	Mondal et al. [63]	Empirical study on bug-propagation through code cloning	Limited to Java open source projects showing code cloning patterns

of the identified bugs, fixes have been proposed and experimented. However, both performance bugs and related fixes are specific to the target hardware environment, *i.e.,* GPU and their portability on different hardware settings is not considered. Selakovic and Pradel [82] conduct an empirical study on performance issues arising from popular client and server JavaScript projects. Root causes of performance issues are identified and inefficient usage of APIs results to be the most prevalent cause. They report that most of the issues can be fixed through the modification of few lines of code.

In the particular case of Android apps, there is a previous work analyzing different performance factors. For instance, Liu *et al.*[57] conducted an empirical study on performance bugs from real-world Android applications. This study has been discussed and compared with our work in Section 1. Hecht *et al.* [27] present an empirical study on the performance impact of Android code smells. By performing series of experiments, authors demonstrated that the correction of code smells has a significant impact on performance metrics.

Linares-Vásquez *et al.* [55] focus on understanding actual developers' practices for detecting and fixing performance bottlenecks in mobile apps. This study points out that developers heavily rely on user reviews and manual execution of the apps for detecting performance bugs. There are some tools able to detect performance bottlenecks, but these tools are mostly for profiling/debugging and do not support the automatic detection and the refactoring

of performance issues. Linares-Vásquez *et al.* [56] also analyze the impact of micro-optimizations, in particular on the memory and CPU consumption of Android apps. Their results suggest that (i) implementing micro-optimizations is not a common practice in Android open source projects; (ii) only one micro-optimization (removing unused resources) has significant impact on the memory consumption of Android apps; and (iii) the impact of micro-optimizations is noticeable under certain load conditions that might appear only on specific types of apps. In addition, Cruz *et al.* [9] study whether performance-based practices for Android apps help in saving energy, finding that energy-aware practices can save up to one hour of battery life.

In addition to empirical studies, other previous works have been devoted to the detection of performance bugs. In [39] a technique that finds performance bugs in distributed systems is proposed. It explores a large number of executions looking for the ones that perform worse than usual, thus driving the debugging activity towards the identification of performance issues.

Parsons and Murphy [72] propose an approach for the automatic detection of performance issues through antipatterns. This approach has been applied to real enterprise applications and monitoring information has been collected from the systems under load. An antipattern-based rule engine has been built to take as input the reconstructed run-time design model and produce as output the detected antipatterns. More recently, Trubiani *et al.* [87] detected performance antipatterns by exploiting load testing and profiling data. Performance characteristics from the system under test are collected using a profiler tool that creates snapshots, thus identifying performance hotspots.

Grechanik *et al.* [18] present an approach to detect performance bottlenecks in software systems by means of feedback-directed systematic experimentation. It analyzes execution traces to generate test scripts which are likely to provoke computationally intensive executions of the system under test, thus allowing to proactively anticipate performance bottlenecks of the software. This approach only works after deployment since rules are extracted from execution traces describing the relation between input data and workloads.

Systematic experiments have also been used by Wert *et al.* [92] to detect performance antipatterns in Java-based three-tier applications. The authors expose the system under test to varying workloads and observe changes to specific runtime metrics, thus to detect occurrences of performance antipatterns using a decision tree. However, the nature of performance problems is specific to source code or resources, hence the proposed heuristics fail when performance issues arise over various places in the source code.

Only a few works focused on improving performance of Android apps. Liu *et al.* [57] propose a static analysis-based approach for detecting performance bugs in Android apps, in particular GUI lagging, energy leaks, and memory bloats. Guo *et al.* [22] present an Android specific approach for detecting resource leaks (also using static analysis), but focused on resource leaks related to operations with hardware-components such as sensors, camera, and speakers.

Lin *et al.* [51] perform a study on Android apps to investigate threading issues. This study provides evidence that even though half of the apps use

`AsyncTask`, there is a huge number of places where long-running operations are not encapsulated in `AsyncTask`.

2.3 Related work on Non-Functional Bugs

In addition to performance bugs, there are other non-functional bugs that have been investigated. Zaman *et al.* [97] present an empirical study on how performance and security bugs differ from each other and from other types of bugs in the Firefox project. Main findings are: (i) security bugs are fixed and triaged much faster, but are reopened and tossed more frequently; (ii) performance bugs require more experienced bug fixers than other bugs and affect more files than security bugs.

Energy bugs in mobile apps have been widely investigated in the recent years [10,76,73,74,75,52,23,46,24,47,45,88,50,44,21,80,48]. One of the seminal works in this direction has been presented by Carroll and Heiser [4], who performed a detailed analysis of the energy consumption of a smartphone. Specifically, they collected energy measurements of a physical device to demonstrate how the different components of the device contribute to overall power consumption. Moreover, they considered different usage scenarios to develop a model predicting the overall energy consumption and battery life of a mobile device and its system architecture. Cruz *et al.* [10] present a catalog of 22 design patterns that contribute to improve the energy efficiency of mobile apps, but their impact is not assessed. Linares-Vásquez *et al.* [52] analyze the energy consumption of APIs used in Android apps. Actual measurements are collected by using a power meter and API calls and patterns are traced onto the source code. Moreover, approaches for improving energy consumption of mobile apps have focused on different aspect such as HTTP requests [48] and OLED displays [88,49,54,50,53].

Oliveira *et al.* [69] study the impact of the most popular development approaches on the energy consumption of Android apps. Experimental results compare the energy efficiency and performance of the most commonly used approaches to develop apps on Android. Specifically, JavaScript was reported as being the more energy-efficient in most of the benchmarks, but both Java and C++ outperformed JavaScript in some cases. Leveraging a combination of approaches is indicated as leading to non-negligible improvements in energy-efficiency and performance.

Security bugs are included in the non-functional bugs list. There are many works dealing with security bugs and we briefly describe some of them here. Gegick *et al.* [17] present an approach to recognize security bugs; based on text mining of natural-language descriptions of bug reports, a statistical model (trained on already manually-labeled bugs) is used to identify security bugs that are manually-mislabeled as not-security bugs.

Zhou and Sharma [99] present an automatic vulnerability identification system to flag vulnerability-related commits and bug reports using machine learning techniques. This way, a wide range of security issues can be identified

from commit messages and bug reports. However, machine learning strongly depends on the expressiveness of the dataset and precision and recall may substantially vary.

Near and Jackson [65] present a technique for finding security bugs by matching extracted access patterns to known safe patterns. An initial catalog of common access control patterns, based on the analysis of real-world applications, is provided and used to find previously unknown bugs in open-source applications.

Recently, Mazuera-Rozo *et al.* [60] investigate vulnerabilities in the Android operating system from different perspectives: (i) most common vulnerabilities, (ii) layers and subsystems impacted by the vulnerabilities, and (iii) survivability of the vulnerabilities.

Noei *et al.* [68] present an empirical study aimed at understanding how Android applications' characteristics (such as code size and UI complexity) and devices' attributes (such as the CPU and the display size) affect the quality of Android apps, as perceived by users. They found that both app and device attributes play a significant role in the user-perceived quality. Syer *et al.* [85] report an exploratory study to compare mobile and server applications along two dimensions: the size of the code base, and the time to fix defects. Reported findings show that: (i) mobile apps are smaller than server ones in terms of the size of the code base and the development team; (ii) fewer users report defects and defects are fixed in roughly one month. This study does focus on performance-related issues. Mcilroy *et al.* [62] study the characteristics of user reviews in the Google Play and in the Apple App stores. They found that downloads and releases of apps are correlated with reviews, whereas app category is less relevant for users. Authors pointed out that many analytics tools are mostly sales-oriented rather than software-quality-oriented.

Other non-functional bugs in mobile apps like behavioral consistency are gaining more attention from researchers [13,34,1]. However, we do not discuss these papers since not strongly related to our research questions.

2.4 Summary of related work

Tables 1, 2, 3 schematically report the related work dealing with performance, non-functional and generic bugs, respectively. First column shows the bugs type, second column lists the different approaches, third and fourth columns report pros and scope of these approaches, respectively. Tables 1 and 2 additionally include a fifth column where there is a match of related work with the performance factors we consider in this paper (see Section 2.1). This literature review is restricted to the papers we found more relevant when compared to our approach, and it is far from being exhaustive.

To the best of our knowledge, our study on performance bugs in mobile apps is (i) the largest conducted in the literature in terms of manually analyzed performance bugs (500); (ii) the only one providing a detailed taxonomy of performance bugs affecting both Android and iOS apps; (iii) the first to

investigate the survivability of apps' performance bugs, and to compare those with other types of bugs affecting mobile apps.

3 Study Design

The *goal* of the study is to investigate performance bugs affecting native mobile apps, and in particular Android and iOS apps. The *purpose* is to (i) define a taxonomy highlighting the types of performance bugs affecting mobile apps, and (ii) investigate the time needed to fix performance bugs. The *context* consists of 500 commits performed by software developers of mobile apps to fix performance bugs in Android and iOS apps (250 commits per operating system, coming from 47 Android apps and 31 iOS apps). The data used in the study are available in our online appendix [61].
The study addresses the following research questions:

RQ_1: *Which types of performance bugs affect mobile apps?*

This research question aims at identifying the types (*e.g.,* inefficient synchronization among threads) of performance bugs affecting mobile apps. We consider both Android and iOS apps. Knowing the types of performance bugs affecting mobile apps can help to guide (i) apps developers, in focusing code inspection/reviewing activities toward the identification of the most frequently reported types of performance bug, (ii) researchers, in investing in the development of detection tools targeting the most diffused types of performance bugs, and (iii) language/API developers, to design/improve mechanisms for promoting the development of efficient code.

RQ_2: *How long does it take to fix performance bugs in mobile apps?*

This research question studies the survivability of the performance bugs subject of our study. In particular, we assess the number of days between the bug introduction and its fixing. We also compare the survivability of (i) different types of performance bugs (classified as output of RQ_1), and (ii) performance bugs versus other types of bugs unrelated to performance. RQ_2's findings support both developers and researchers in assessing the usefulness of detection tools able to immediately catch an introduced performance bug (*i.e.,* a long survivability of the bugs would indicate the urge for such tools).

3.1 Study Context

To answer RQ_1 and RQ_2 we started by collecting a set of open source mobile apps available at GitHub. The availability of the versioning system was a needed selection criterion given our goal of (i) investigating bug-fixing commits to categorize performance bugs affecting mobile apps (RQ_1), and (ii) study the survivability of performance bugs (RQ_2).

We started by randomly selecting 100 Android and 100 iOS apps hosted on GitHub. The 100 Android apps were extracted from the `open − source − android − apps` project[1], *i.e.,* a collection of open source Android apps, while the 100 iOS apps were collected from the similar `open − source − ios − apps` project[2]. Once collected these 200 apps, we manually inspected all of them to verify that they actually were mobile apps. We found 15 repositories of software projects not related to Android/iOS apps and 1 duplicated app (*i.e.,* the same app contained twice in the automatically crawled lists of apps). We replaced these apps with 16 well-known mobile apps having their code repository available (*e.g.,* wikipedia, wordpress). The complete list of considered 200 apps is available in our online appendix [61]. We used a crawler to analyze their change history and identify commits aimed at fixing performance bugs. To do that, we used a keywords-matching mechanism on the commit notes/messages, looking for commit notes/messages reporting one of the following terms:

performance, slow, latency, throughput, lagging, leak, suboptimal, bloat, utilization, ANR, lag, OOM, bottleneck, hot-spot, hot spot, energy greedy, drain, optimization, wakelock, lengthy, optimize, consumption

While such a list might be non-comprehensive and also result in the identification of false positives (*i.e.,* commits having one of the matched terms in the commit note but not actually fixing a performance bug), this does not represent a strong limitation for our study since the keywords-matching mechanism is only used to pre-filter candidate commits, that will then be manually analyzed (as described later in this section) and discarded if considered false positives. By using this process, we extracted 1,396 candidate commits: 1,016 related to Android apps and 380 to iOS apps. In total, these commits have been found in 78 apps (47 Android apps and 31 iOS apps). Tables 4 and 5 report the characteristics of the considered Android and iOS apps, respectively. These apps represent the study context for both RQ_1 and RQ_2. The only exception is represented by the 9 apps marked with *, which are not considered in RQ_2 since, as explained later, they do not provide data useful for answering it.

