How Developers Micro-Optimize Android Apps

Mario Linares-Vásquez, Christopher Vendome, Michele Tufano, Denys Poshyvanyk

PII: S0164-1212(17)30081-X
DOI: 10.1016/j.jss.2017.04.018
Reference: JSS 9953

To appear in: *The Journal of Systems & Software*

Received date: 23 November 2016
Revised date: 28 March 2017
Accepted date: 24 April 2017

Please cite this article as: Mario Linares-Vásquez, Christopher Vendome, Michele Tufano, Denys Poshyvanyk, How Developers Micro-Optimize Android Apps, *The Journal of Systems & Software* (2017), doi: 10.1016/j.jss.2017.04.018

Highlights

- Open source apps have a significant number of micro-optimization opportunities
- The practice of detecting and implementing micro-optimizations is not prevalent
- The impact of micro-optimizations might be noticeable under certain load conditions
- Removing unused resources can significantly reduce the memory consumed by apps

How Developers Micro-Optimize Android Apps

Mario Linares-Vásquez[1], Christopher Vendome[2], Michele Tufano[2], Denys Poshyvanyk[2]

[1] *Universidad de los Andes, Bogotá, Colombia*

[2] *The College of William and Mary, Williamsburg, VA, USA*

m.linaresv@uniandes.edu.co, {cvendome, mtufano, denys }@cs.wm.edu

Abstract

Optimizing mobile apps early on in the development cycle is supposed to be a key strategy for obtaining higher user rankings, more downloads, and higher retention. In fact, mobile platform designers publish specific guidelines, and tools aimed at optimizing apps. However, little research has been done with respect to identifying and understanding actual optimization practices performed by developers. In this paper, we present the results of three empirical studies aimed at investigating practices of Android developers towards improving the performance of their apps, by means of micro-optimizations. We mined change histories of 3,513 apps to identify the most frequent micro-optimization opportunities in 297K+ snapshots and to understand if (and when) developers implement these optimizations. Then, we performed an in-depth analysis into whether implementing micro-optimizations can help reduce memory/CPU usage. Finally, we conducted a survey with 389 open-source developers to understand how they use micro-optimizations to improve the performance of Android apps. Surprisingly, our results indicate that developers rarely implement micro-optimizations. Also, the impact of the analyzed micro-optimization on CPU/memory consumption is negligible in most of the cases. Finally, the results from the survey shed some light into why this happens as well as upon which practices developers rely upon.

Key words: Optimizations, Mining Software Repositories, Empirical Studies, Android, Measurement

1. Introduction

Every software developer has heard the famous quote by Sir Tony Hoare (promoted by Donald Knuth) that "*premature optimization is the root of all evil*". In fact, this quote can be easily taken to an extreme by developers, who tend to postpone optimizing their applications until the very end of the software development cycle. However, with the advent of mobile platforms, optimizing the quality of mobile apps *early* is becoming a key strategy for obtaining higher user rankings, more downloads (and revenue), and higher retention (i.e., longer install periods). Designers of mobile platforms outline various guidelines, such as Android performance tips [1] and best practices [2], aimed at helping developers optimize their mobile apps. Moreover, tools for debugging (e.g., Dalvik Debug Monitor Server [3]) and memory analysis (e.g., Eclipse Memory Analyzer [4]) are available to help developers detect optimization opportunities at run-time.

Additionally, a number of static analysis tools, such as *Lint* [5], *PMD* [6], *FindBugs* [7], *PerfChecker* [8], and *Relda* [9] have been designed to aid developers in optimizing their apps, by detecting micro-optimization opportunities (e.g., use `StringBuffer` instead of `String`). Micro-optimizations consist of changes to the source code that are applied mostly at statement level and are not intended to change the system design or architecture. For example, the micro-optimizations UnusedLocalVariable and UnusedPrivateMethod from PMD suggest removing unused code in order to reduce the memory footprint of the application. Other micro-optimizations aim at improving the performance; for example, the FloatMath micro-optimization from the tool Lint suggests replacing the type `android.util.FloatMath` with the type `java.lang.Math`, which is optimized better by the JIT compiler and provides better performance when performing numerical operations.

Some of the tools for detecting micro-optimizations opportunities have been widely adopted by software developers and companies building Java and C/C++ systems [10]. However, the effectiveness of the warnings provided by these tools

has been previously questioned in the context of fault detection in C/C++ and Java systems [11, 12, 13, 14, 15]. Moreover, little research has been done into understanding whether mobile developers perform micro-optimizations, given the fact that mobile apps are more prone to performance bugs [8].

In this paper, we present the results of three empirical studies aimed at understanding the usage of micro-optimizations in the wild, the reasons, and their impact when optimizing Android apps. First, we measured the persistence of micro-optimization opportunities[1] in change histories of open source Android apps; in particular, we mined the change histories of 3,513 Android apps hosted on GitHub to identify the most frequent micro-optimization opportunities in 297K+ snapshots of these apps (at commit level granularity) and to understand if (and when) developers implement these micro-optimization opportunities. Second, we analyze the effectiveness of micro-optimizations suggested by two static analysis tools (i.e., *PMD* and *LINT*) on the resource consumption of Android apps by means of an in-depth analysis into whether implementing micro-optimizations can help reduce memory and CPU usage in a sample of eight Android apps. Finally, we investigated current practices of Android developers for improving the performance of their apps by conducting a survey involving 389 open-source Android developers to understand the state of practice with respect to micro-optimizing apps and the reasons for applying (or not) micro-optimizations in practice.

Our findings suggest that although open source Android apps have a significant number of micro-optimization opportunities throughout their change histories, developers tend to disregard them (Section 4). Regarding the value (i.e., impact on resource consumption) of micro-optimizations, we found that, from the analyzed micro-optimizations, removing unused resources is the only

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

one that can actually have a large and significant improvement on the memory usage in eight open-source Android apps used in the study (Section 5). Also, the lack of micro-optimization implementations might be explained by our survey results (Section 6) that indicate that (i) most of the developers do not know about micro-optimizations, (ii) they consider the apps too simple to require optimizations, (iii) they do not consider micro-optimizations as a worthwhile investment of their time, or (iv) they do not believe that implementing those optimizations will have any impact on the performance of Android apps.

These results contribute to understanding the nature of Android apps and the need for specific practices that consider the context of the apps (i.e., device or OS); the theoretical assumption that practices with certain success in non-mobile systems can be transferred to mobile apps with the same results is not totally valid as in the case of micro-optimizations, which require specific conditions that are not available in mobile apps (e.g., large number of String operations or a large number of instantiations of objects in loops). One limitation of our study is that Android-specific micro-optimizations for the GUI were not included, because they can not be automatically performed and their implementation is always app-specific. Therefore, this paper represents a "starting point" for future efforts in terms of validating the impact of existing Android-specific micro-optimizations and providing developers with micro-optimizations that can have real impact on the performance of their apps.

Paper organization. Section 2 reports on the previous studies analyzing effectiveness of static analysis tools for detecting micro-optimizations and bugs, and previous work on improving resource consumption of Android apps. Section 3 introduces the three empirical studies. Section 4 describes the mining-based study and presents the corresponding results. In Section 5, we present the design of our study for measuring impact of micro-optimizations on Android apps and discuss our findings. Section 6 reports on the survey, participants, and analysis of the responses. Finally, Section 7 draws the conclusions and outlines some future work.

2. Related Work

This paper is mostly related to previous work on the analysis of source code changes that reduce resource consumption. However, because we rely on static analysis tools, we also summarize the studies aimed at verifying the effectiveness of tools, such as *FindBugs* and *PMD*. Other papers have analyzed several aspects of performance bugs [16, 17, 18, 19, 20]; however, none of them are Android-specific.

2.1. Effectiveness of Tools for Static Analysis

Static analysis tools such as *FindBugs* [7], *PMD* [6], and *LINT* [5] evaluate source code statically and provide developers with a list of warnings (a.k.a., violations or checks) that could help improve the software quality [12, 15]. In particular, the effectiveness of the reports generated by these tools and the relationship between the reported issues and fault-proneness has been analyzed previously. While Ayewah *et al.* [15] report successful industrial experience with *FindBugs* detecting defects in Google's code base and Sun Microsystems' JDK 1.6, Wagner *et al.* [21] show that field defects were not detected by *FindBugs* and *PMD* on two sale support systems. However, the results by Wagner *et al.* [21] suggest that classes with warnings generated by both tools are more likely to be fault-prone than classes without warnings generated by either of the tools.

Concerning the precision of these static analysis tools, several studies determined that only a limited set of the reported warnings have high precision and that there is a high ratio of false positives [11, 12, 13, 14]. Wagner *et al.* [11] analyzed *FindBugs*, *PMD*, and *QJPro* [22] in the context of three industrial systems and one small academic system; Zeng *et al.* [12] considered the results of using *FlexeLint*, *Klockwork*, and *Reasoning's Illuma* on three products of Nortel Networks; and, Vetro *et al.* [13, 14] focused on analyzing 301 projects from an object-oriented programming class with *FindBugs*. *FindBugs* and *PMD* were also investigated by Araujo *et al.* [23] in a study with five releases of Eclipse and 12 open source systems; the main findings of the study are that the rate of false

positives generated with *FindBugs* can be reduced drastically, when developers select categories that are relevant to the target system. The authors also found the rate of true positives generated with *PMD* to be low for the Eclipse releases.

Previous studies have also used the reported warnings as a proxy for software quality [24, 25, 26]. In a recent paper by Khalid *et al.* [27], the authors analyzed the relationship between warnings generated by *FindBugs* and the user ratings of Android apps, and the results suggest that warnings from the Bad Practice, Internationalization, and Performance categories have significantly higher densities in low-rated apps than high-rated apps.

2.2. Improving Resource Consumption

Optimization and profiling of resource consumption in the Android platform (i.e., apps, operating system, and hardware) has focused mostly on improving the operating system and hardware components (*e.g.*, [28, 29]). The most targeted resource is energy consumption in the app layer, which is currently considered to be a significant research problem. On one hand, several studies describe energy bugs [30, 31, 32, 33], energy greedy APIs [34, 35], energy leaks [36], and Android code smells [37]; on the other hand, a plethora of frameworks and methods have been proposed for energy profiling and optimization [38, 39, 40, 41, 34, 42, 43]. Other studies interested in the impact of source code changes on energy consumption – not specifically focused on mobile apps – are the "green mining" family of papers by Hindle *et al.* [44, 45, 42] and the study by Sahin *et al.* [46] on the impact of refactorings. Recently, similar studies have been performed on Android repositories, which have been mined with the aim of detecting UI performance regressions [47] and the impact of performance-directed changes to the source code [48].

Only a few works focused on studying or improving performance of Android apps. For instance, Liu *et al.* [8] proposed an approach based on static analysis called *PerfChecker* for detecting performance bugs in Android apps, whereas Guo *et al.* [9] developed an Android specific approach for detecting resource leaks – also using static analysis. The former aims at detecting performance

bugs (i.e., GUI lagging, energy leak, memory bloat) represented by patterns, while the latter focuses on resource leaks related to operations with hardware-components such as sensors, camera, and speakers. In addition, a recent paper by Lin *et al.* [49] presents an approach for automatically refactoring Android asynchronous tasks in Android apps to reduce performance bugs. Conversely to [8, 9, 49], we do not propose an approach for detecting opportunities in order to improve performance of Android apps; however, we present and analyze the results of three empirical studies aimed at identifying actual developers' practices for improving the performance of Android apps by means of micro-optimizations.

Besides the aforementioned papers, three other papers are related to ours: (i) the study by Sahin *et al.* [50] that measures the impact of different obfuscation tools on the energy consumption in Android apps, (ii) the work by Gui *et al.* [51] on the cost of ads in mobile software, and (iii) our previous work on surveying the tools and practices used by Android developers to detect and fix performance bottlenecks [20]. Sahin *et al.* [50] found – in 15 usage scenarios from 11 Android apps - that obfuscated versions of the apps consume more energy. Moreover, Gui *et al.* [51] – in a study with 21 Android apps – showed that ads embedded in Android apps have higher impact on resource consumption, in particular bandwidth and CPU. In our prior work [20], we conducted a survey of Android developers identifying the tools and practices that they employ to detect performance bottlenecks. Additionally, in [20], we investigated actual issues related to such bottlenecks to identify the way in which developers fix specific instances of performance bottlenecks.

In summary, compared to the previous work, this paper focuses on micro-optimizations for mobile apps and their impact on CPU and memory consumption. We analyze such phenomena in three studies: (i) mining the evolution of micro-optimization opportunities (as reported by static analysis tools) in the change history of Android apps, (ii) measuring the reduction in terms of memory and CPU used by Android apps after implementing micro-optimizations in a subset of eight apps, and (iii) surveying developers on their practices with

respect to micro-optimizing Android apps.

3. Overview of the Empirical Studies

The main *goal* of this paper is to identify whether Android developers use micro-optimizations to improve the performance of mobile apps, their reasons, and the real impact of micro-optimizations on resource consumption. We define micro-optimizations as source code changes (often performed at statement level) that are intended to improve the performance or reduce resource consumption (i.e., memory and CPU). The micro-optimization opportunities in code, which were analyzed in this paper, are based on warnings/violations detected by tools such as *PMD* and *LINT*. It is worth noting that these tools report warnings related to several categories, such as correctness, security, style, and performance among the others. However, our focus is only on performance and resource consumption (i.e., micro-optimization opportunities); thus, we only considered the warnings reported under the *Unused Code*, *String*, *Optimization*, and *Migration* categories from *PMD*, and the *Performance* category from *LINT* (see Table 1 for a subset of studied micro-optimizations). The complete list of micro-optimizations is detailed in our online appendix [52]; Table 1 also lists with * the micro-optimizations selected for the second study[2].

For our analysis, we considered three different perspectives. Firstly, we wanted to analyze the persistence of micro-optimization opportunities through the evolution of the Android apps from GitHub (i.e., trends) and to determine which of those opportunities are taken into account by Android developers. In particular, we selected GitHub as the datasource, since it has the largest set of open source Android apps (16,331 apps as of February 2014), including their change history at commit level (2,063,523 commits as of February of 2014). The second perspective is from the point of view of resource consumption in order to determine the real impact of micro-optimizations on the memory and

[2]More details about the selection process for Study 2 are provided in Section 5.

Table 1: A Subset of the list of micro-optimizations opportunities used in the studies and tools to identify them. The micro-optimizations marked with * were used in the second study.

Micro-optimization	**Category**	**Tool**
UnusedModifier	Unused Code	PMD
UnusedPrivateField	Unused Code	PMD
UnusedLocalVariable	Unused Code	PMD
UnusedPrivateMethod	Unused Code	PMD
AvoidDuplicateLiterals*	String	PMD
UseEqualsToCompareStrings*	String	PMD
StringInstantiation*	String	PMD
InefficientEmptyStringCheck*	String	PMD
UseIndexOfChar*	String	PMD
StringToString*	String	PMD
AppendCharacterWithChar*	String	PMD
ConsecutiveLiteralAppends*	String	PMD
SimplifyStartsWith*	Opt./Mig.	PMD
RedundantFieldInitializer*	Opt./Mig.	PMD
UseArrayListInsteadOfVector	Opt./Mig.	PMD
UnnecessaryWrapperObjectCreation*	Opt./Mig.	PMD
ReplaceVectorWithList	Opt./Mig.	PMD
ReplaceHashtableWithMap	Opt./Mig.	PMD
UseValueOf*	Performance	Lint
FloatMath*	Performance	Lint
UnusedResources*	Performance	Lint
UnusedNamespace*	Performance	Lint

CPU consumption in eight subject apps. Finally, for the third perspective, we identified the current practices of Android developers "in the wild", in particular the usage (and rationale) of micro-optimizations for improving the performance of mobile apps; we collected this information with an online survey targeting 24K+ contributors of 16,331 Android apps at GitHub.

We designed three empirical studies to address these three perspectives. The details on the data collection, research questions, and analysis method for each of the studies are presented in the following sections (i.e., Sections 4 to 6). Additional material regarding the design of the three studies is provided in our online appendix [52].

4. The First Study: Micro-Optimizations in Android Apps on GitHub

The first study focused on the evolutionary perspective of performance-related micro-optimizations in Android apps. The *goal* of this study is to (i) measure the frequency of the micro-optimization warnings — by rule and by violation category — reported by *PMD* and *LINT*, and (ii) identify the trends that describe the persistence of these micro-optimization warnings through the commit history of a large set of Android apps. The *context* consists of 3,513 open source Android apps and 56 types of warnings for micro-optimization opportunities detected by *LINT* and *PMD*. The *perspective* is of researchers interested in analyzing the usage of micro-optimizations in Android apps. Consequently, we formulated the following research questions:

RQ$_1$: *What is the distribution of micro-optimization warnings across Android Apps at GitHub?*

RQ$_2$: *To what extent are micro-optimizations introduced or avoided during the evolution and maintenance of Android apps?*

4.1. Analysis Method

To answer **RQ**$_1$ and **RQ**$_2$, we extracted Android projects on GitHub by first identifying all Java projects (381,161) through GitHub's API and locally

cloning the projects. We searched the local repositories for the presence of an *AndroidManifest.xml* file, required by Android apps. We refined this list to projects where the Android manifest file was at the top-level directory of the repository. While it filtered out certain projects, it removed the bias of single repositories that house multiple apps or multiple versions of the same app, which would inflate the number of violations. To both select active/real projects and to prevent duplicate projects, we filtered projects that had at least one fork, star, or watcher and that were not a fork themselves. After filtering, we found 16,331 potential repositories from which we randomly sampled 3,513 projects due to prohibitively high running time of applying static analysis tools at commit-level granularity.

With the projects locally cloned, we utilized *LINT* [5] and *PMD* [6] to identify micro-optimization opportunities. We chose *LINT* as it is incorporated into Android Studio for the purpose of improving Android apps [53]. Similar to *LINT*, *PMD* performs static analysis that identifies regions of code that do not conform to best programming practices. In total, we detected 56 types of warnings (24 from *LINT* and 32 from *PMD*) and a subset is outlined in Table 1. Each category is predefined in the original documentation for *LINT* [54] or *PMD* [55]. We also grouped the projects according to the quartiles of commit distribution to analyze similar projects in terms of the size of their history: projects with a number of commits less than the first quartile, Q1, (group-low), projects in the interquartile range (group-med), projects with more commits than the third quartile, Q3, (group-high), and the entire set of projects (group-all). Additionally, we excluded all the projects with less than five commits, since we deemed them to have insufficient change history. It is worth noting that the paper is not focusing on the differences between successful and non-successful apps; thus, we do not group the considered app by success-related measurements such as the ratings that are available in markets. In addition, analyzing apps with ratings would require downloading and decompiling apps from markets, which imposes some issues because decompilation to Java source code discards the Android project structure [56], which is required by *LINT* to

Table 2: Top Five Micro-Optimization Opportunities and Raw Frequency

Micro-Opt. Opportunity	Count	Norm. by LOC	Norm. by #Files
MethodArgumentCouldBeFinal	932,180	155.12	13,908.60
LocalVariableCouldBeFinal	640,574	113.69	10,184.00
UnusedResources	35,596	11.81	836.50
UnusedIds	27,817	10.90	788.10
AvoidInstantiatingObjectsInLoops	26,107.00	4.43	420.30

detect micro-optimization opportunities.

For $\mathbf{RQ}_1$, we used descriptive statistics, and the results are presented in section 4.2. The analysis method for $\mathbf{RQ}_2$ consisted of identifying the trends by finding the function that best describes (i.e., best fit) the evolution of the number of micro-optimization opportunities for each project. The function fitting was implemented in MATLAB targeting a diverse set of functions to avoid bias towards widely analyzed models (e.g., linear and power); in particular, for the fitting, we used the following functions available with MATLAB: linear, quadratic, cubic, exponential, power, logarithmic, Weibull, sine, and first order Fourier series model (fourier1). Then, we identified for each group of projects (i.e., group-low, group-med, group-high), the top behaviors (e.g., decreasing exponentially) at different levels: global (all the micro-optimization opportunities), category, and single warnings. In our results, we annotate the behaviors as a subscript to the function; for example, $power_+$ represents an increasing power function and $power_-$ represents a decreasing power function. In addition to the absolute values (i.e., number of warnings), we analyzed the density of micro-optimization opportunities by normalizing the frequencies by the number of files and lines of code (LOC) in order to avoid the impact of size as a confounding factor. The results for $\mathbf{RQ}_2$ are presented in section 4.3.

4.2. Results for RQ_1: Top Micro-Optimization Opportunities in Apps at GitHub

To understand the most prevalent micro-optimization opportunities, we considered the most recent snapshot of the 3,513 Android apps in our dataset. Table

2 shows the top five most common performance-related warnings (detected by *LINT* and *PMD*) by raw counts, normalized by LOC ($\sum_{p=1}^{3,513} \frac{\#warnings_p}{LOC_p}$, where p corresponds to a project in the dataset), and normalized by the number of files ($\sum_{p=1}^{3,513} \frac{\#warnings_p}{\#files_p}$, where p corresponds to a project in the dataset). The normalization aims to avoid the impact of large systems "inflating" the frequencies (i.e., avoiding a small number of large projects biasing the results to a particular micro-optimization opportunity). The top most prevalent warning was MethodArgumentCouldBeFinal with 932,180 occurrences. This micro-optimization describes arguments passed to a method that are not changed within the method body. The second most prevalent micro-optimization is LocalVariableCouldBeFinal with 640,574 occurrences. In this case, the warning suggests that variables that do not change in their local scope can also be made final, since their values are detected as constants. These two micro-optimizations would likely allow compilers to better optimize the code when generating byte-code, since the `final` keyword signifies that these values do not change. However, the value of improving these violations depends upon the compiler's ability to optimize the generated byte-code. For instance, the response to the Stack Overflow's question 4279420 suggests that the impact of these micro-optimizations is rather negligible [57].