For Android, 41 out of the 47 apps (87%) are published at the Google Play store, while for iOS 18 out of the 31 apps (58%) are published at the iOS app store. The considered apps cover a set of 13 different categories in Android and 10 in iOS[3]. Also, the subject projects include apps covering a wide size range (going from 1,597 to 1,526,886 LOCs, average: 98,718.4), having a different length of the change history both in terms of time (from 24 to 221 months, average: 78.1) as well as in commits extracted from their versioning systems (from 42 to 70,002, average: 5,057.4). We cannot claim that this set of apps is representative of the whole population of apps developed for the two mobile platforms, and we acknowledge this as an internal threat to the validity of

[1] `https://github.com/pcqpcq/open-source-android-apps`

[2] `https://github.com/dkhamsing/open-source-ios-apps`

[3] For apps not published on the app stores we manually assigned the category by reading their description.

our study. However, the heterogeneity of the considered apps may foster the generalization of our findings.

3.2 RQ_1: Data Collection & Analysis

The collected 500 commits were manually analyzed by the four authors with the goal of assigning to each of them a "tag" describing the reasons behind the performance bug fixed in the analyzed commit. We targeted the tagging of 500 commits (250 for each platform), representing a statistically significant sample with 99% confidence level ± 5%. Note that we targeted 500 manually tagged commits excluding those that we classified (tagged) as *false positives* (*i.e.,* commits not really aimed at fixing performance bugs).

The tagging process was supported by a web application that we developed to classify the commits and to solve conflicts between the authors. Each author independently tagged the commits randomly assigned to her/him by the web application, defining a "tag" describing the cause behind each analyzed bug (*e.g.,* object declaration inside a loop). To define such a tag the authors manually inspected the diff of the commit and the commit note accompanying it.

Every time the authors had to tag a commit, the web application also showed the list of tags created so far, allowing the tagger to select one of the already defined tags. In a context like the one encountered in this work, where the number of possible tags (*i.e.,* cause behind the bug) is extremely high, such a choice helps using consistent naming while not introducing a substantial bias.

Each commit was assigned to two authors by the web application and, in cases for which there was no agreement between the two evaluators (*i.e.,* different tags assigned by the two authors), the document was automatically assigned to a third evaluator. If, after the third evaluation, there was still no majority of the evaluators with the same tag, the conflicts were solved through an open discussion aimed at defining the most appropriate tag to assign.

To reach our goal of 500 commits tagged, a total of 1,780 manually assigned tags was required. In particular:

- *A total of 1,010 commits were tagged*: 510 were classified as false positives.
- *Conflicts were arisen for 403 cases (40%)*: in 240 of these cases, the third evaluator automatically added by the web application was enough to solve the conflict and reach a majority (this accounts for $240 \times 3 = 720$ tags); in 163 cases an open discussion was needed, and this resulted in the addition of a fourth tag to solve the conflict (accounting for $163 \times 4 = 652$ tags). Finally, the 204 commits for which there was immediate agreement accounted for $204 \times 2 = 408$ tags.

An example of conflict that was solved through the addition of the third evaluator concerns the commit `58273a4` from the `wordpress-mobile/WordPress--Android` project. The commit note reports a quite general "*date formatting performance tweaks*". The first evaluator, looking at the code diff, assigned as tag "*Objects instantiation in loop*". Indeed, the creation of a `Date` and of a

Table 4 Characteristics of the considered Android apps.

Name	Age(Months)	Language	Files	LOC	Commits	Contributors	Size	On store	Category
AFWall+	83	Java	300	46,373	1,416	18	24,929	Yes	Tools
Amaze File Manager	68	Java	537	60,788	3,490	91	44,978	Yes	Tools
AntennaPod	93	Java	556	74,229	4,501	90	31,201	Yes	Photo and Video
Antox	68	Scala	362	22,358	2,640	64	25,811	Yes	Communication
Better Battery Stats	33	Java	289	30,578	206	7	54,417	Yes	Tools
Bitcoin Wallet	102	Java	379	36,668	3,199	26	32,405	Yes	Books and Reference
Cgeo	98	Java	1,586	171,197	11,199	90	300,719	Yes	Entertainment
ConnectBot	221	Java	317	77,496	1,702	42	6,997	Yes	Communication
Conversations	68	Java	466	76,346	5,225	100	28,273	Yes	Communication
DeviceControl	69	Java	466	35,386	1,042	11	16,446	No	Productivity
DuckDuckGo	32	Kotlin	446	39,818	529	11	14,592	Yes	Tools
Etar	131	Java	392	63,623	5,593	70	33,206	Yes	Productivity
FBReaderJ	143	Java	1,525	233,523	9,034	38	64,544	Yes	Books and Reference
K-9 Mail	131	Java	1,222	157,089	8,045	191	58,834	Yes	Communication
Kontalk Messenger	100	Java	569	66,921	6,463	43	56,596	Yes	Communication
Lightning Browser	79	Kotlin	301	22,309	1,818	69	13,203	Yes	Communication
Link Bubble	77	Java	258	34,462	2,372	8	58,729	Yes	Communication
Mapbox Demo	67	Java	327	27,443	655	22	117,920	Yes	Libraries and Demo
MDX	41	Java	945	15,255	58	2	698	No	Tools
Mirakel	113	Java	750	68,873	3,722	9	106,267	No	Productivity
MultiROM Manager	71	Java	151	12,269	403	36	4,141	No	Productivity
NewPipe	49	Java	456	53,279	3,732	254	24,076	No	Photo and Video
Omni-Notes	71	Java	383	36,137	2,635	18	40,365	Yes	Productivity
Opengur	62	Java	315	21,290	1,289	3	34,276	Yes	Entertainment
OpenKeychain	113	Java	1,113	133,468	7,071	94	77,491	Yes	Communication
OpenLauncher	38	Java	207	21,152	836	45	29,478	Yes	Personalization
OS Monitor	77	Java	201	21,080	353	6	5,141	Yes	Tools
OsmAnd	113	Java	2,037	584,702	51,395	549	409,580	Yes	Maps and Navigation
OwnCloud	97	Java	521	76,024	6,825	63	208,059	Yes	Productivity
Red Moon	51	Kotlin	121	7,247	969	76	4,945	Yes	Tools
RedReader	78	Java	351	44,499	1,242	86	4,529	Yes	News and Magazines
Riot.im*	52	Java	557	101,762	5,725	158	857,383	Yes	Communication
RunnerUp	82	Java	438	49,313	2,555	32	8,248	Yes	Health and Fitness
Signal Private Messenger	93	Java	1,169	150,534	3,758	174	143,355	Yes	Communication
Silence*	93	Java	785	79,413	2,560	121	68,188	Yes	Communication
Simple Calendar	44	Kotlin	231	20,828	2,784	65	13,988	Yes	Tools
Simple Camera	42	Kotlin	83	5,429	973	38	8,157	Yes	Tools
Simplenote	84	Java	228	14,610	1,093	25	4,800	Yes	Productivity
Slide	48	Java	734	121,123	3,613	62	125,421	Yes	News and Magazines
Status	38	Java	332	17,901	538	3	1,561	Yes	Personalization
Tasks	129	Java	765	75,088	7,159	29	101,214	Yes	Productivity
Telegram	71	Java	2,356	874,408	316	10	155,063	Yes	Communication
Terminal Emulator for Android	109	Java	197	19,048	1,038	40	6,230	Yes	Tools
Timber	51	Java	310	27,224	586	33	17,394	Yes	Music and Audio
Tomahawk	86	Java	586	50,558	2,525	13	29,509	No	Photo and Video
Wire*	45	Scala	1,197	87,168	4,285	27	63,870	Yes	Business and Economy
WordPress	120	Java	1,906	249,466	32,195	86	199,191	Yes	Productivity

Table 5 Characteristics of the considered iOS apps.

Name	Age(Months)	Language	Files	LOC	Commits	Contributors	Size	On store	Category
ART SY*	56	Objective-C	1,070	63,953	5,093	43	216,114	Yes	Shopping
Breadwallet	76	Objective-C	238	25,738	1,641	15	24,580	Yes	Business and Economy
Bridges	85	Objective-C	489	67,826	804	3	108,980	No	Games
Buck Tracker	48	Swift	190	19,176	61	3	11,838	Yes	Utilities
ChatSecure	97	Objective-C	510	40,399	3,263	16	47,532	Yes	Social Networking
CodeCombat	61	C	140	33,118	275	4	21,085	No	Games
Colloquy	199	Objective-C	742	88,255	70,002	10	174,527	No	Utilities
DesignerNews	57	Objective-C	140	4,639	375	5	24,177	No	News and Magazines
Endless	58	Objective-C	85	9,080	408	3	26,770	Yes	Utilities
Firefox	58	Swift	1,089	257,929	7,162	120	443,212	Yes	Utilities
GBA4iOS*	88	Objective-C	10,315	1,526,886	596	9	197,000	No	Games
Gorillas	130	Objective-C	150	7,920	337	1	559,224	Yes	Games
Hackers	82	Swift	50	1,919	609	3	143,243	Yes	News and Magazines
HN iOS Reader	58	Objective-C	80	3,787	42	8	40,534	Yes	News
IRCCloud iOS App	79	Objective-C	304	74,374	2,801	6	127,729	Yes	Business and Economy
Little Go	104	Objective-C	430	43,154	1,336	1	20,091	Yes	Games
Longboxed*	63	Objective-C	145	10,957	598	2	33,648	No	Utilities
Megabite	47	Objective-C	208	40,245	102	2	42,258	No	Photo and Video
Mumble	116	Objective-C	2,374	605,060	666	3	12,673	No	Games
OnionBrowser	58	Objective-C	123	10,765	510	5	39,721	Yes	Utilities
ParkenDD	56	Swift	34	1,597	536	3	37,790	Yes	Utilities
Poppins	58	Objective-C	132	2,888	134	2	16,759	No	Photo and Video
Quick Chat	37	Swift	239	25,409	83	3	111,068	No	Social Networking
Rocket.Chat*	49	Swift	758	38,780	5,890	48	79,508	Yes	Business and Economy
The Oakland Post*	63	Objective-C	208	13,924	286	1	152,374	No	News and Magazines
Trust Wallet	24	Swift	522	23,945	2,268	29	13,268	Yes	Business and Economy
Twidere*	40	Swift	224	11,559	189	2	1,459	No	Social Networking
V2EX	66	Objective-C	311	25,172	140	9	21,903	Yes	Social Networking
Wikipedia	71	Objective-C	1,412	102,237	23,326	63	560,851	Yes	Books and Reference
WordPress	134	Swift	1,664	153,219	36,294	87	447,403	Yes	Productivity
Yep	54	Swift	579	50,374	7,288	13	52,598	No	Social Networking

`SimpleDateFormat` objects was moved from inside a `for` loop to the lines of code preceding it. The second evaluator, instead, did not notice this change in the diff, and assigned a more general "*GUI-related*" tag to indicate that the performance issue was affecting the formatting of dates printed in the GUI. The conflict was solved with the third evaluator using exactly the same tag of the first evaluator (*i.e.,* "*Objects instantiation in loop*").