UnusedResources was the third most prevalent micro-optimization opportunity with 35,596 occurrences. This micro-optimization is present when the project counts resources (i.e., non-source code such as image files) that are not utilized in the application. UnusedIds was the fourth most frequent micro-optimization with 27,817 occurrences. Resources in Android apps are accessed from the source code by referencing the ids used to declare the resources [58]. Therefore, UnusedIds refers to resource ids that are not referenced in source code.

Finally, AvoidInstantiatingObjectsInLoops was the fifth most prevalent micro-optimization with 26,107 occurrences, and it relates to creating new objects within a loop body that could be declared outside of the loop and reused. Thus, removing this type of micro-optimizations avoids creating unnecessary

or duplicate objects inside loops which could impact memory usage.

Table 3 shows the top categories of micro-optimization opportunities in descending order of frequency and normalized by LOC (i.e., frequency divided by LOC) and number of files (i.e., frequency divided by the number of files). When considering the categories, we found that warnings labeled as *Optimization* by PMD and the warnings corresponding to the performance category by *LINT* were the most prevalent in our dataset. The *Optimization* category has substantially more warnings in the analyzed apps than any other category by at least one order of magnitude. While there are 2.1 times more LINT warnings than String warnings, the density of warnings is 4.4 times larger when the data was normalized by LOC and 3.42 larger when normalized by the number of files. We also observe a smaller difference between the frequency of *String* warnings as compared to the frequency of *UnusedCode* violations (1.84 times for raw frequency, 1.3 times for normalized by LOC, and 1.4 times for normalized by the number of files). *Migration* is the warnings category with smallest frequency by an order of magnitude when compared to the previous two categories; however, this observation is expected since this category groups the fewest individual violations.

Summary for RQ$_1$. We observe a large number of unaddressed micro-optimization opportunities in Android open source apps. When analyzing the most recent snapshot of 3,513 open source Android apps, we found a large number of micro-optimizations opportunities (detected by *PMD* and *LINT*) as reflected in the raw frequencies. Importantly, we observed that the density of these warnings (i.e., normalized by the number of files and LOC) reflected the high prevalence of micro-optimizations opportunities, since the densities mirrored the results for the raw data. The top three warnings (i.e., MethodArgumentCouldBeFinal, LocalVariableCouldBeFinal, UnusedResources) amount to over 1.5 million instances, where micro-optimizations might be implemented by developers of open source Android apps.

Table 3: Frequencies of Micro-Optimization Opportunities by Category

Category	#Rules	Raw count	Norm. by LOC	Norm. by #Files
Optimization (PMD)	10	1,616,022	276.95	24,852.90
Performance (LINT)	24	82,463	27.59	2,002.99
String (PMD)	15	39,663	6.31	584.60
UnusedCode (PMD)	4	21,485	4.83	428.56
Migration (PMD)	3	4,002	0.58	56.11

4.3. Results for RQ_2: Persistence of Micro-Optimization Opportunities in Change Histories of Android Apps

While the previous question investigated the number of micro-optimization opportunities in the current releases, we also sought to understand the extent to which (if at all) developers address the micro-optimizations across the projects' change histories. We analyzed change histories of the 3,513 projects to understand if micro-optimizations were implemented. We detected the models (*i.e.*, $y = f(x)$, where y is the number of micro-optimization opportunities, and x is the commit number) that fit the evolution of the micro-optimization opportunities and the behaviors (positive or negative), when applicable (as described in Section 4). The models are helpful to better understand the behavior in terms of the rate of which these micro-optimizations are being introduced and addressed. For example, an exponential positive model would indicate that the number of micro-optimizations is rapidly increasing. We also partitioned the apps by the number of commits to investigate the impact of the change history length.

The warnings reported by the *PMD* and *LINT* tools represent micro-optimization opportunities. In that sense, if the number of warnings detected in release k is reduced in release $k + 1$, we assume that the developers have implemented the micro-optimizations; otherwise, if the number of warnings increases for release $k + 1$, we assume that new code has been introduced with more opportunities for micro-optimizations. Consequently, the behavior of developers in terms of introducing or implementing micro-optimizations opportunities is represented

as the number of warnings detected by the tools across the whole history of the considered systems.

It is possible that the number of micro-optimization opportunities is reduced from release k to release $k+1$ because the code unit containing the warnings (e.g., class or method) was deleted with refactoring, cleaning, or optimizations purposes. Also, it is worth noting that the warnings considered in this study are only the ones related to performance and we filtered the list to avoid warnings that reported false positives. In the following, we describe our findings highlighting the top behaviors identified at category level and globally.

4.3.1. Highlights

In this subsection, we provide key insights from our analysis for **RQ$_2$**; however, subsections 4.3.2 and 4.3.3 provide a more in-depth analysis of the results.

Finding 1 A first order *fourier* model (i.e., $y = a_0 + a_1 \cdot cos(x \cdot p) + b_1 \cdot sin(x \cdot p)$) most frequently best fit the data when considering both global data and data for each category of micro-optimization opportunities.

Finding 2 Micro-optimization opportunities are prevalent during the maintenance and evolution of Android apps, as we found that only 12 projects never had a warning of any type introduced during their revision history.

Finding 3 The most prevalent behaviors suggest that developers mostly do not implement micro-optimizations, since micro-optimization opportunities either increase during the app history (i.e., more micro-optimization opportunities are being introduced during the development of the apps) or are constant (i.e., they are introduced, but neither increase nor decrease).

4.3.2. Models by Micro-Optimization Opportunity Category

Figure 1 shows the top-3 best fitting functions for our dataset for each category and for different project sizes. Additionally, we provide the frequency and illustrative behavior of the function. For example, the cell (Figure 1) in column

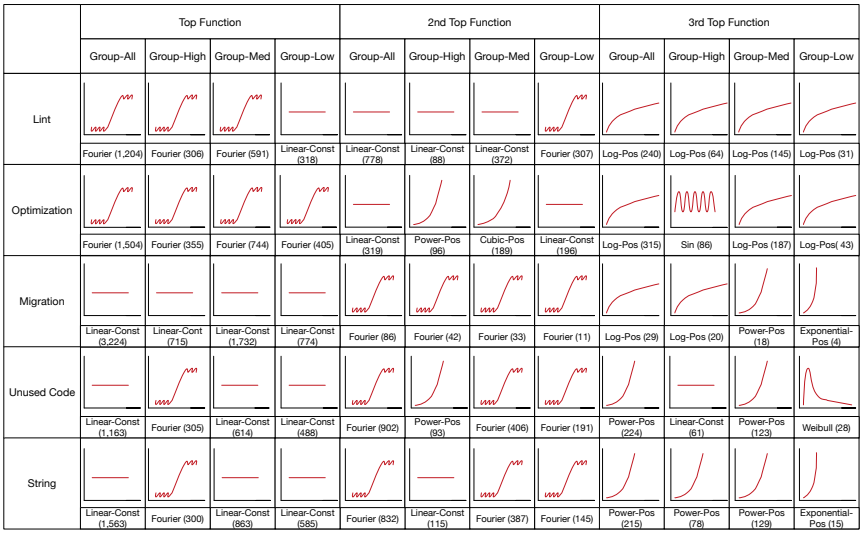


Figure 1: Top-3 best fitting for each type of micro-optimization opportunity and project size in our dataset. Graphs represent the behavior of the function and the number after the function name represents the frequency that function was found for the given optimization and project size.

Top-function/Group-All and row Lint shows a plot representing the behavior of the *fourier1* function and the label underneath has the frequency in parentheses.

Top-1 Functions: We found that the *fourier1* (i.e., $y = a_0 + a_1 \cdot cos(x \cdot p) + b_1 \cdot sin(x \cdot p)$) and *linear-constant* (i.e., $y = k$) models are the two most prevalent functions describing the behavior of micro-optimization opportunities in our entire dataset.

For the warnings in the *Optimization* category, we observed that the *fourier1* model, best fits the data for all groups (low, med, high, and all), for both normalized and raw values. In terms of the raw number of warnings for the *Optimization* category, we observed 1,504 projects for group-all, 405 projects for group-low, 744 projects for group-med, and 355 projects for group-high. For group-all's *Optimization* data (raw frequency), we observed an average coeffi-

cient of determination $R^2 = 0.89 \pm 0.11(\sigma)$. We omit the R^2 values for group-low, group-med, and group-high for each warning category, because they are similar in value and range. However, the complete results can be found in our online appendix [52].

The *fourier1* model also best fits the *LINT* violations for both raw and normalized data for group-med, group-high, and group-all. For group-all's raw data, we observed an average $R^2 = 0.83 \pm 0.12$. In terms of the raw number of warnings for the *LINT* category, we observed 1,204 projects for group-all, 591 projects for group-med, and 306 projects for group-high. For group-low, *fourier1* model fits the normalized data, but the raw values were fit by *linear-constant* with an average $R^2 = 1$ and accounted for 318 projects

Conversely, the *linear-constant* model fits *String* violations for group-low (585 projects for raw frequency), group-med (863 projects for raw frequency), and group-all (1,563 projects for raw frequency) in terms of both raw and normalized data. In all these cases, we observed an average $R^2 = 1$. However, group-high (300 projects for raw frequency) was fit by the *fourier1* model for both raw and normalized values. For instance, Figure 2.a depicts an example of *String* warnings for the project `ActiveAndroid`, which is fit by a *fourier1* model. The figure shows a period of quick growth followed by small fluctuations in the number of raw warnings of this category. This behavior suggests that there are periods where a large number of micro-optimization opportunities are introduced that are followed by a period of minor additions and deletions. Interestingly, we observe the *fourier1* model also best fits the data when normalized by LOC as seen in Figure 2.b. This suggests that the number of *String* violations for `ActiveAndroid` fluctuates in a similar way with the ratio of violations to LOC is relatively similar (i.e., the ratio may not be constant, but the general behavior for raw and normalized data is the same).