Another example of conflict that required an open discussion among the authors is related to commit `313aa46` from the `herzbube/littlego` iOS app. The commit note here, is quite self-explanatory: "*fix memory leaks (most of them UIBarButtonItems not autoreleased)*". The tags assigned by the three evaluators were: "*Avoid memory leaks with autorelease*", "*Memory leak*", and "*Memory leaks due to unreleased objects*". While the meaning of the three tags was basically the same, the authors agreed on adopting the third one since (i) it was more specific as compared to the second one (*i.e.,* "*Memory leak*"), (ii) as opposed to the first one, describes the problem and not the adopted solution, and (iii) considers the fact that not all leaks in the commit have been fixed through the use of `autorelease`.

We answer RQ_1 by presenting a taxonomy of the types of performance bugs identified in the manual analysis and we complement our discussion with qualitative examples. To ease the discussion of the performance bugs, we also link them to the performance criteria described in Section 2.1, indicating the performance factors on which each type of bug has an impact.

3.3 RQ_2: Data Collection & Analysis

To answer RQ_2 we need to identify the commit in which each of the 500 performance bugs has been introduced and fixed. As for the commit fixing each performance bug, we already have such an information, since the commits we manually analyzed were the bug-fixing commits. Concerning the bug-introducing commit, we used the SZZ algorithm [83] to identify it. The algorithm relies on the annotation/blame feature of versioning systems. Given a bug-fixing commit BF_k (where k identifies the bug), the approach works as follows:

1. For each file f_i, involved in BF_k and fixed in its revision *rel-fix*$_{i,k}$, we extract the file revision just *before* the bug fixing (*rel-fix*$_{i-1,k}$).
2. Starting from the revision *rel-fix*$_{i-1,k}$, for each source code line in f_i changed to fix the bug k, the *blame* feature of Git is used to identify the file revision where the last change to that line occurred.

In doing that, blank lines are ignored. This produces, for each file f_i, a set of $n_{i,k}$ fix-inducing revisions *rel-bug*$_{i,j,k}$, $j = 1 \ldots n_{i,k}$.

Since more than one commit can be indicated by the SZZ algorithm as responsible for inducing the performance bug-fix, there are time ranges defined by lower (minimum survivability) and upper bounds (maximum survivability). Therefore, we answer RQ_2 by following a meta analysis-based procedure [28,

11]: the minimum and the maximum survivability of the bugs (*i.e.,* number of days between the bug introduction and fixing) are plotted using forest plots with confidence intervals, and a central tendency measure of the survivability is computed by using the random effects model [11]. The minimum survivability is the one observed when considering the most recent commit identified by the SZZ algorithm as the one that induced the bug-fix. *Vice versa*, the maximum survivability is observed when considering the least recent commit identified by the SZZ algorithm as the one that induced the bug-fix. The forest plots are depicted by considering a 95% confidence interval. The SZZ algorithm was able to identify at least one bug-inducing commit for 380 of the 500 bug-fixing commits. For this reason, the 9 apps marked with * in Tables 4 and 5 have been excluded from RQ_2, since they did not contribute any bug-introducing commit to our dataset.

The usage of the survivability intervals also helps in dealing with the intrinsic limitations of the SZZ algorithm. To better understand why, let's assume the case of a bug-fixing commit c_{bf} performed in December 2018 and modifying the files F_i and F_j. Let's also assume that c_{bf} is a tangled commit [29], meaning that besides the bug-fixing, it also features some other types of changes (*e.g.,* refactoring). In particular, F_i has actually been modified to fix the bug, while F_j has been changed for other reasons. By running the SZZ algorithm on c_{bf}, two commits are identified: c_{Fi}, performed in March 2018, and c_{Fj} performed in August 2018, representing the most recent commits that modified F_i and F_j, respectively. Clearly, c_{Fj} is a false positive, since no bug was fixed in F_j. Thus, the correct survivability of the bug fixed in c_{bf} is 8 months (from March to December 2018). By considering both the minimum and the maximum survivability, we provide lower and upper bounds of survivability that delimit the time period in which a bug has likely been introduced. In other words, going back to our example, claiming that the bug fixed in c_{bf} survived from a maximum of 8 to a minimum of 4 months is a correct claim.

We also verified whether different types of performance bugs (as classified by the taxonomy output of RQ_1) have different survivability. In particular, we compared the distributions of the survivability of the different types of performance bugs (*e.g., resource leak vs. GUI-related*) via (i) forest plots, and (ii) statistical tests. For the latter we used the Mann-Whitney test [7] with results intended as statistically significant at $\alpha = 0.05$. To control the impact of multiple pairwise comparisons (*e.g.,* the survivability of the performance bugs in the *resource leak* category is compared against the survivability of those in the *GUI-related* and *bad practices* categories), we adjust p-values using the Holm's correction [30]. We also estimate the magnitude of the differences by using the Cliff's Delta (d), a non-parametric effect size measure [20] for ordinal data. We follow well-established guidelines to interpret the effect size: negligible for $|d| < 0.10$, small for $0.10 \leq |d| < 0.33$, medium for $0.33 \leq |d| < 0.474$, and large for $|d| \geq 0.474$ [20].

To ease the interpretation of the achieved results, we also compare the survivability of the performance bugs with that of other types of bugs, unrelated to performance issues and extracted from the same repositories in which

the performance bugs were fixed. In particular, for each performance-bug-fixing commit, we randomly selected, from the same repository in which the performance bug was fixed, two bug-fixing commits having their commit notes matching the regular expression `fix(es)(ed) bug(s)`. Then, knowing the possible imprecisions introduced by this procedure, two of the authors manually analyzed all candidate bug-fixing commits discarding the ones that were related in some way to performance issues **or** that were not bug-fixing activities. This left us with 295 valid bug-fixing commits not related to performance issues that we use to compare the survivability of the bugs they fix to that of the performance bugs fixed in the 380 performance-bug-fixing commits. Also in this case, we compute the minimum and the maximum survivability using the same procedure previously described, and we statistically compare the survivability of performance bugs with that of other types of bugs by using the Mann-Whitney test and the Cliff's Delta.

Finally, we investigate the impact of several co-factors on the survivability of performance bugs. In particular, given d as the date in which a performance bug has been fixed, we took into consideration as confounding factors:

- *Characteristics of the repository (computed at date d for each performance bug)*: the number of releases, the number of total issues, the number of issues in the state "open", the number of issues in the state "closed", the age of the repository in days, the LOC, and the number of contributors. This data has been mostly extracted by using the GitHub APIs[4]. The most notable exception is the LOC, for which we cloned each repository, checked-out the snapshot preceding d, and computed the lines of code for that snapshot using the CLOC tool. Concerning the age of the repository, rather than computing it as the days elapsed from the repository creation, we considered the days between the date of the first commit and d. This choice was done to avoid issues related to the repositories imported from other versioning systems, that could have a quite recent creation date on GitHub, but old commits in their change history.
- *Code-related factors (computed at date d for each performance bug)*: the average and median number of lines of code added and deleted per week. We included these four factors as proxies for the activity level of the studied repositories. Also this data was extracted through the GitHub APIs.
- *Patch-related factors*: the number of lines added and deleted to implement the patch, as well as the total number of modified lines of code. These three variables are proxies for the size of the patch needed to fix a certain performance bug.

We fit a Cox proportional hazards regression model [77] to analyze the impact of the above-described factors on the performance bugs survivability. To maximize the statistical power of the model, we consider the whole dataset of bug-fixing commits (*i.e.,* we merge the ones related to Android and to iOS apps), and we use the estimated minimum and maximum survivability of each bug as

[4] `https://developer.github.com/v3/`

dependent variable to build two models. For each explanatory variable used to build the model (*e.g.,* the characteristics of the repository), we analyze the returned hazard ratio (HR) as effect size measure. HR > 1 indicates, for that variable, a lower survival time for corresponding increases of the variable value. Vice versa, a HR < 1 indicates a higher survival time for higher values of the variable. Further details on survival analysis can be found in [40].

Note that, before using all explanatory variables to fit the Cox proportional hazards model, we use the *R varclus* function of the *Hmisc* package [25] to produce a hierarchical clustering of features based on their correlation, in turn, computed with a specified correlation measure (we use the Spearman's ρ rank correlation). Then, we identify clusters by cutting the tree at a given level of ρ^2 that we set at $\rho^2 = 0.49$, which corresponds to a strong correlation (*i.e.,* $\rho = 0.7$) [6]. We only keep one variable per cluster, by randomly selecting it in the cluster of correlated metrics. Also, to properly interpret the importance of each independent variable in the model, we normalize variable values, within each project, in the interval [0, 1].

4 Results discussion

This section discusses our main findings related to the two research questions that have been presented in Section 3.

4.1 Which types of performance bugs affect mobile apps?

Figures 1 and 7 show the taxonomy of performance bugs for Android and iOS apps, respectively, that we found in the 1,510 manually inspected commits. Note that, as explained in Section 3, the two figures report the classification for 500 performance bugs (250 each). This is due to the fact that we classified 510 of the inspected commits as *false positive* during our manual analysis.

One important aspect we must clarify about the two shown taxonomies is the specialization of the performance bugs into subcategories, *e.g.,* when moving from *resource leak*, to *memory leak*, down to *cursor leak* in Fig. 1. We classified each analyzed performance bug in the lowest-level category we were able to "safely" assign to that bug. For example, we identified 61 performance bugs as belonging to the *memory leak* category. Then, for 21 of them, we were able to further specialize the cause behind the bug into *issues because of strong references* (6 bugs), *cursor leak* (3), *unneeded elements* (6), *activity leaks* (7), *view leaks* (2), and *steam leaks*(3). Thus, the hierarchy of classification shown in Fig. 1 does not imply that all *memory leak* bugs can be classified into one of its three sub-categories, since for some bugs we were not able to identify the exact cause behind the memory leak.

We also notice that the types of bugs presented in Figures 1 and 7 are not restricted to bug types that do *only* affect mobile apps. For example, *Inefficient SQL* queries can be found in any type of system. The goal of the

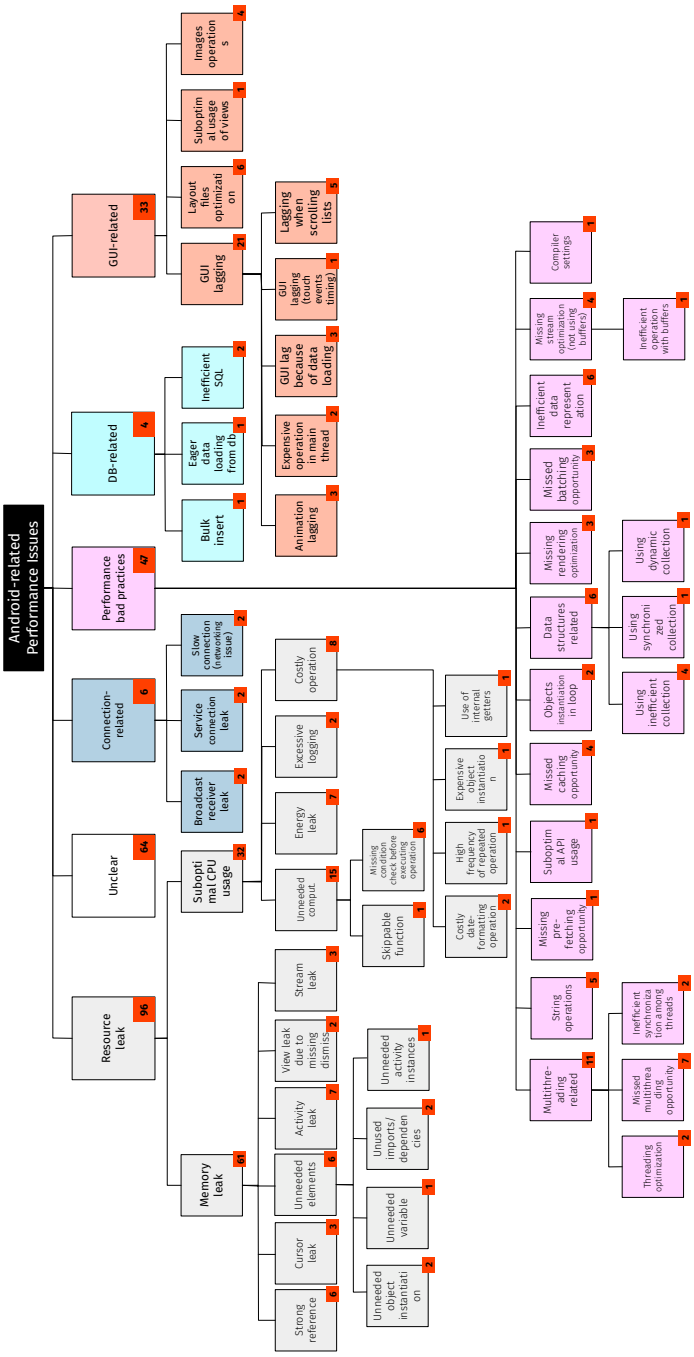

Fig. 1 Types of performance bugs found in Android apps.

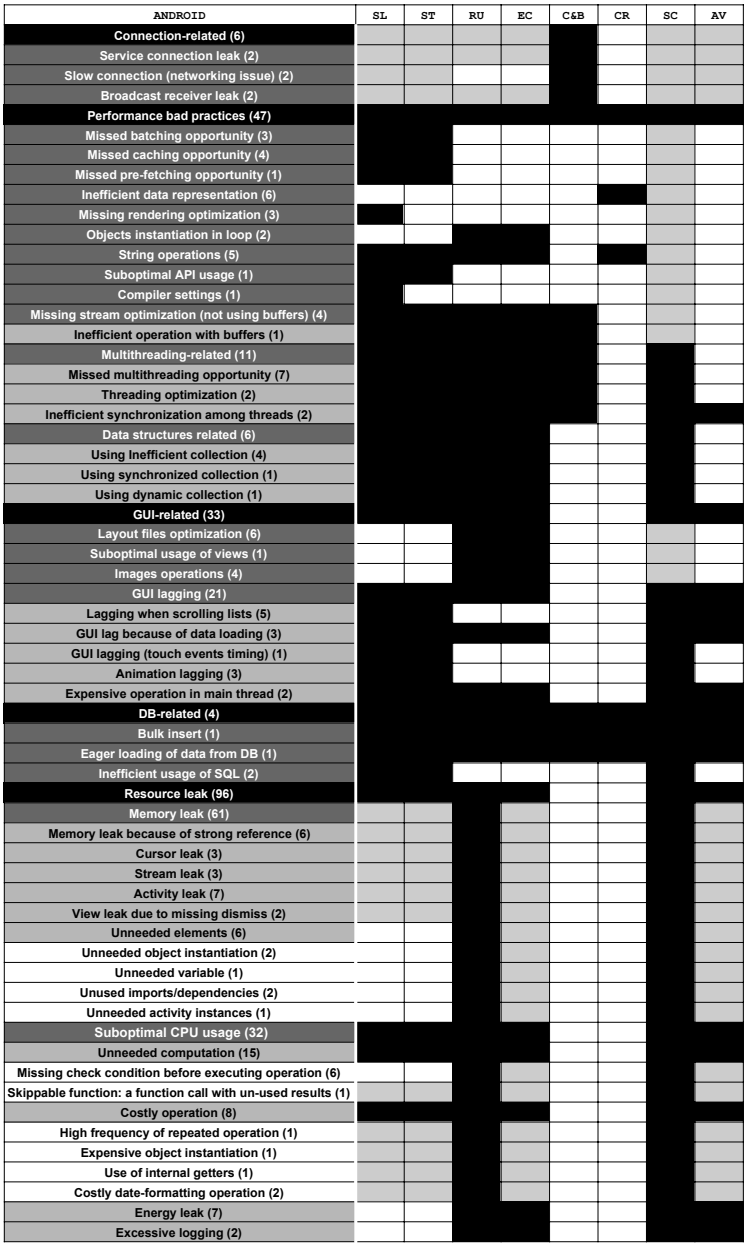

Fig. 2 Impact of performance bugs in Android apps to performance-related factors (Section 2.1): Service Latency (SL), System Throughput (ST), Resource Utilization (RU), Energy Consumption (EC), Communication & Bandwidth (C&B), Compression Ratio (CR), Scalability (SC), Availability (AV). A black entry indicates a strong impact, a gray entry a weak impact, and a white entry no impact.

two taxonomies is to present performance bugs that we found in mobile apps through our manual inspection process, despite the fact that they are specific of mobile apps (*e.g., Suboptimal usage of views*) or that they can also affect other types of software (*e.g., Excessive logging*).

In addition to the two taxonomy figures, we also present in Figures 2 and 8 how the bug types identified in Android and iOS apps, respectively, can potentially impact the performance factors we defined in Section 2.1. The color associated to each bug type (*i.e.,* from white to black using different scales of gray) indicate their level in the respective taxonomy, with black being the root categories, and white the lowest-level subcategories. A black entry at the intersection between a bug type and a performance criterion indicates a strong negative impact of that type of bug on the related criterion (*e.g.,* bugs in the *Connection-related* category can potentially have a strong impact on the *Communication & Bandwidth* criteria), a gray entry a weak impact, and a white entry no impact. The impact of each bug type has been defined by the first author and double-checked and refined by the other authors, and it is based on the bug instances in each category as well as on the description of the performance criteria we reported in Section 2.1.

4.1.1 Performance bugs in Android apps

The performance bugs most frequently affecting Android apps are related to ***resource leak*** (96 instances). This category groups all the bugs that are related to improper handling of device resources, mostly because of bad programming practices (*e.g.,* missing to close a stream)[5]. We classified these bugs into two subcategories: *memory leak* and *suboptimal CPU usage*. The former includes cases in which the app incorrectly manages memory allocations (*e.g.,* it does not destroy unnecessary objects in memory after their usage).

Activity leaks and *strong references* are the top causes we were able to identify for *memory leaks*. Strong references may prevent the garbage collection process in both Android and iOS when the references create cyclic dependencies between objects (*e.g.,* A has a reference to B and B has a reference to A) and one of the sides in the dependency cycle can not be claimed (in terms of memory management) because there is still an "active" reference. One specific case of this is represented by the activity leaks, and in particular when activities are passed as references to long-term background execution components in Android such as AsyncTasks, Threads, etc. (*e.g.,* worker threads). When a worker thread is executed but the GUI moves from one activity to another, the first activity (*i.e.,* the one that started the service) can not be disposed if the worker has a reference to the activity and the worker is still running. It can derive in cases where the activity is allocated multiples times in memory, because the activity cycle is blocked by the worker execution.

[5] Note that there is also a category "performance bad practices" in which we group issues related to high-level practices, *i.e.,* issues due to the fact that developers do not apply performance best practices proposed by the platform designers.

Fig. 3 is an example of strong reference between a `AsyncTaskLoader` and an activity. Such reference generates an *activity leak* when the activity is used to create a toast (on the GUI thread) to notify the user when the loader finishes the task. Fig. 3 reports: (i) the GitHub repository name (*i.e.,* cgeo) and the specific commit in which the bug has been fixed (*i.e.,* 837b8cc); (ii) the commit note left by the commit author; (iii) the buggy code (red and top rectangle); and (iv) the fixed code (green and bottom rectangle). Note that in Fig. 3, as well as in all figures we will use for qualitatively discuss our findings, we sometimes omit parts of the code and summarize it with comments (see *e.g.,* the comment used as the `[..] more code` body in Fig. 3). The fix to the performance bug consisted of removing the strong reference by converting it into a weak reference at the `AsyncTaskLoader` side, by using the Android `WeakReference` API.

Activity leak | **Android** | cgeo@837b8cc

Commit note: *memory leak in cache list loaders*

```
public abstract class AbstractSearchLoader extends
AsyncTaskLoader<SearchResult> implements RecaptchaReceiver {
    //[..] more code
    private final Activity activity;
    //[..] more code
    public AbstractSearchLoader(final Activity activity) {
        super(activity);
        this.activity = activity;
    }
    //[..] more code
    public SearchResult loadInBackground() {
        //[..] more code
       activity.runOnUiThread(new Runnable() {/**more**/});
    }
}
```

```
public abstract class AbstractSearchLoader extends
AsyncTaskLoader<SearchResult> implements RecaptchaReceiver {
    //[..] more code
    private final WeakReference<Activity> activityRef;
    //[..] more code
    public AbstractSearchLoader(final Activity activity) {
        super(activity);
        this.activityRef = new WeakReference<>(activity);
    }
     //[..] more code
    public SearchResult loadInBackground() {
        //[..] more code
        final Activity activity = activityRef.get();
        if (activity != null) {
          activity.runOnUiThread(new Runnable() {/**more**/});
        }
    }
}
```

Fig. 3 Activity leak issue in Android app when having strong references.

The performance bug summarized in Fig. 4 is a representative example for a memory leak created when not closing a stream. The involved snippet of

Cursor Leak | **Android** | signalapp@77e846d

Commit note: *Fix cursor leak when resolving contact photos*

```
if (cursor != null && cursor.moveToFirst()) {
      //use the cursor to load the contact photo
} else {
      //load the default contact photo
}
```

```
try {
      if (cursor != null && cursor.moveToFirst()) {
            //use the cursor to load the contact photo
      } else {
           //load the default contact photo
      }
    } finally {
      if (cursor != null) cursor.close();
    }
```

Fig. 4 Cursor leak bug fixed in Android app.

code is in charge of loading the photo of a given contact, and uses a `Cursor` to access the result set obtained by querying the database. The `Cursor` object in Android implements the `Closable` interface and, as a consequence, it must implement the `close`() method, in charge of releasing all resources that the object (the `Cursor` in this case) is holding (thus freeing up the memory). In the buggy version, the `Cursor` object was not closed after its usage (see red and top rectangle in Fig. 4), leading to a memory leak. The problem has been solved by properly invoking the `close`() method on the cursor in the `finally` block.

Similarly, in commit `376bc22` performed in the Osmand GitHub repository, a *stream leak* with a XML serializer is fixed (commit note: *Close file streams to avoid leakage*) by closing the previously created `FileOutputStream` object in a `finally` block.

The previous two examples are representative of several performance bugs we identified in the *memory leak* category, and highlight as a good number of such bugs could be simply avoided by invoking the `close`() method on objects having a type implementing the `Closeable` interface. This result might look surprising considering that modern IDEs usually provide warnings when these objects are not closed by the developers.

Moving to the *suboptimal CPU usage* performance bugs, we classified as such 32 bugs resulting in the excessive/unnecessary usage of the CPU (*e.g.,* unneeded computation, excessive logging, etc.). We were able to identify the causes for the 32 bugs and we sub-categorized them in *unneeded computation*(15), *energy leaks*(7), *excessive logging*(2), and *costly operations*(8).

Fig. 5 summarizes a bug falling in the *costly operation* category. In the fixing (green and bottom box) the developer is improving performance by changing the `launchMode` for an activity (*i.e.,* `AllInOneActivity`). In Android, activities

are defined as "*a single, focused thing that the user can do*", and, as explained in the official Android documentation: "*When a new activity is started, it is placed on the top of the stack and becomes the running activity – the previous activity always remains below it in the stack, and will not come to the foreground again until the new activity exits*" [6]. The number of running activities has an impact on the consumed CPU cycles. The `launchMode` allows Android developers to specify in which situation a new activity instance is created. With the standard `launchMode` (*i.e.,* the default one used in the "buggy" version), a new activity instance is created every time there is a new `Intent`, while with `launchMode` set to `singleTop` an existing instance of an activity is reused if it is at the top of the stack when the new `Intent` is created. This allows to save computational resources.