For the *Migration* category, the representative function is *linear-const* (i.e., $y = k$), which represents a history with no change in warnings of that particular type, for all groups. In terms of the raw number of warnings for the *Migration* category, we observed 3,224 projects for group-all, 774 project for group-low,

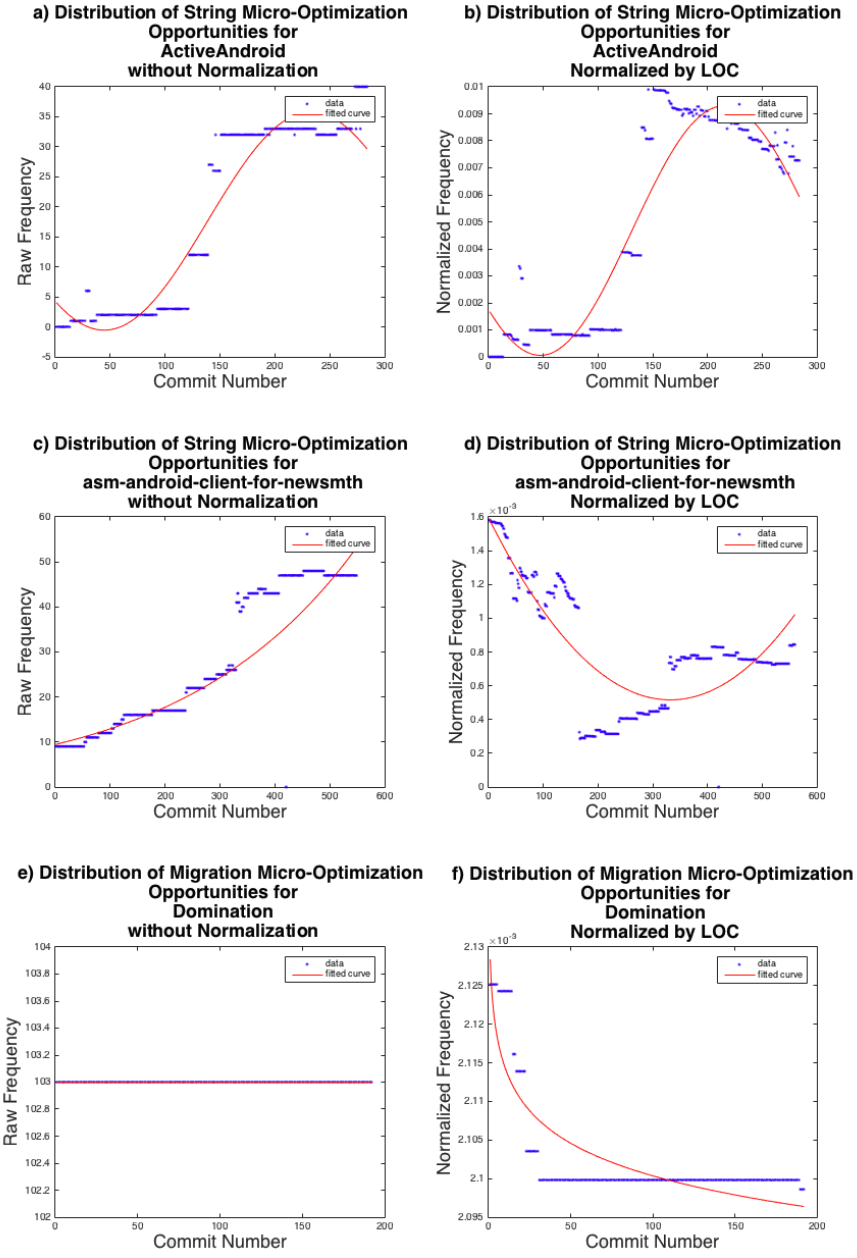


Figure 2: Persistence of micro-optimization opportunities in three Android apps. The frequencies are presented as raw data and normalized by LOC. The function fitting the data is presented in red color.

1,732 projects for group-med, and 715 projects for group-high. In these cases, we observed an average $R^2 = 1$. The prevalence of this function indicates that some Android apps have a consistent number of warnings from the same category (*Migration* in this case) that are not fixed during the app's history. For instance, Figure 2.e presents *Migration* warnings for the project `Domination`, which have a linear-constant behavior with 103 warnings introduced with the initial commit. While the number of raw warnings are constant, it shows that density by LOC is decreasing (Figure 2.f). Thus, the developers continually added new code, but it did not introduce any new *Migration* warning.

Top-2 Functions: When considering the second tier of functions that fit the data (i.e., top-2 frequent), we find that *fourier1* and *linear-constant* were still dominant functions. The *fourier1* model fits *Migration, UnusedCode,* and *String* warnings for group-low, group-med, and group-all for both raw and normalized values; it also fits *Migration* for group-high. For both raw and normalized data, *linear-constant* best fits *LINT* for group-med and group-all, *Optimization* for group-low. *LINT*'s group-high was also fit by *linear-constant*, except when normalized by the LOC (in that case, it was fit by $power_-$). We found that the $cubic_+$ (i.e., $y = a \cdot x^3$), $power_+$ (i.e., $y = a \cdot x^b$), $power_-$, *sine* (i.e., $y = a \cdot sin(b \cdot x + c)$), and *Weibull* also became prevalent distributions of both the raw frequency of micro-optimization opportunities and normalized frequencies in the dataset ($power_-$ only appears when normalizing).

$Cubic_+$ best fits group-med for the *Optimization* warnings' raw data. $Power_+$ fits group-high for *Optimization* and *UnusedCode* violations' raw data. The *sine* function best fits the normalized data for group-med, group-high, and group-all for *Optimization* warnings. *Sine* also fits group-high for *UnusedCode*, when normalized by LOC, while *Weibull* fits when normalized by the number of files. The *sine* function suggests that the number of warnings behaves in a wave-like fashion. Since it corresponds to normalized values, this behavior suggests that for certain projects micro-optimization opportunities are introduced periodically during particular points in an app's development lifecycle. Since most of these functions are positive, it indicates that the frequency of micro-optimization op-

portunities is increasing as the apps continually evolve. However, we observed a *power*_ distribution in group-high for *LINT* normalized by LOC, which suggests that prevalence of warnings per LOC is decreasing despite new code being added. For example, the `MidWorx` app is best fit by $exponential_+$ for raw *LINT* micro-optimization opportunities as well as normalized by LOC; it is also fit by *Weibull*, when normalized by the number of files. Thus, the *LINT* warnings are consistently increasing during the system's development.

Top-3 Functions: At the third tier of most prevalent functions, we observe group-high for *UnusedCode*'s raw data was the only one described by *linear-constant*, with 61 projects. The *logarithmic* function ($y = a{\cdot}log(x){+}b$), log_+, fits *Optimization* warnings for group-med and group-all in terms of raw (187 projects for group-med and 315 projects for group-all) and normalized data. Group-high was also fit by log_+ for normalized data, while group-low was fit by it for raw data with 43 projects. It also fits *LINT* warnings for the raw data of all groups: 240 projects for group-all, 31 projects for group-low, 145 projects for group-med, and 64 projects for group-high. It also best fits *Migration* for group-high and group-all in terms of raw data with 20 projects and 29 projects, respectively. Log_- fits *UnusedCode* for all groups, when the data was normalized by LOC. *Migration* violations were also best fit by log_- in group-high and group-all for normalized data; group-med was also fit by it, but only when normalized by LOC. *String* warnings were also represented by this function, when normalized by LOC, for group-low and group-all.

We also observed $exponential_+$ as the representative function for *Migration* and *String* in in group-low for raw data. In Figure 2.c, we observe the behavior of an $exponential_+$ ($y = a \cdot e^{b \cdot x}$) model of *String* warnings for the `asm-android-client-for-new- smth` app. It demonstrates the sharp increase similar to the $fourier1$ function example; however, it is not wave-like, which suggests a consistent increase in violations. The normalized value is represented by a *fourier1* function, which shows that density of decrease during the app's early evolution, but increases midway in the app's development. This behavior means that the lines of the code increases as the number of *LINT* warnings increases,

but at a lower rate. Conversely, $exponential_-$ fits *LINT* warnings for group-high, when the data was normalized by LOC. For example, the *LINT* warnings for the `memory_hog_android` app is fit by $exponential_-$ when normalized by LOC and raw warnings. The behavior suggests that the system is experiencing rapid growth while the performance is simultaneously being improved, since both the density (i.e., normalized *LINT* warnings) and raw number of *LINT* warnings are rapidly decreasing. These two distributions (i.e., $exponential_+$ and $exponential_-$) suggest a rapid increase or decrease in warnings. In the former scenario (i.e., $exponential_+$), the developers are unlikely to consider the impact of micro-optimizations on the app's performance, while the latter (i.e., $exponential_-$) suggest a large refactoring to improve the system's performance.

$Cubic_+$ fits *Migration* warnings for group-low and *String* warnings for group-med, when the data for both groups was normalized by LOC. In terms of the raw data, $power_+$ best fits *String* violations in group-med, group-high, and group-all as well as *UnusedCode* and *Migration* for group-med, and *UnusedCode* group-all. The *sine* function best fits *Optimization* violations in group-high for the raw data. For the data normalized by LOC, *sine* also fits the *Optimization* and *LINT* violations for group-low and *String* violations for group-high. When normalized by the number of files, the *sine* function fits *LINT* violations for all groups, *String* violations for group-low, group-high, and group-all, and *Migration* and *Optimization* warnings for group-low. The *Weibull* function fits the remaining data: *UnusedCode* warnings for group-low (raw data and normalized by the number of files), for group-med and group-all (data normalized by the number of files); *String* and *Migration* for group-med (data normalized by the number of files); and *LINT* for group-med and group-all (data normalized by LOC). For the 28 *UnusedCode* projects that were best fit by *Weibull* in group-low, 9 of the projects have a strictly decreasing Weibull behavior and 19 of the projects have a Weibull function with a peak (as the one in Fig. 1). This behavior suggests that either early on (in the strictly decreasing case) or after a period where these warnings were introduced (in the case of a peak) developers implemented micro-optimizations to reduce unused code.

In general, we observed that the top-1 and top-2 functions were predominantly *fourier* and *linear-constant*. These behaviors demonstrate either a consistent number of micro-optimization opportunities (*linear-constant*) despite the evolution and maintenance of the system, or it demonstrates a period of small wave-like fluctuations followed by a sharp change (generally increasing) with more wave-like fluctuations (*fourier*). In the former case (i.e., *linear-constant*), developers are neither introducing nor taking care these potential micro-optimizations. In the latter case (i.e., *fourier*), developers periodically address a small number of these warnings but typically introduce substantially more at a point in the project's development. For top-3, we typically observe a positive increasing behavior without waves. This behavior suggests that developers are consistently introducing more opportunities for micro-optimizations or the developers are introducing them at a sufficiently faster rate than the developers are removing them.