Unneeded activity instances | **Android** | Etar-Calendar@2dd566a

Commit note: *Fixed AllInOneActivity leak*

```
<activity
      android:name="AllInOneActivity"
      android:theme="@style/CalendarTheme.WithActionBar">
</activity>
```

```
<activity
      android:name="AllInOneActivity"
      android:theme="@style/CalendarTheme.WithActionBar"
      android:launchMode="singleTop">
</activity>
```

Fig. 5 Costly operation fixed in Android app.

The second most occurring category of performance bugs in our Android taxonomy is ***performance bad practices*** (47 instances), grouping bugs due to the non-application of well-known performance best practices. This includes the sub-category *missed batching opportunity*, referring to bugs in which batching (*i.e.,* a strategy for increasing performance and scalability when interacting with a database and/or a remote service) has not being used in the first place. One representative case is the commit `852e513`, performed in the K-9 Mail GitHub repository (an open-source email client for Android). In this commit, the developer is fixing a performance bug well summarized in the commit note:

> *Modified fetch() to call fetchEnvelope() which runs recursively, grabbing message ENVELOPEs 10 at a time rather than attempting all 100 at once. Shows a significant performance increase and a significant reduction in memory usage.*

Another sub-category of the *performance bad practices* is the *missed caching opportunity*, related to all performance bugs due to the missing "storage" of

[6] `https://developer.android.com/reference/android/app/Activity.html`

information. Such information is reused multiple times by the app but it is recomputed (or re-queried) every time is needed. Fig. 6 reports a performance bug belonging to this category from the WordPress Android app.

Missed caching opportunity | **Android** | WordPress-Android@941cb8a

Commit note: *[...] status colors are now cached for performance*

```
private ListView getListView() {
  switch (comment.getStatusEnum()) {
    case SPAM :
      //[..] more code
      txtStatus.setTextColor(Color.parseColor("#FF0000"));
      break;
    case UNAPPROVED:
      //[..] more code
      txtStatus.setTextColor(Color.parseColor("#D54E21"));
      break;
    //[..]
}
```

```
private ListView getListView() {
  switch (comment.getStatusEnum()) {
    case SPAM :
      //[..] more code
      txtStatus.setTextColor(mStatusColorSpam);
      break;
    case UNAPPROVED:
      //[..] more code
      txtStatus.setTextColor(mStatusColorUnapproved);
      break;
    //[..]
}
```

Fig. 6 Missed caching opportunity bug fixed in Android app.

Due to space constraints, part of the fixing is not reported in the figure, and concerns the initialization of two variables, namely `mStatusColorSpam` and `mStatusColorUnapproved`, aimed at storing the colors used to highlight comments marked as spam and unapproved, respectively. These variables have been introduced to avoid the invocation of the `Color.parseColor(String)` every time the two colors are needed when invoking the `getListView()` method (see buggy and fixed versions in Fig. 6). While such optimizations might look minimal in terms of performance gain, it is important to keep in mind the context in which apps are executed, meaning mobile devices showing limited resources, especially for what concerns battery. Thus, every "drop of energy" that can be saved really must be preserved.

The third most frequent category of performance bugs we identified on Android apps groups is named ***GUI-related*** bugs (see Fig. 1). For example, performance bugs causing *animation lagging* belong to this category. One of the bugs in this category is described in the issue #79 of the simplenote-android app[7], that has been fixed in one of the commits we analyzed (*i.e.,* commit `bb9beec`). The bug causes a lag when the app's user navigates from a note back

[7] `https://github.com/Automattic/simplenote-android/issues/79`

to its list of notes. One of the contributors taking part in the issue discussion noted that the lag was particularly visible when the keyboard was shown on screen after editing a note. The fixing is described in the commit note:

> *Moved the keyboard hide code out of onPause of NoteEditorFragment and into onResume of NoteListFragment to improve the transition animation performance.*

Thus, the hiding of the keyboard is now performed when the user resumes (*i.e.,* goes back to) the list of notes rather than when pausing the fragment showing a single node. This resulted in smoother animations, possibly due to the `onPause` method in `NoteEditorFragment` which is also in charge of performing several other operations (*e.g.,* saving changes made to the note) that, combined with the hiding of the keyboard, might have resulted in the GUI lagging.

Other less diffused categories of performance bugs in Android apps are those related to: (i) *connectivity* (6 instances), such as broadcast receiver leaks and network delays; and (ii) *database operations* (4 instances), such as inefficient queries and eager data loading.

4.1.2 Performance bugs in iOS apps

Before starting the discussion of the performance bugs we found in iOS apps, it is important to highlight the introduction of the Swift[8] programming language in 2014, that represents the recommended language for the implementation of iOS apps. Before 2014, the supported programming language was Objective-C[9]. This means that some of the apps we analyzed might have been subject to a migration from Objective-C towards Swift during the analyzed change history and, as a consequence, some of the performance bug-fixing commits might be related to Swift while others to Objective-C. Note also that iOS apps can contain both Swift and Objective-C code.

As already observed for Android apps, also in iOS apps the performance bugs most widely diffused are related to ***resource leak*** (120 instances) and, the vast majority of these (110), fall in the *memory leak* category. Many of them are related to the missing use of the `autorelease`, *i.e.,* a mechanism implemented both in Objective-C and in Swift to manage the deallocation of objects from the memory.

Fig. 9 depicts an example of *unreleased objects* performance bug fixed in the Little Go iOS app. This app implements the game of Go, in which users can play against a human or against the computer. The fixing of the bug consists of the simple addition of the `autorelease` keyword in the return statement. This allows the developer to "ask" the deallocation of the returned object from the memory but not immediately (*i.e.,* not before it is returned). This bug is related to Objective-C code.

[8] `https://www.apple.com/it/swift/`

[9] `https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`

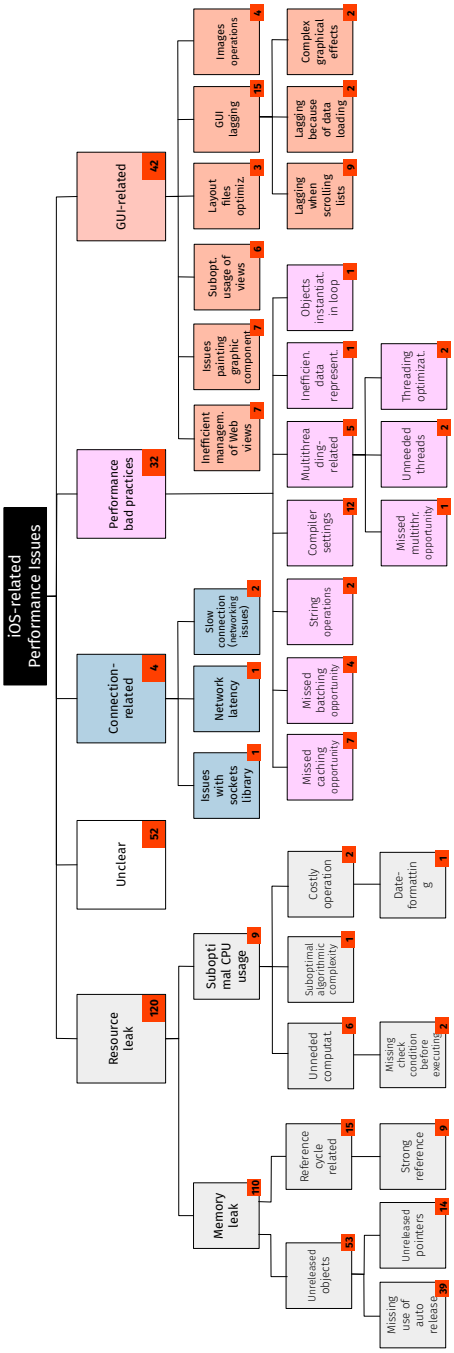

Fig. 7 Types of performance bugs found in iOS apps.

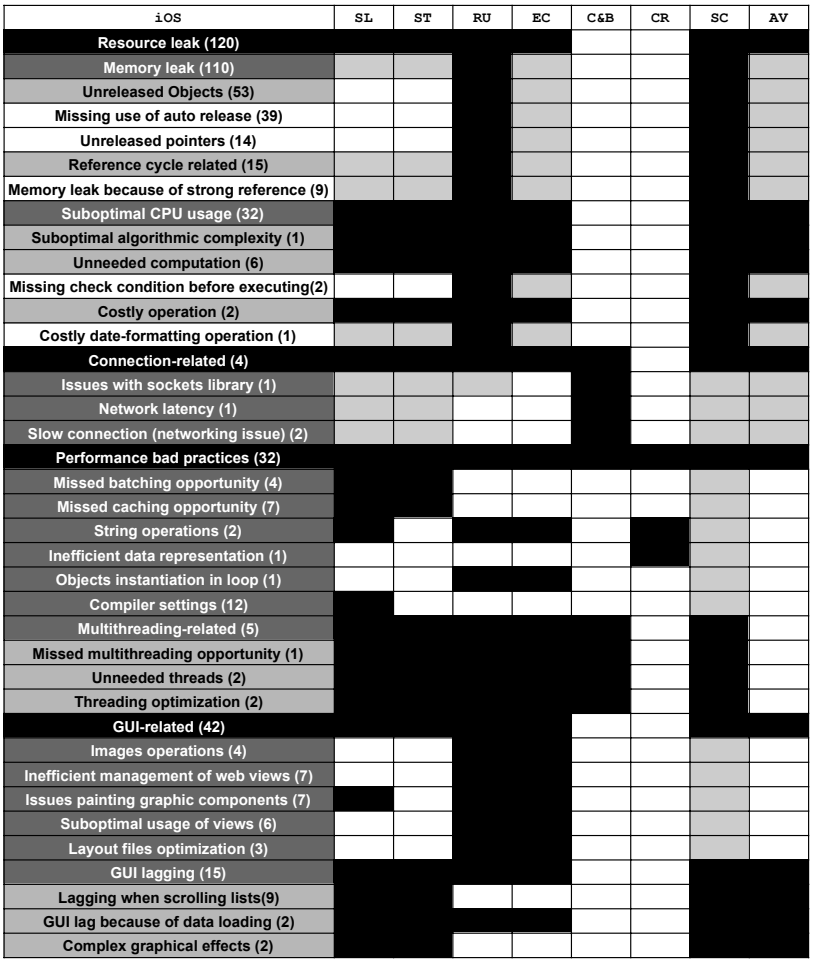

Fig. 8 Impact of performance bugs in iOS apps to performance-related factors (Section 2.1): Service Latency (SL), System Throughput (ST), Resource Utilization (RU), Energy Consumption (EC), Communication & Bandwidth (C&B), Compression Ratio (CR), Scalability (SC), Availability (AV). A black entry indicates a strong impact, a gray entry a weak impact, and a white entry no impact.

An example of bug affecting Swift code is instead the one reported in Fig. 10, and related to the *missing use of unowned* category (see Fig. 7). The bug refers to the Firefox iOS app, and specifically to two table view controllers in the `AppSettingsOptions` class that were missing the usage of `unowned`. Swift provides two ways to resolve strong reference cycles, *i.e.,* weak and unowned references. Both enable one instance in a reference cycle to refer to the other instance without keeping a strong hold on it. An unowned reference results useful when the other instance (that can be deallocated first) has the same or a longer lifetime. This prevents an excessive usage of the memory. Fig. 11

Unreleased objects | **iOS** | littlego@2f46e5e

Commit note: *fix memory leak in ArchiveViewModel [...]*

```
if (! [self gameWithName:preferredGameName])
          return [preferredGameName copy];
```

```
if (! [self gameWithName:preferredGameName])
          return [[preferredGameName copy] autorelease];
```

Fig. 9 Unreleased objects bug fixed in iOS app.

shows an example of how to avoid a strong reference by declaring a variable as `weak`; note that a weak variable can be declared only in classes, therefore the `WeakTabManagerDelegate` structure in the example was changed to a class.