4.3.3. Global Behavior

While we are interested in looking at the functions fitting the different types of warnings, we also wanted to investigate the extent to which the functions fit the data globally, (i.e., across all the categories). The top five global models for raw frequencies are positive, but the normalization by LOC suggests a logarithmic decline; this observation demonstrates that the micro-optimizations opportunities density is decreasing with the number of files. However, the warnings themselves are not necessarily decreasing in those apps as exemplified by the positive raw frequency growth. Tables 4, 5, and 6 show the functions and frequencies for the global data. A complete table with the functions and frequencies can be found within our online appendix [52]. In the following, we describe the most prevalent models.

Top-1 Functions: The top global function for all groups as well as normalization was the *fourier1* function, which fits to 1,552 projects with the average $R^2 = 0.89 \pm 0.11$ for the raw data, $R^2 = 0.78 \pm 0.16$ for the data normalized by LOC, and $R^2 = 0.81 \pm 0.14$ for the data normalized by the number of files.

Table 4: Top 5 best fitting functions for the global set of micro-optimizations opportunities using the raw data and the number projects fit by the function

	Top Function Global	2nd Top Function Global	3rd Top Function Global	4th Top Function Global	5th Top Function Global
Group-All	Fourier (1552)	Log-Pos (337)	Sin (289)	Cubic-Pos (265)	Linear-Const (235)
Group-Low	Fourier (435)	Linear-Const (146)	Log-Pos (45)	Cubic-Pos (38)	Sin (37)
Group-Med	Fourier (767)	Log-Pos (208)	Sin (175)	Cubic-Pos (169)	Power-Pos (113)
Group-High	Fourier (350)	Power-Pos (98)	Log-Pos (84)	Sin (77)	Exponential-Pos (63)

Table 5: Top 5 best fitting functions for the global set of micro-optimizations opportunities normalized by LOC and the number projects fit by the function

	Top Function Global	2nd Top Function Global	3rd Top Function Global	4th Top Function Global	5th Top Function Global
Group-All	Fourier (1592)	Sin (305)	Log-Pos (210)	Power-Neg (205)	Log-Neg (190)
Group-Low	Fourier (402)	Linear-Const (75)	Sin (57)	Power-Neg (50)	Cubic-Neg (32)
Group-Med	Fourier (805)	Sin (158)	Log-Pos (135)	Log-Neg (108)	Power-Neg (96)
Group-High	Fourier (385)	Sin (90)	Power-Neg (59)	Log-Neg (51)	Log-Pos (45)

Because, the fourier model can represent different behaviors, we sampled 905 projects out of the 1,552 projects and manually investigated the distributions of the raw warning frequencies to further understand the behavior of the micro-optimization opportunities in those projects. From the inspection, it appeared that in the projects in group-low, micro-optimization opportunities were introduced as the systems evolve, but the distributions are much less distinguishable (i.e., less clear visually) toward the low-end (i.e., five commits) due to the limited data points. In terms of projects with medium to large histories, a general pattern of increasing warnings is quite evident. While there are projects that have a stronger oscillatory behavior or declining trend, the vast majority shows an increase in the raw number of warnings with a chaotic behavior along this trend or minor fluctuations after the initial peak. Based on this observation, it would suggest that (i) developers only fix certain micro-optimization opportunities, or (ii) the developers do not actively use tools to find micro-optimization opportunities nor take advantage of micro-optimization opportunities.

Top-2 Functions: For the next level of prevalent functions, *linear-constant* fits 142 projects in group-low with the average $R^2 = 1$. Overall, log_+ best fits the raw data (337 projects) with the average $R^2 = 0.91 \pm 0.08$, while the *sine* function best fits the normalized frequencies (305 and 309 projects normalized

Table 6: Top 5 best fitting functions for the global set of micro-optimizations normalized by the number of files and the number projects fit by the function

	Top Function Global	2nd Top Function Global	3rd Top Function Global	4th Top Function Global	5th Top Function Global
Group-All	Fourier (1687)	Sin (309)	Log-Pos (283)	Cubic-Pos (205)	Linear-Const (175)
Group-Low)	Fourier (440)	Linear-Const (117)	Sin (50)	Log-Pos (45)	Cubic-Pos (33)
Group-Med	Fourier (856)	Log-Pos (174)	Sin (172)	Cubic-Pos (121)	Power-Pos (85)
Group-High	Fourier (391)	Sin (87)	Log-Pos (64)	Power-Pos (56)	Cubic-Pos (51)

by LOC and File, respectively). In terms of raw data, Log_+ was also the best fit for group-med projects (208) with the average $R^2 = 0.90 \pm 0.07$, while $power_+$ was the best fit for group-high projects (98) with the average $R^2 = 0.94 \pm 0.11$. The group-low projects were fit best by *linear-constant* with 75 projects.

Top-3 Functions: At the third most prevalent fitting, we see a similar result in which the *sine* function best fits the raw data for group-all (289 projects) with the average $R^2 = 0.88 \pm 0.1$, while the log_+ function best fits the normalized data for group-all (210 and 283 projects normalized by LOC and number of of files, respectively). This observation suggests that while the raw number of warnings fluctuates as represented by the *sine* function, the rate of growth of warnings overall is relatively small compared to the growth of size of the apps. Additionally, we also see that the *sine* function was the best fit for group-med (158) in terms of raw data. Log_+ best fits group-low (45 projects) and group-high (84 projects) for the raw data. Our online appendix contains a complete table of the top 5 best fitting for both the micro-optimization opportunities categories and global data with the number of projects represented by each function.

Summary for $\mathbf{RQ_2}$. We observed a dominance of a first order *fourier* model fitting the data when considering both global data and data for each category of micro-optimization opportunities. The global distributions only showed *linear-const* as the dominant group-all fitting at the 5th ranking (i.e., top-5) of functions (235 projects), but it was prevalent for certain history sizes and particular warning categories. Of these projects, only 12 projects never had a warning of any type introduced during their revision history. However, we observed a high prevalence of *linear-constant* as the top two dominant func-

tion, when we investigated the warnings categories. Additionally, we observe a decreasing function, in the top three fittings for our global pattern, but when normalized by LOC and only for projects with long histories. *The results indicate that the most prevalent behaviors in terms of performance-related warnings detected in the analyzed apps are (i) the micro-optimization opportunities increase during the app history, and (ii) the micro-optimization opportunities are constant.*

4.4. Threats to Validity

Threats to internal validity relate to result bias from confounding factors. In order to reduce the bias of a project's change history on the number of micro-optimization opportunities, we broke the dataset into three tiers (low, med, and high) by considering the distribution of the number of commits. Additionally, we presented the raw frequencies as well as normalized frequencies (by both number of files and LOC) to control for project size that may impact the results as well. Another internal threat is the way in which we measured the micro-optimization opportunities based on the number of warning reported by the tools and the existence of tangled changes; it is possible that a commit in the analyzed systems implemented micro-optimization opportunities, but at the same time introduced new code with several opportunities for micro-optimizations; thus, the number of warnings when reported on that case of commit does not show the whole story. However, our study is evolutionary and considers the whole history of the analyzed systems, and our conclusions are based on the general behaviors we identified with the "function-fitting" analysis.

Threats to construct validity relate mainly to imprecision that would cause a bias between the theory and our observations. It would relate to the underlying tools to identify the micro-optimization opportunities. To reduce this threat, we relied on *LINT*, which is distributed with the Android SDK by Google, and *PMD*, which has been actively developed. We did not implement our own rules for detecting micro-optimizations opportunities, but relied on the existing tools for detecting such warnings.

Threats to external validity expressed the limitation of the results with respect to generalizing them. Although the set of analyzed apps and commits is large, the apps are not commercial apps, and we are mining projects from GitHub. According to [59], from a sample of 434 repositories, 64.4% of the repositories were for software development; in other words, there are also academic and experimental projects. Although our sample is large (3.5K+ repositories), our results cannot be generalized to commercial apps. However, our goal is not to bias the results towards only the practices of experienced developers, but to understand Android developers in general, which include programmers of experimental and academic projects. Moreover, we filtered some projects (as described in section 4) to remove inactive ones and projects with few commits.

5. The Second Study: Measuring the Impact of Micro-Optimizations

Our mining-based study suggests that implementing micro-optimizations is not a practice widely used by open-source Android developers. Consequently, as a first step to understand the reasons quantitatively, we designed a second study with the *goal* of identifying the real impact — of implementing micro-optimizations — on the resource consumption of Android apps. The *quality focus* of the study was the difference in CPU and memory consumption before and after applying micro-optimizations to Android apps. The *context* consists of eight open source apps with a large number of micro-optimization opportunities. The *perspective* is of researchers interested in identifying the benefits of micro-optimizations in Android apps. Therefore, for this study, we were interested in the following research question:

RQ$_3$ *How do micro-optimizations impact CPU and memory usage of Android apps?*

5.1. Analysis Method

To measure the impact of micro-optimizations, we focused on apps with a large number of micro-optimizations opportunities. Instead of measuring the

impact of incrementally applying one, two, three, etc. micro-optimizations, we decided to measure the impact in a scenario when all the micro-optimizations opportunities are addressed. To this end, we developed a tool to automatically refactor (i.e., apply micro-optimizations) Android projects (i.e., source code), based on the list of warnings reported by *PMD* and *LINT*. We utilized a combination of AST-based analysis and textual transformations depending on the type of micro-optimization opportunity and required fix. The implementation of the micro-optimizations is based on the practices and solutions suggested by *PMD* and *LINT*. The tool generates a refactored version of an app (source code and APK) implementing either all the micro-optimizations belonging to a target category, or all the micro-optimizations in all the categories. Our tool, after modifying the source code, automatically builds Android APKs.

Our decision to implement the tool is based on the fact that some of the apps have a large number of performance-related warnings reported by *PMD* and *LINT*, and implementing the corresponding micro-optimizations manually would be both error-prone and require a substantial amount of time. The list of micro-optimizations selected for the second study are those that do not provide false positives and those that can be applied automatically without affecting the external/expected behavior of the app. For instance, the *PMD* warning UseArrayListInsteadOfVector suggests replacing usages of the `java.util.Vector` API with of `java.util.Array List` to improve the performance; however, the change could affect the behavior of the app, because of the synchronized nature of the `java.util.Vector` API (`java.util.ArrayList` is not synchronized). Another example is the ReplaceHashtableWithMap micro-optimization; we discarded ReplaceHashtableWithMap, because there is no direct mapping between the methods in `java.util.Hashtable` and `java.util.HashMap`; therefore, there is no way to automatically implement the micro-optimization.