Missing use of unowned | **iOS** | firefox-ios@d1a84ba

Commit note: *Bug 1278355 - Memory Leak in AppSetting*

```
let settings: SettingsTableViewController
//[..] more code
let settings: SettingsTableViewController
```

```
unowned let settings: SettingsTableViewController
//[..] more code
unowned let settings: SettingsTableViewController
```

Fig. 10 Missing use of unowned bug fixed in iOS app.

Strong reference | **iOS** | firefox-ios@c4f2ddc

Commit note: *Bug 1157843 - Use a weak ref for tabTrayController*

```
struct WeakTabManagerDelegate {
    var value : TabManagerDelegate?
```

```
class WeakTabManagerDelegate {
    weak var value : TabManagerDelegate?
```

Fig. 11 Removing strong reference in iOS app.

Concerning the *suboptimal CPU usage* category we put here, for example, performance bugs due to *unneeded computations*, meaning computations that could be avoided. One of these cases is the bug `0886c86` from the Colloquy app, *i.e.,* a chat client. As described by the developer in the commit message:

Outline list performance GREATLY improved:

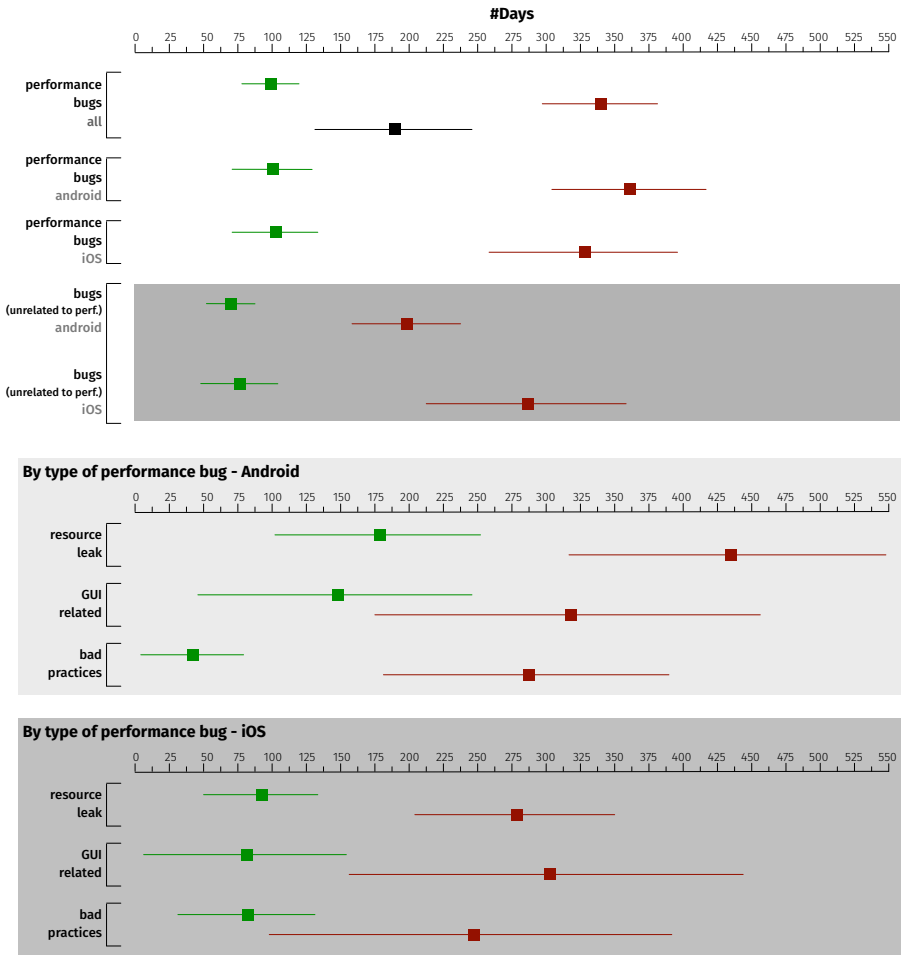

Fig. 12 Survivability in days of performance bugs in Android and iOS apps. Green (red) depicts minimum (maximum) estimates at 95% confidence interval. Black shows the results of the random effect model.

- *sizeLastColumnToFit moved into outlineViewDidExpand item, so it is only called when needed;*

- *removed called to _refreshSelectionMenu since it wasn't need since SelectionDidChange was already implemented to do it.*

Now these calls are only done when necessary rather than for every cell.

Basically, the developer removed a number of unneeded method calls, that had a negative impact on the performance of the corresponding mobile app.

The second most diffused category of performance bugs in iOS apps are the *GUI-related* ones, with 42 instances. This includes *issues painting graphic components* (*e.g.,* commit `1365614` from Colloquy: "*Improve scroll performance in the chat room list*"), and *inefficient management of Web views* (*e.g.,* commit `e03050c` from Firefox-iOS: *Improve performance by adding/removing webviews instead of layering them*).

Finally, further less diffused categories of performance bugs in iOS apps are those related to the *performance bad practices* (32 instances), and to *connectivity* (4 instances), such as sockets library and network latencies.

4.2 How long does it take to fix performance bugs in mobile apps?

Fig. 12 depicts the forest plots reporting the survivability of performance bugs in the analyzed mobile apps (*i.e.,* the number of days between the performance bug introduction and its fixing). As explained in Section 3, we report the minimum (green) and the maximum (red) survivability intervals as computed with the SZZ algorithm. In each forest plot the square represents the average value of the distribution, while the line passing through it depicts the 95% confidence interval. The black line shown for the overall set of performance bugs depicts the results of the random effects model [5], used in meta-analysis to combine the results of different studies in a single result outcome. In our case, the set of "different studies" includes the survivability estimates when considering the minimum (*study I*) and the maximum (*study II*) survivability.

Starting from the top, Fig. 12 shows the survivability intervals when (i) considering all the analyzed performance bugs together (*i.e.,* "performance bugs all"); (ii) when grouping them by mobile platform (Android and iOS); and (iii) considering bugs unrelated to performance issues, for Android (*i.e.,* "bugs unrelated to perf. android") and iOS.

Finally, the bottom part of Fig. 12 compares the survivability intervals for different types of Android and iOS performance bugs. For this analysis, we only considered the types of performance bugs for which we had at least 20 instances (*i.e.,* 20 bug-fixing commit with their related bug-inducing commits) in our dataset. Indeed, computing the survivability intervals for a type of performance bugs for which we only have a few data points would result in an almost meaningless statistical analysis of confidence intervals.

One interesting result to highlight from the analysis of Fig. 12 is the long survivability of all the analyzed performance bugs. Indeed, even when considering the most conservative results (*i.e.,* the minimum estimated survivability—green line), the number of days needed to fix an introduced performance bug is, on average, 98, and it grows to 178 for the random effects model, and to 342 for the maximum estimated survivability. It is important to note that this is not the number of days needed to fix a performance bug after *it has been reported*, as investigated in the work by Liu *et al.* [57], but after *it has been introduced*. This means that a performance bug could remain unnoticed in the app for months before being identified and then fixed.

The analysis of Fig. 12 also shows that performance bugs have a similar survivability in Android and iOS apps, with no significant difference observed (p-value=0.91 for minimum and 0.21 for maximum survivability). When compared to bugs unrelated to performance issues, we observed differences in the maximum survivability intervals. In particular, the maximum survivability of performance-related bugs is higher as compared to that of bugs unrelated to performance issues (avg. 357 *vs* 197 in Android, and 326 *vs* 283 in iOS), with a significant difference for Android: p-value$<$0.01 with a small effect size (0.24). While we do not have any empirical evidence for explaining such a finding, one possible reason behind it may be that performance-bugs might manifest only in specific usage scenarios (*e.g.,* when loading large amount of data), thus only affecting a minority of the users and requiring more time to be identified as compared, for example, to functional bugs.

We also discuss differences observed for types of performance bugs, as shown in the bottom part of Fig. 12. Looking at Android, the only interesting pattern we observe is that "resource leak" performance bugs tend to survive more as compared to the other types of bug. This is also the only statistical comparison for which we observed significant differences. Indeed, as explained in Section 3, we compare the minimum and maximum survivability of (i) different types of bugs within the same operating system (*e.g.,* "resource leak" *vs* "bad practices", both in Android apps), and (ii) the same type of bugs in the apps of the two operating systems (*e.g.,* "resource leak" bugs in Android apps *vs* "resource leak" bugs in iOS apps). The only significant difference across these comparisons is observed for the minimum survivability of resource leak bugs in Android as compared to bad performance practices in Android (adjusted p-value=0.016 with a medium effect size d=0.36). This pattern (*i.e.,* the higher survivability of "resource leak" bugs) is instead not observed in iOS, in which the survivability of the different performance bug types is very similar (as also confirmed by the statistical analysis, it does not show any significant difference). This finding might be explained by the fact that memory leaks, representing the majority of the resource leaks in iOS, are a well-known issue for iOS developers. Summarizing, our main finding on the difference in survivability between different types of performance bugs is that bugs related to resource leak tend to survive longer than other performance bugs in the context of Android apps.

Finally, we study the impact of the confounding factors described in Section 3 on the survivability of performance bugs. Table 6 reports the seven clusters of factors we identified through correlation analysis, and shows the one we randomly picked from each cluster (marked with ✓and *emphasized*).

The seven selected factors have been used to build two Cox proportional hazards regression models, one using the minimum survivability as dependent variable, and one using the maximum survivability. Tables 7 and 8 report the obtained models for minimum and maximum survivability, respectively. We focus our discussion on the HR of each factor (see Section 3 for a description of the HR and its interpretation). The statistically significant HRs are reported in bold. All null hypothesis (that must be rejected in order to have a significant

Table 6 Correlation analysis for studied confounding factors. Emphasized the ones used to build the Cox proportional hazards regression model.

Factor	Cluster ID	Selected
Number of releases	1	✓
Number of issues	2	✓
Number of closed issues	2	✗
Number of opened issues	2	✗
Age of the repository (days)	3	✓
Lines of Code	4	✓
Number of Contributors	5	✓
Code addition (mean per week)	6	✓
Code addition (median per week)	6	✗
Code deletion (mean per week)	6	✗
Code deletion (median per week)	6	✗
Patch: Impacted lines	7	✓
Patch: Added lines	7	✗
Patch: Deleted lines	7	✗

Table 7 Minimum survivability: HR of the analyzed confounding factors.

Factor	HR	p-value
Number of releases	0.38	0.098
Number of issues	0.77	0.449
Age of the repository (days)	**0.34**	**0.005**
Lines of Code	**0.47**	**0.021**
Number of Contributors	1.19	0.531
Code addition (mean per week)	1.23	0.364
Patch: Impacted lines	**3.88**	**0.009**

Cox model [77]) have been rejected in our case (*i.e.,* Likelihood ratio test, Wald test, and logrank test, all reporting a p-value < 0.001).

Three factors play a significant role on the minimum estimated survivability of performance bugs (see Table 7). The first, *i.e.,* the *age of the repository in days*, shows an expected correlation with the minimum (as well as the maximum — Table 8) survivability. The HR lower than 1 indicates that higher values for the factor (*i.e.,* longer history) results in higher survivability. This result is not surprising, considering that a long history is needed to have bugs with a long survivability. More interesting are the results observed for the *lines of code* and the *size of the patch* used to fix the bug. A larger size of the repository results in longer bug survivability. This may be due to the fact that identifying a bug in a larger repository is likely to take longer. Note that we are not referring to the bug localization task (as in, the bug has been observed, and should be localized in the code); larger systems are likely to implement more features and, thus, the chance of observing issues caused by a performance bug may be lower (*e.g.,* if the bug does only affect a poorly used feature).