Regarding false positives, we found that all the warnings in the *UnusedCode* category from PMD generate false positives. Therefore, for the second study, we only considered a total of 15 micro-optimizations marked with a star "*" in

Table 1[3]. Also, for $\mathbf{RQ}_3$, we decided to merge the *Optimization* and *Migration* warnings from *PMD* in the group *Opt./Mig.*, because after filtering the types of warnings (i.e., removing the ones with false positives and the ones not able to be implemented automatically), we ended up having only one type in the *Migration* group.

In addition to our tool, we used the optimizations provided by *Proguard* [60], which is the state-of-the practice tool for optimizing APKs. We used the specific configuration of *Proguard* for optimizing apps as suggested by the related work [61]. In particular, this configuration performs (i) peephole optimizations for arithmetic instructions and casting operations, (ii) removes write-only fields, (iii) marks fields as private, if possible, (iv) propagates the values of fields across methods, and (v) merges classes vertically and horizontally in the class hierarchy, whenever possible. It is worth noting that the micro-optimizations implemented by our tool and the ones in *Proguard* are not the same; thus, each tool provides a different set of optimizations. Also, note that the micro-optimizations performed by our tool are done in the source code, meanwhile *Proguard* optimizations work directly at the bytecode level.

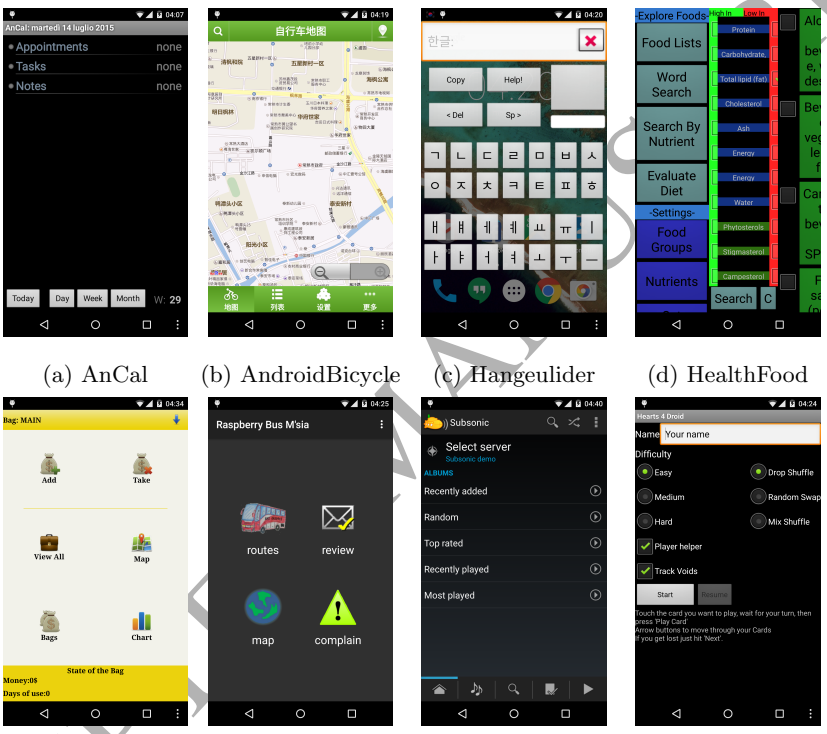
5.1.1. Android Apps Analyzed in the Second Study

Table 7 shows general information on the Android apps selected for the second study. In particular, the table shows (i) the size of the apps in terms of thousands of line of code; (ii) the category in which the apps have been published in the Google Play Store[4]; (iii) the number of installations; (iv) the average rating obtained by the users[5]; (v) the number of ratings; (vi) the date of the last apps' update on the Google Play Store. To select the target projects used to

[3]There is a previous work by Sahin *et al.* [46] on measuring the impact of refactorings on energy consumption of Java apps. In particular, they analyzed the impact of six refactorings on nine Java systems. Therefore, the number of apps and micro-optimizations used in our study is in the same order of magnitude of the study by Sahin *et al.* [46].

[4]`https://play.google.com/store/apps`

[5]Data collected on July 2015



(a) AnCal (b) AndroidBicycle (c) Hangeulider (d) HealthFood

(e) SaveApp (f) RaspberryBus (g) SubsonicAnd. (h) HeartsSky

Figure 3: Screenshots of the Android apps used in the second study

Table 7: The list of Android apps used in the second study. The apps and measurements were collected in 2015. By that time HeartsSky was the only app not available at GooglePlay

App		Google Play				
Name	KLOC	Category	Installs	Rating	# Ratings	Last Update
AnCal	14	Productivity	10,000 - 50,000	3.3	83	1/20/2010
AndroidBicycle	5	Travel & Local	1,000 - 5,000	3.5	19	6/18/2012
Hangeulider	1	Communication	10,000 - 50,000	4.1	110	5/4/2009
HealthFoodConcepts	20	Health & Fitness	50 - 100	-	-	6/18/2013
SaveApp	5	Finance	1,000 - 5,000	4.7	12	4/18/2012
RaspberryBusMalaysia	4	Travel & Local	1,000 - 5,000	3.8	9	7/18/2013
Subsonic-Android	14	Music & Audio	100,000 - 500,000	4.3	6,203	8/31/2014
HeartsSky	4	-	-	-	-	-

answer $\mathbf{RQ}_3$, one of the authors inspected the list of the Android projects from GitHub with the most number of micro-optimizations opportunities (reported by *PMD* and *LINT*) grouped by category (as detected in the first study). Given three lists of top apps (one for each micro-optimization category) ranked in descending order of micro-optimization opportunities, during the inspection, the author reviewed the lists (i.e., *String*, *Opt./Mig.*, *LINT*) from top to bottom looking for projects that could be actually built to generate an APK and are able to run on a device (i.e., the author manually compiled/built and ran the apps from the lists). The number of micro-optimization opportunities reported by *PMD* and *LINT* in the selected apps are listed in Table 8. The analyzed apps are described as in the following (screenshots are depicted in Figure 3):

(a) Ancal: Calendar, Tasks, and Notes organizer[6];

(b) AndroidBicycle: Provides information about bike distribution points for several Chinese cities[7];

(c) Hangeulider: Virtual Keyboard for writing Korean using a QWERTY keyboard[8];

[6] `https://play.google.com/store/apps/details?id=pl.magot.vetch.ancal`
[7] `https://play.google.com/store/apps/details?id=com.dreamcatcher.bicycle`
[8] `https://play.google.com/store/apps/details?id=com.choibean.android.`

Table 8: Number of warnings detected in the Android apps used in the second study

App	Micro-optimization opportunities			
Name	Opt./Mig.	String	LINT	Total
AnCal	334	22	22	378
AndroidBicycle	136	2	6	144
Hangeulider	148	1	147	296
HealthFoodConcepts	26	343	9	378
SaveApp	10	64	27	101
RaspberryBusMalaysia	6	59	6	71
Subsonic-Android	6	20	100	126
HeartsSky	60	1	169	230

(d) HealthFoodConcepts: Provides information about foods' nutritional components, and help to construct a personal diet[9];

(e) SaveApp: Finance app that helps to keep track of the users' economic movements[10];

(f) RaspberryBusMalaysia: Provides information about routes of buses as well as the possibility to review and rates transport services[11];

(g) Subsonic-Android: Music and Video streaming app[12];

(h) HeartsSky: Hearts card game[13];

hangeulider

[9] https://play.google.com/store/apps/details?id=co.harlequinmettle.healthfoodconcepts

[10] https://play.google.com/store/apps/details?id=com.loopback.androidapps.saveapp

[11] https://play.google.com/store/apps/details?id=com.sweetiepiggy.raspberrybusmalaysia

[12] https://play.google.com/store/apps/details?id=net.sourceforge.subsonic.androidapp

[13] https://github.com/Shadoath/heartsSkyThread

The diversity of the apps is summarized as in the following according to the statistics reported by Google Play:

- *Size*: the size of the apps varies between just 1 KLOC to 20 KLOC;
- *Category*: the apps belong to 6 different Google Play Categories;
- *Installs*: 4 different intervals of installations from 50 to 500,000.
- *Rating*: the average ratings vary between 3.3 and 4.7, while the number of ratings goes from only 9 to 6,203.

It is worth noting that for the app HeartsSky, there is no Google Play information, since this app has not been published in the market, but the app source code is available at GitHub.

5.1.2. Data Collection

For each app in the second study, we measured the CPU and memory usage of four versions of each app: (i) original app, (ii) refactored app with all the micro-optimizations applied, (iii) refactored app with only micro-optimizations of a target category, and (iv) app with *Proguard* optimizations. This procedure allowed us to analyze whether the improvement (if any) in CPU and memory usage was because of the micro-optimizations in the target category. After refactoring the source code and building the APKs, we inspected the generated APKs to ensure that they compiled and the micro-optimizations did not introduce syntactic bugs. Additionally, we tested the functionality of each generated APK by executing them manually and with scripts to ensure the changes (introduced by our refactoring tool) did not crash the application or caused inconsistent behavior.

In order to measure the CPU and memory usage of the APKs automatically, we used a record-and-replay strategy similarly to previous work on automated testing of Android apps [62, 63, 64, 65, 66, 67]. For instance, for each app A, we recorded an execution scenario E_A, which was replayed on the four versions of app A (e.g., $A_{Original}$, A_{String}, A_{All}, $A_{Proguard}$). To collect the replayable

Table 9: Coverage obtained for each app and a number of warning-related statements not covered by the collected traces

App	Coverage				Micro-Optimization Opportunities
Name	Branch	Statement	Method	Total	Non-covered
AnCal	73.6%	82.6%	85%	83.2%	2
AndroidBicycle	35.3%	57.9%	59.1%	54.5%	5
Hangeulider	69%	89.8%	90%	86.3%	0
HealthFoodConcepts	74.4%	77.1%	78.8%	76.7%	0
SaveApp	54.4%	70.3%	63.3%	67.8%	2
RaspberryBusMalaysia	63%	86.7%	89.2%	82.1%	0
Subsonic-Android	58.9%	71.8%	74.8%	69.6%	3
HeartsSky	51.1%	63.4%	55.7%	60.7%	8

scenarios, we manually generated a meaningful sequence of GUI events (i.e., executions scenarios) executing the original APK directly on a device (Nexus 4 with Android 4.4.4) and recorded the kernel GUI events using the `adb shell getevent -t` command. The recorded scenarios were created aiming to cover as much functionality as possible in the APK under analysis. To guarantee high coverage, we measured the statement coverage obtained by each scenario. To do so, we used the Clover [14] tool to collect a statement coverage report (Clover was also used by Sahin *et al* [46] in their study of impact of refactorings on energy consumption). The coverage achieved for each app with the collected scenarios is showed in Table 9, while a complete report can be found within our online appendix [52].

Table 9 also shows the number of statements reported by *PMD/LINT* as part of a micro-optimization opportunity but not covered by the replayable scenario. We computed these results by performing an automatic cross analysis between the statement-level coverage and the micro-optimization location. For each app, we miss less than 3.5% of statements involved in micro-optimization opportunities (0% for 3 apps) and globally we miss less than 1.2%. We manually inspected the source code of the apps in order to understand which code

[14] `https://www.atlassian.com/software/clover/overview`

was not covered by the execution scenarios. For instance, in the *AnCal* app, the recorded scenario does not cover five classes. Four of the classes are not involved in any micro-optimization opportunity; but, the remaining uncovered class, `AlarmSound`, has two micro-optimization opportunities (two RedundantFieldInitializer) in statements that are unreachable (i.e., dead code).

For the *AndroidBicycle* app, we were not able to cover several classes because when the execution reached those classes, exceptions were thrown[15]. In our recorded scenario, we avoided those classes in order to have an exception-free execution. Regarding the *HealthFoodConcepts* app, we found that the non-covered statements were because of (i) 13 blocks of dead code (`if(false)` statements), and (ii) several getter/setter methods, which are never referenced (i.e., dead methods). Nevertheless, it is worth noting that the code not covered by the recorded scenario does not contain any micro-optimization opportunities. In the case of *SaveApp*, for six classes, we could not execute any statement; three of those classes are disabled functionality (e.g., take a photo, edit a note) and not even specified in the `AndroidManifest.xml`. *SubsonicAndroid* is a large Android music application consisting of 14KLOC, thus, reaching a high coverage with the collected trace was a non-trivial task. We checked the classes that were not executed with our recorded scenario, and we found that only three out of 152 micro-optimization opportunities involved those classes. Finally, for the *HeartSky* app, we found three classes for which we could not execute any statement, however, none of them had a reference in the source code. We achieved only 18.2% of coverage on the class `Game` of *HeartSky*. The code never executed in the `Game` class is responsible for managing the end of the game. Obviously, in a random card game like *Hearts*, it is a hard task to guarantee the execution of that part of code (with a single execution scenario).

The collected scenarios were used to automatically reproduce the exact sequence of GUI events and inputs on all the APKs of each app. It is worth

[15]Note that this happened because of bugs in the original APK, and are not related to the refactorings performed by our tool

noting that each scenario was executed 30 times in order to minimize the effect of possible abnormal executions or race conditions. Before each scenario execution the device was rebooted. Then, we waited additional 60 seconds after the system started-up, and we uninstalled/installed the APK to avoid race conditions related to app pre-fetching saved/cached data from previous executions. If the app did not require Wi-Fi, we set the device in *airplane mode*; otherwise, we tested all the different versions of the same app positioning the device at the same distance from the router with good Wi-Fi signal.

The following high-level steps were automatically executed for each single execution using an automatic script:

1. The APK is installed in the device using `adb shell pm install <apk>`.
2. The script waits for 30 seconds before starting a new execution without interference from previous executions and the installation phase.
3. The APK is launched using `adb shell am start -n <app>`.
4. The scenario is replayed.
5. When the replaying finishes, the APK is force-closed by the script.
6. Memory and CPU statistics are collected from the device.
7. The script cleans all the APK data (cache and other files) using `adb shell pm clear <app>` and specific commands for those apps which create files in the sdcard.
8. Finally, the APK is uninstalled using `adb shell pm uninstall <app>` command.

5.1.3. Memory Statistics

The memory data is collected automatically using the Procstats Android utility released with Android KitKat [68]. Procstats can be invoked remotely via ADB using `adb shell dumpsys procstats`. It tracks the overall accumulated memory consumption for the entire app execution during individual executions

of apps and services. In particular, the values reported by Procstats are *minPss, avgPss, maxPss, minUss, avgUss, maxUss*, where:

Uss: (Unique Set Size) is the set of pages that are unique to a process. This measurement is the amount of memory that would be freed at the moment that the app is terminated [69].

Pss: (Proportional Set Size) is the amount of memory shared with other processes, accounted in a way that the amount is divided evenly between the processes that share it. This measurement is memory that would not be released if the process was terminated, but it is indicative of the amount that this process is "contributing" [69].

5.1.4. CPU Statistics

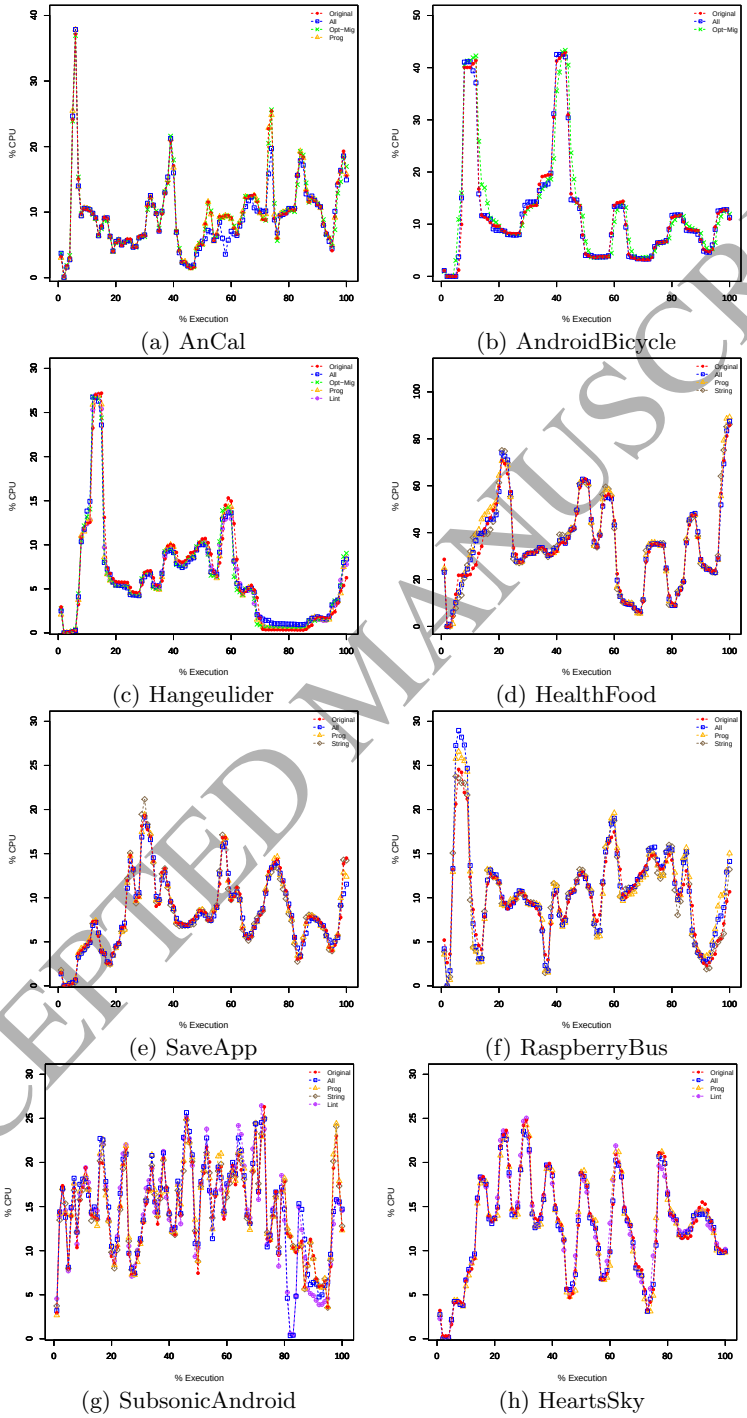
The instant CPU usage of an Android app can be queried with `adb shell dumpsys cpuinfo`. Conversely to Procstats, the `dumpsys` utility does not provide aggregate statistics. Therefore, we adopted a sampling-based approach to generate the CPU statistics of a single APK execution. Before the app is launched, our execution script creates a new thread (on the machine-side, not on the device) that continuously collects CPU samples for the desired app by calling the `dumpsys` command repetitively. The aforementioned command returns: (i) user, (ii) kernel, (iii) and total CPU usage percentage. Because the thread is started by code outside of the APK, it does not impact the CPU measurements for an APK under analysis.

The sampling process generates a series representing the instantaneous CPU usage for each execution of an APK. We analyzed the series using descriptive statistics and checked if there were statistically significant ($\alpha = 0.05$) differences between the series for different APKs (i.e., original and optimized versions) of the same app. To validate whether the results are statistically significant, we used the Kruskal-Wallis test [70] with post-hoc test procedure for pairwise comparisons when required (Mann-Whitney test with Bonferroni correction [70]). We also estimated the magnitude of the difference with the Cliff's Delta (or

d) coefficient, a non-parametric effect size measure [71] for ordinal data. We followed the guidelines in [71] to interpret the effect size values: negligible for for $d < 0.147$ small for $d < 0.33$ (positive as well as negative values), medium for $0.33 < d < 0.474$ and large for $d \geq 0.474$. We are neither assuming population normality nor homogeneous variances; therefore, we choose non-parametric methods (Kruskal-Wallis test, Mann-Whitney test, and Cliff's delta). The results for $\mathbf{RQ}_3$ are presented in Section 5.2.