Concerning the *size of the patch* used to fix the bug, we can see from Table 7 that bugs requiring larger patches have a much lower minimum survivability (HR=3.88). Our conjecture is that these bugs, requiring a larger fix, are likely to be easier to catch, since possibly affecting different parts of the code.

Table 8 Maximum survivability: HR of the analyzed confounding factors.

Factor	HR	p-value
Number of releases	**0.09**	**0.002**
Number of issues	0.85	0.658
Age of the repository (days)	**0.04**	**<0.001**
Lines of Code	1.09	0.792
Number of Contributors	1.18	0.578
Code addition (mean per week)	0.72	0.139
Patch: Impacted lines	0.59	0.473

Looking at Table 8 (*i.e.,* maximum survivability), besides the already discussed *age of the repository*, we also observe the role played by the number of releases issued by the project which, also in this case, has a positive correlation with survivability (the higher the number of releases the longer the survivability). While this result may be counterintuitive since one would expect that projects issuing releases more frequently tend to fix bugs quicker as compared to projects rarely issuing releases, it is important to notice that our *number of releases* factor computes the number of releases issued until the date of the fixing, not the frequency of releases. Thus, it is in some way likely to be related to the age of a repository (even if it does not correlate with it).

Finally, it is interesting to notice that, while we observed a strong impact of the patch size on the minimum survivability, this factor does not influence the maximum survivability. Speculating about this finding is difficult. However, one possible explanation for it is that the maximum survivability estimates represent the worst-case scenario in which a bug has affected the system for very long time without being noticed (*e.g.,* because it does not have a strong negative effect on the apps' performance). Thus, it is not susceptible of influence from any of the investigated factors if not those in some way related to the longevity of the repository.

5 Threats to Validity

Threats to *construct validity* concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. This is the most important kind of threat for our study, and is related to:

- RQ_1*: Selection of the considered mobile applications.* Our study focuses on the analysis of 1,396 commits: 1,016 related to Android apps and 380 to iOS apps. In total, these commits have been found in 78 apps: 47 Android apps (87% published at the Google Play store) and 31 iOS apps (58% published at the iOS app store). The selection is motivated to consider different characteristics (*i.e.,* category, size, history, and commits), but we are aware that this set of apps is not representative of the whole population of apps developed for the two mobile platforms.

- *RQ_1: Subjectivity in the manual classification.* We identified through manual analysis the types of performance bugs affecting mobile apps. To mitigate subjectivity bias in such a process, two authors have been assigned to each performance bug and, in cases for which there was no agreement between the two authors, the document was automatically assigned to a third evaluator. In cases in which a majority agreement was not reached even with the third evaluator, an open discussion was performed to solve the conflict. Note also that, when the type of the performance bug was unclear, we preferred to explicitly label it as "unclear" rather than risking to introduce imprecisions.
- *RQ_2: Approximations due to identifying bug-inducing commits using the SZZ algorithm [83].* We computed both the minimum and the maximum survivability estimates on the basis of the SZZ outcome.
- *RQ_2: Imprecision due to tangled code changes [29].* We cannot exclude that some bug-fixing commits grouped together tangled code changes, of which just a subset was focusing on the performance bug fix. This would result in imprecisions when running the SZZ algorithm on the fixing commit. Again, by presenting both the minimum and the maximum survivability estimates such a risk is mitigated.

Threats to *internal validity* concern external factors we did not consider that could affect the variables and the relations being investigated. When possible, we addressed internal validity by qualitatively analyzing interesting cases.

Threats to *conclusion validity* concern the relation between the treatment and the outcome. Although this is mainly an observational study, wherever possible we used an appropriate support of statistical procedures, integrated with effect size measures that, besides the significance of the differences found, highlight the magnitude of such differences.

Threats to *external validity* concern the generalization of results. In RQ_1 we considered a total of 500 performance bugs (250 commits in Android apps, and 250 in iOS apps), while the RQ_2's findings are based on the analysis of 380 performance bugs due to 120 bugs for which the fixing-commit was not found anymore in the target repository. This is due to the fact that the manual analysis for RQ_1 has been performed one year before the RQ_2's survivability analysis. Thus, some repositories have been moved or rebased, do not allowing to retrieve the bug-fixing commit.

6 Conclusion and future work

In this paper we investigated the most common performance bugs affecting Android and iOS apps. After classifying them in different categories (*i.e.,* resource leak, performance best practices, etc.), we analyzed their survivability (*i.e.,* the number of days between the bug introduction and its fixing). Our manual analysis allowed to define a detailed taxonomy of performance bugs that, to the best of our knowledge, is the one based on the largest set of manually analyzed performance bugs affecting mobile apps. Such a taxonomy (RQ_1)

can point practitioners to common bad practices that should be avoided; also, researchers could design and implement approaches focused on the detection and fixing of different types of performance bugs in both platforms; finally, API/platform designers can use our taxonomy to implement specific mechanisms in the languages and IDEs to warn and avoid performance bugs in the source code.

We also found that performance bugs survive long in the system (RQ_2). While the reasons behind this result may be multiple, one possible explanation for this finding has been provided in previous work showing that performance bugs are more difficult to fix as compared to other bugs [67] and require the involvement of more expert developers [97]. Future work could further investigate the reasons behind this finding through surveys/interviews with software developers involved in the fixing of performance bugs. Based on our RQ_2's findings, highlighting the low survivability of performance bugs, more effort should be invested in characterize the code idioms causing these bugs, developing tools to (i) improve the performance testing practices, and (ii) automatically identify performance bugs.

Acknowledgements Mazuera-Rozo and Bavota gratefully acknowledge the financial support of the Swiss National Science Foundation for the CCQR project (SNF Project No. 175513).

References

1. Ali, M., Joorabchi, M.E., Mesbah, A.: Same app, different app stores: A comparative study. In: International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 79–90 (2017)
2. Boehm, B.W., Brown, J.R., Lipow, M.: Quantitative evaluation of software quality. In: Proceedings of the 2nd international conference on Software engineering, pp. 592–605. IEEE Computer Society Press (1976)
3. Brebner, P.C.: Automatic performance modelling from application performance management (APM) data: An experience report. In: International Conference on Performance Engineering (ICPE), pp. 55–61 (2016)
4. Carroll, A., Heiser, G.: An analysis of power consumption in a smartphone. In: USENIX Annual Technical Conference (2010)
5. Christensen, R.: Plane Answers to Complex Questions: The Theory of Linear Models, fouth edn. Springer Texts in Statistics. Springer (2011)
6. Cohen, J.: Statistical power analysis for the behavioral sciences, 2nd edition edn. Lawrence Earlbaum Associates (1988)
7. Conover, W.J.: Practical Nonparametric Statistics, 3rd edition edn. Wiley (1998)
8. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis. Springer (2011)
9. Cruz, L., Abreu, R.: Performance-based guidelines for energy efficient mobile applications. In: International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 46–57 (2017)
10. Cruz, L., Abreu, R.: Catalog of energy patterns for mobile applications. Empirical Software Engineering pp. 1–27 (2019)
11. Cumming, G.: Introduction to the New Statistics:Effect Sizes, Confidence Intervals, and Meta-Analysis. Routledge (2011)
12. Di Franco, A., Guo, H., Rubio-González, C.: A comprehensive study of real-world numerical bug characteristics. In: International Conference on Automated Software Engineering (ASE), pp. 509–519 (2017)

13. Fazzini, M., Orso, A.: Automated cross-platform inconsistency detection for mobile apps. In: International Conference on Automated Software Engineering (ASE), pp. 308–318 (2017)
14. Fling, B.: Mobile design and development: Practical concepts and techniques for creating mobile sites and Web apps. " O'Reilly Media, Inc." (2009)
15. Franke, D., Weise, C.: Providing a software quality framework for testing of mobile applications. In: International Conference on Software Testing, Verification and Validation (ICST), pp. 431–434 (2011)
16. Gao, Z., Bird, C., Barr, E.T.: To type or not to type: quantifying detectable bugs in javascript. In: International Conference on Software Engineering (ICSE), pp. 758–769 (2017)
17. Gegick, M., Rotella, P., Xie, T.: Identifying security bug reports via text mining: An industrial case study. In: Working conference on Mining software repositories (MSR), pp. 11–20 (2010)
18. Grechanik, M., Fu, C., Xie, Q.: Automatically Finding Performance Problems with Feedback-directed Learning Software Testing. In: International Conference on Software Engineering (ICSE), pp. 156–166 (2012)
19. Gregg, B.: Systems performance: enterprise and the cloud. Pearson Education (2013)
20. Grissom, R.J., Kim, J.J.: Effect sizes for research: A broad practical approach, 2nd edition edn. Lawrence Earlbaum Associates (2005)
21. Gui, J., Mcilroy, S., Nagappan, M., Halfond, W.G.J.: Truth in advertising: The hidden cost of mobile ads for software developers. In: International Conference on Software Engineering (ICSE), pp. 100–110 (2015)
22. Guo, C., Zhang, J., Yan, J., Zhang, Z., Zhang, Y.: Characterizing and detecting resource leaks in android applications. In: International Conference on Automated Software Engineering (ASE), pp. 389–398 (2013)
23. Hao, S., Li, D., Halfond, W.G.J., Govindan, R.: Estimating Android applications' CPU energy usage via Bytecode profiling. In: International Workshop on Green and Sustainable Software (GREENS), pp. 1–7 (2012)
24. Hao, S., Li, D., Halfond, W.G.J., Govindan, R.: Estimating mobile application energy consumption using program analysis. In: ICSE'13, pp. 92–101 (2013)
25. Harrell Jr, F.E., with contributions from Charles Dupont, many others.: Hmisc: Harrell Miscellaneous (2017). URL `https://CRAN.R-project.org/package=Hmisc`. R package version 4.0-3
26. Harter, D.E., Krishnan, M.S., Slaughter, S.A.: Effects of process maturity on quality, cycle time, and effort in software product development. Management Science **46**(4), 451–466 (2000)
27. Hecht, G., Moha, N., Rouvoy, R.: An empirical study of the performance impacts of android code smells. In: International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 59–69 (2016)
28. Hedges, L.V., Olkin, I.: Statistical Methods for Meta-Analysis. Academic Press (1985)
29. Herzig, K., Zeller, A.: The impact of tangled code changes. In: Working Conference on Mining Software Repositories (MSR), pp. 121–130 (2013)
30. Holm, S.: A simple sequentially rejective multiple test procedure. Scandinavian journal of statistics pp. 65–70 (1979)
31. Incerto, E., Tribastone, M., Trubiani, C.: Combined vertical and horizontal autoscaling through model predictive control. In: International Conference on Parallel and Distributed Computing (Euro-Par), pp. 147–159 (2018)
32. Jain, R.: The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. John Wiley & Sons (1990)
33. Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S.: Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices **47**(6), 77–88 (2012)
34. Joorabchi, M.E., Ali, M., Mesbah, A.: Detecting inconsistencies in multi-platform mobile apps. In: International Symposium on Software Reliability Engineering (ISSRE), pp. 450–460 (2015)
35. Joorabchi, M.E., Mesbah, A., Kruchten, P.: Real challenges in mobile app development. In: International Symposium on Empirical Software Engineering and Measurement (ESEM), pp. 15–24 (2013)

36. Jovic, M., Adamoli, A., Hauswirth, M.: Catch me if you can: performance bug detection in the wild. In: ACM SIGPLAN Notices, vol. 46, pp. 155–170 (2011)
37. Kan, S.H.: Metrics and models in software quality engineering. Addison-Wesley Longman Publishing Co., Inc. (2002)
38. Khalid, H., Nagappan, M., Hassan, A.E.: Examining the relationship between findbugs warnings and app ratings. IEEE Software **33**(4), 34–39 (2016)
39. Killian, C., Nagaraj, K., Pervez, S., Braud, R., Anderson, J.W., Jhala, R.: Finding latent performance bugs in systems implementations. In: International Symposium on Foundations of Software Engineering (FSE), pp. 17–26 (2010)
40. Kleinbaum, D.G., Klein, M.: Survival Analysis: A Self-Learning Text (2005)
41. Knoche, H., Eichelberger, H.: Using the raspberry pi and docker for replicable performance experiments: Experience paper. In: International Conference on Performance Engineering (ICPE), pp. 305–316 (2018)
42. Lampson, B.W.: Computer security in the real world. Computer **37**(6), 37–46 (2004)
43. Lee, S., Heo, M., Lee, C., Kim, M., Jeong, G.: Applying deep learning based automatic bug triager to industrial projects. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 926–931 (2017)
44. Li, D., Halfond, W.G.J.: Optimizing energy of http requests in android applications. In: International Workshop on Software Development Lifecycle for Mobile (DeMobile), pp. 25–28 (2015)
45. Li, D., Hao, S., Gui, J., Halfond, W.: An empirical study of the energy consumption of Android applications. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 121–130 (2014)
46. Li, D., Hao, S., Halfond, W.G.J., Govindan, R.: Calculating source line level energy information for android applications. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 78–89 (2013)
47. Li, D., Jin, Y., Sahin, C., Clause, J., Halfond, W.: Integrated energy-directed test suite optimization. In: International Symposium on Software Testing and Analysis (ISSTA), pp. 339–350 (2014)
48. Li, D., Lyu, Y., Gui, J., Halfond, W.G.: Automated energy optimization of http requests for mobile applications. In: International Conference on Software Engineering (ICSE), pp. 249–260 (2016)
49. Li, D., Tran, A.H., Halfond, W.: Making web applications more energy efficient for OLED smartphones. In: International Conference on Software Engineering (ICSE), pp. 573–538 (2014)
50. Li, D., Tran, A.H., Halfond, W.G.J.: Nyx: A display energy optimizer for mobile web apps. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 958–961 (2015)
51. Lin, Y., Radoi, C., Dig, D.: Retrofitting concurrency for android applications through refactoring. In: International Symposium on Foundations of Software Engineering (FSE), pp. 341–352 (2014)
52. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Oliveto, R., Penta, M.D., Poshyvanyk, D.: Mining energy-greedy API usage patterns in android apps: an empirical study. In: Working Conference on Mining Software Repositories (MSR), pp. 2–11 (2014)
53. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Penta, M.D., Oliveto, R., Poshyvanyk, D.: Multi-objective optimization of energy consumption of guis in android apps. ACM Trans. Softw. Eng. Methodol. **27**(3), 14:1–14:47 (2018)
54. Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C.E., Oliveto, R., Penta, M.D., Poshyvanyk, D.: Optimizing energy consumption of guis in android apps: a multi-objective approach. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 143–154 (2015)
55. Linares-Vásquez, M., Vendome, C., Luo, Q., Poshyvanyk, D.: How developers detect and fix performance bottlenecks in android apps. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 352–361 (2015)
56. Linares-Vásquez, M., Vendome, C., Tufano, M., Poshyvanyk, D.: How developers micro-optimize android apps. Journal of Systems and Software **130**, 1 – 23 (2017)
57. Liu, Y., Xu, C., Cheung, S.C.: Characterizing and detecting performance bugs for smartphone applications. In: International Conference on Software Engineering (ICSE), pp. 1013–1024 (2014)

58. Madan, B.B., Gogeva-Popstojanova, K., Vaidyanathan, K., Trivedi, K.S.: Modeling and quantification of security attributes of software systems. In: International Conference on Dependable Systems and Networks (DSN), pp. 505–514 (2002)
59. Martin, W., Sarro, F., Jia, Y., Zhang, Y., Harman, M.: A survey of app store analysis for software engineering. IEEE Transactions on Software Engineering **43**(9), 817–847 (2017)
60. Mazuera-Rozo, A., Bautista-Mora, J., Linares-Vásquez, M., Rueda, S., Bavota, G.: The android OS stack and its vulnerabilities: an empirical study. Empirical Software Engineering **24**(4), 2056–2101 (2019)
61. Mazuera-Rozo, A., Trubiani, C., Linares-Vásquez, M., Bavota, G.: Replication package. `https://github.com/amazuerar/perf-bugs-mobile/`
62. Mcilroy, S., Shang, W., Ali, N., Hassan, A.E.: User reviews of top mobile apps in apple and google app stores. Communications of the ACM **60**(11), 62–67 (2017)
63. Mondal, M., Roy, C.K., Schneider, K.A.: Bug propagation through code cloning: An empirical study. In: International Conference on Software Maintenance and Evolution (ICSME), pp. 227–237 (2017)
64. Nagappan, M., Shihab, E.: Future trends in software engineering research for mobile apps. In: International Conference on Software analysis, evolution, and reengineering (SANER), vol. 5, pp. 21–32 (2016)
65. Near, J.P., Jackson, D.: Finding security bugs in web applications using a catalog of access control patterns. In: International Conference on Software Engineering (ICSE), pp. 947–958 (2016)
66. Nistor, A., Chang, P.C., Radoi, C., Lu, S.: CARAMEL: detecting and fixing performance problems that have non-intrusive fixes. In: International Conference on Software Engineering (ICSE), pp. 902–912 (2015)
67. Nistor, A., Jiang, T., Tan, L.: Discovering, reporting, and fixing performance bugs. In: International Working Conference on Mining Software Repositories (MSR), pp. 237–246 (2013)
68. Noei, E., Syer, M.D., Zou, Y., Hassan, A.E., Keivanloo, I.: A study of the relation of mobile device attributes with the user-perceived quality of android apps. Empirical Software Engineering **22**(6), 3088–3116 (2017)
69. Oliveira, W., Oliveira, R., Castor, F.: A study on the energy consumption of android app development approaches. In: International Conference on Mining Software Repositories (MSR), pp. 42–52 (2017)
70. Olivo, O., Dillig, I., Lin, C.: Static detection of asymptotic performance bugs in collection traversals. In: ACM SIGPLAN Notices, vol. 50, pp. 369–378 (2015)
71. Panichella, S., Panichella, A., Beller, M., Zaidman, A., Gall, H.C.: The impact of test case summaries on bug fixing performance: an empirical investigation. In: International Conference on Software Engineering (ICSE), pp. 547–558 (2016)
72. Parsons, T., Murphy, J.: Detecting performance antipatterns in component based enterprise systems. Journal of Object Technology **7**(3), 55–91 (2008)
73. Pathak, A., Hu, Y., Zhang, M.: Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices. In: Workshop on Hot Topics in Networks (HotNets), p. Article No 5 (2011)
74. Pathak, A., Hu, Y., Zhang, M.: Where is the energy spent inside my app? fine grained energy accounting on smartphones with eprof. In: European Conference on Computer Systems (EuroSys), pp. 29–42 (2012)
75. Pathak, A., Hu, Y., Zhang, M., Bahl, P., Wang, Y.M.: Fine-grained power modeling for smartphones using system call tracing. In: European Conference on Computer Systems (EuroSys), pp. 153–168 (2011)
76. Pathak, A., Jindal, A., Hu, Y., Midkiff, S.P.: What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps. In: International Conference on Mobile Systems, Applications, and Services (MobiSys), pp. 267–280 (2012)
77. R., C.D.: Regression models and life tables. Journal of the Royal Statistic Society **B**(34), 187–202 (1972)
78. Ramakrishnan, R., Kaur, A.: Technique for detecting early-warning signals of performance deterioration in large scale software systems. In: International Conference on Performance Engineering (ICPE), pp. 213–222 (2017)

79. Rodríguez, R.J., Trubiani, C., Merseguer, J.: Fault-tolerant techniques and security mechanisms for model-based performance prediction of critical systems. In: International Symposium on Architecting Critical Systems, ISARCS, pp. 21–30 (2012)
80. Sahin, C., Wan, M., Tornquist, P., McKenna, R., Pearson, Z., Halfond, W.G.J., Clause, J.: How does code obfuscation impact energy usage? Journal of Software: Evolution and Process pp. 565–588 (2016)
81. Schulz, H., Okanovic, D., van Hoorn, A., Ferme, V., Pautasso, C.: Behavior-driven load testing using contextual knowledge - approach and experiences. In: International Conference on Performance Engineering (ICPE), pp. 265–272 (2019)
82. Selakovic, M., Pradel, M.: Performance issues and optimizations in javascript: an empirical study. In: International Conference on Software Engineering (ICSE), pp. 61–72 (2016)
83. Sliwerski, J., Zimmermann, T., Zeller, A.: When do changes induce fixes? In: Proceedings of the 2005 International Workshop on Mining Software Repositories (2005)
84. Syer, M.D., Jiang, Z.M., Nagappan, M., Hassan, A.E., Nasser, M., Flora, P.: Continuous validation of load test suites. In: International Conference on Performance Engineering (ICPE), pp. 259–270 (2014)
85. Syer, M.D., Nagappan, M., Hassan, A.E., Adams, B.: Revisiting prior empirical findings for mobile apps: An empirical case study on the 15 most popular open-source android apps. In: Conference of the Center for Advanced Studies on Collaborative Research, pp. 283–297 (2013)
86. Trivedi, K.S., Bobbio, A.: Reliability and Availability Engineering - Modeling, Analysis, and Applications. Cambridge University Press (2017)
87. Trubiani, C., Bran, A., van Hoorn, A., Avritzer, A., Knoche, H.: Exploiting load testing and profiling for performance antipattern detection. Information & Software Technology **95**, 329–345 (2018)
88. Wan, M., Jin, Y., Li, D., Halfond, W.G.J.: Detecting display energy hotspots in Android apps. In: International Conference on Software Testing, Verification and Validation (ICST) (2015)
89. Wan, Z., Lo, D., Xia, X., Cai, L.: Bug characteristics in blockchain systems: a large-scale empirical study. In: International Conference on Mining Software Repositories (MSR), pp. 413–424 (2017)
90. Wang, J., Dou, W., Gao, Y., Gao, C., Qin, F., Yin, K., Wei, J.: A comprehensive study on real world concurrency bugs in node.js. In: International Conference on Automated Software Engineering (ASE), pp. 520–531 (2017)
91. Wasserman, A.I.: Software engineering issues for mobile application development. In: International Workshop on Future of Software Engineering research (FSE/SDP), pp. 397–400 (2010)
92. Wert, A., Happe, J., Happe, L.: Supporting swift reaction: Automatically uncovering performance problems by systematic experiments. In: International Conference on Software Engineering (ICSE), pp. 552–561 (2013)
93. Williams, L.G., Smith, C.U.: Making the business case for software performance engineering. In: International Conference on Computer Measurement Group (CMG), pp. 349–358 (2003)
94. Woodside, C.M.: Wosp-c'15: Workshop on challenges in performance methods for software development. In: International Conference on Performance Engineering (ICPE), pp. 349–350 (2015)
95. Woodside, C.M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: International Workshop on the Future of Software Engineering (FOSE), pp. 171–187 (2007)
96. Yang, Y., Xiang, P., Mantor, M., Zhou, H.: Fixing performance bugs: An empirical study of open-source gpgpu programs. In: International Conference on Parallel Processing (ICPP), pp. 329–339 (2012)
97. Zaman, S., Adams, B., Hassan, A.E.: Security versus performance bugs: a case study on firefox. In: International Conference on Mining Software Repositories (MSR), pp. 93–102 (2011)
98. Zaman, S., Adams, B., Hassan, A.E.: A qualitative study on performance bugs. In: International Conference of Mining Software Repositories MSR, pp. 199–208 (2012)

99. Zhou, Y., Sharma, A.: Automated identification of security issues from commit messages and bug reports. In: Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 914–919 (2017)