5.2. Results for RQ_3: Measuring the Impact of Micro-Optimizations on Performance and Resource Usage

Figure 4 shows the CPU usage (%) of each APK version (**Original**, optimized with target violations category: **String**, **Lint**, **Opt-Mig**; **All** the violations fixed; and **Prog**uard version) when replaying the recorded scenarios. In order to obtain these plots, for each APK version, we divided the CPU sample (i.e., the data series collected for each APK) into 100 bins based on their order (the i-th bin contains the sample collected at the i% of completion of the execution scenario) and plotted for each bin the corresponding average of the sample (i.e., average over the 30 executions). The results depicted in Figure 4 suggest that, overall, the optimized APKs have similar CPU usage as the original APK (red full dots) during the whole execution of the apps. However, small differences can be noted in *AnCal* and *SubsonicAndroid*. Regarding the former, the fully optimized version (blue square) have slightly lower CPU consumption between 50-70% of the execution trace. The CPU saving is more evident in *SubsonicAndroid* around 80% of the execution trace, where there is a significant drop of the CPU usage from 10% of the original version to almost 0% of the fully optimized and *LINT* version (purple circle). However, *All* and *LINT* versions have a subsequent peak higher than the original version by 3-4%. Overall, these graphs indicate that even though micro-optimizations can provide in few cases some benefits in terms of CPU consumption, these are rather small and isolated cases. The results are confirmed by the statistical test results listed in Table 10. There are some APK versions with CPU improvements that are statistically



(a) AnCal (b) AndroidBicycle

(c) Hangeulider (d) HealthFood

(e) SaveApp (f) RaspberryBus

(g) SubsonicAndroid (h) HeartsSky

Figure 4: CPU consumption comparison of different APKs versions across the execution trace

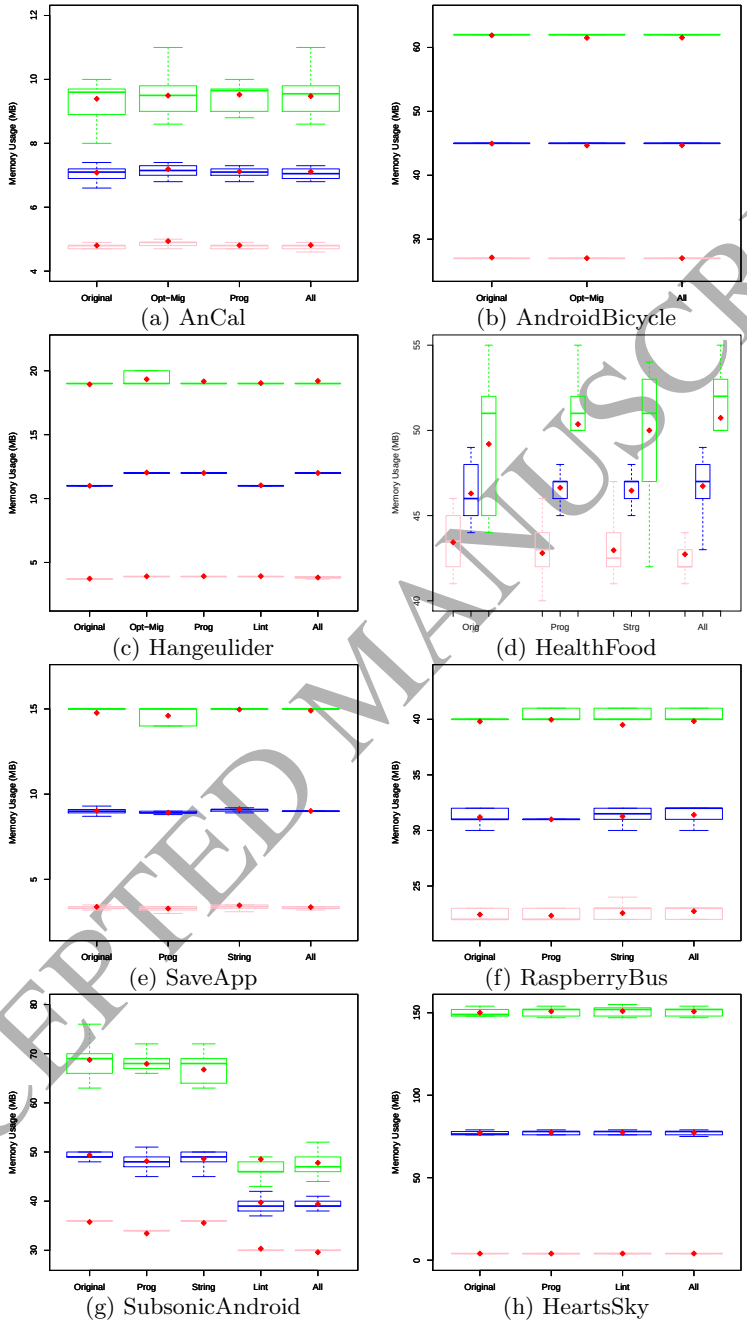


Figure 5: Memory consumption comparison of different APKs versions. Three boxplots are displayed vertically for each APK version: $minUss$ (pink), $avgUss$ (blue), and $maxUss$ (green). *HealthFood* boxplots are presented horizontally, because of distributions overlap.

significant as compared to the original version, however, the magnitude of the difference is negligible (Cliff's delta) in all the cases.

Figure 5 depicts the box-plots for the memory consumption of the different APKs' versions. For each version, we plot the distribution of values for *minUss*, *avgUss* and *maxUss*. The results for *Pss* are available within our online appendix [52]. It is worth noting that for *Hangeulider*, we have four optimized versions, since this app was selected as representative for both the *LINT* and *Opt./Mig.* categories. For *AndroidBicycle*, we do not have the *Proguard* version, since the optimized APK crashed on every execution. Similar to the CPU usage, there is no general improvement in the memory consumption of the optimized versions, except for the case of *Subsonic Android*. In fact, for several APKs, we obtained even worse results as compared to the original version. However, concerning the statistical tests for *Subsonic-Android*, the differences are large for all the memory metrics, when considering the *LINT* and *All* versions; the differences are also large for the *Proguard* version in all the memory metrics, except for the *maxUss* and *maxPss*. Table 11 shows the statistical test results comparing the *avgUss* of the optimized versions against the original version. Note that negative Cliff's Delta refers to cases where the optimized version consumes more memory than the original one. All the results are available on our online appendix [52].

Finally, we performed a deeper analysis of the `Subsonic-Android` app that shows significant improvement in the memory consumption across all the optimized versions. In addition to the *LINT* version, we analyzed the String and *Opt./Mig.* APK versions to understand whether the improvement was only due to the *LINT*-related micro-optimizations. In fact, the improvement was a consequence of the *LINT* optimizations. Subsequently, in order to shed light on the sources of the improvement, we manually inspected the micro-optimizations.

We found the improvement to be mainly due to the removal of several unused resources (images, strings and xml files) in the optimized version. In particular, the memory saving comes mainly from the deletion of seven unused hdpi images (in the `drawable-hdpi` directory), four full xml files, and more than 100 unused

strings (mainly in `res/values/strings.xml` and `res/values/colors.xml`). In Android apps, the resources (i.e., images, strings, styles) declared statically by means of files in the "res" folder are usually referenced in the source code by means of ids declared in the `R` class and its subclasses [58]. The `R` class is loaded into memory when the application starts, therefore the lower the number of resources, the lower the number or resource IDs in the `R` class and the memory consumption.

Nothing can prevent Android to cache or pre-load resources into the memory [72]. This seems to be the case especially for drawable resources. In fact, we found, on the popular Q&A website StackOverflow, several developers experiencing delays or memory bloats due to unused resources. For example, an Android developer asked: *"I've added about 35 images to my res/drawable folder and another 30 sound files to res/raw folder. Now my application takes a good 5 seconds longer to start even tough only 1 image and 1 sound resource is used for the startup screen. That makes me ask: When are my image resources loaded into memory?"* [73]. Our experimental results confirm that unused resources, for which we manually checked that there were no reference in the code, can be pre-loaded by Android contributing to the overall app memory usage.

Finally, regarding the actual amount of memory usage, it's worth noting that even relatively small drawables (in terms of file size) can significantly affect the memory usage when loaded. Indeed, images are compressed when stored on disk, while they are uncompressed when loaded into the memory.

Summary for RQ$_3$. The results demonstrate that the analyzed micro-optimizations do not significantly improve CPU or memory consumption (except for the UnusedResources case), even in those apps that contain a high number of warnings. However, we found evidence that developers can significantly reduce the memory consumption of the running app, by removing unused resources, which can be automatically detected by the Android *LINT* tool. Our empirical results suggest that while developers should not generally expect significant improvements in CPU or memory consumption when applying

micro-optimizations, there is no harm in performing such micro-optimization - especially in a fully automated setting - which in some cases could lead to performance improvements (e.g., in the case of SubsonicAndroid).

5.3. Comparison of the Results

In this section, we compare and discuss our results with respect to other Android-based performance studies as well as traditional Java/C applications.

In terms of *Android OS* optimizations, Kim *et al*[29] showed that packing sequential writes can significantly improve the storage performances on android smartphones. Jeong *et al*[28] reported a performance increase of 300% in SQLite when optimizing the I/O stack on Android. *Wake Locks* have also been shown to be an important target for optimizations. Indeed, an incorrect use of wake locks can lead to an additional battery drain that can ranges between 5% and 25% [32]. Zhang *et al*[36] described a set of common causes for *Energy Leaks* in Android apps, showing that eliminating such leaks results in an average of 56.5% reduction in energy consumption. *API Usage Patterns* [35] and *Android Code Smells* [37] have also been shown to have a non-negligible impact on the app energy consumption and performance. Our empirical results show that, in the context of Android optimizations, micro-optimizations provide in most of the cases negligible performance improvements with respect to the aforementioned Android optimizations studied in previous works.

Regarding studies on Java/C desktop applications and libraries, Hindle [44] showed that power consumption can vary across software versions. Moreover, the choice of the right Collection [74] and appropriate refactoring operations [46] can significantly impact the energy used by a software application. While desktop and mobile applications have very different non-functional requirements in terms of power consumption, micro-optimizations appear to have a smaller impact with respect to what observed for refactoring operations on desktop Java

Table 10: Statistical tests comparing optimized versions against the original for CPU consumption

App	Version	p-value	Cliff's Delta	Magnitude
AnCal	Opt-Mig	0.27	< 0.01	negligible
	Prog	0.01	< 0.01	negligible
	All	< 0.01	0.05	negligible
AndroidBicycle	Opt-Mig	< 0.01	< 0.01	negligible
	All	0.39	< 0.01	negligible
Hangeulider	Lint	< 0.01	0.03	negligible
	Opt-Mig	< 0.01	0.03	negligible
	Prog	< 0.01	0.02	negligible
	All	< 0.01	0.04	negligible
HealthFoodConcepts	String	0.38	< 0.01	negligible
	Prog	0.03	< 0.01	negligible
	All	< 0.01	< 0.01	negligible
SaveApp	String	< 0.01	0.01	negligible
	Prog	0.94	< 0.01	negligible
	All	< 0.01	< 0.01	negligible
RaspberryBusMalaysia	String	0.58	0.01	negligible
	Prog	< 0.01	0.02	negligible
	All	< 0.01	0.02	negligible
Subsonic-Android	Lint	< 0.01	0.02	negligible
	Prog	< 0.01	0.03	negligible
	All	< 0.01	0.06	negligible
HeartsSky	Lint	0.92	< 0.01	negligible
	Prog	< 0.01	0.01	negligible
	All	< 0.01	0.01	negligible

Table 11: Statistical tests comparing optimized versions against the original for Memory consumption (avgUss)

App	Version	p-value	Cliff's Delta	Magnitude
AnCal	Opt-Mig	0.14	-0.22	small
	Prog	0.47	-0.11	negligible
	All	0.90	< 0.01	negligible
AndroidBicycle	Opt-Mig	0.10	0.13	negligible
	All	0.60	0.16	small
Hangeulider	Lint	0.58	0.03	negligible
	Opt-Mig	< 0.01	-0.97	large
	Prog	< 0.01	-0.97	large
	All	< 0.01	-0.97	large
HealthFoodConcepts	String	0.55	-0.08	negligible
	Prog	0.41	-0.12	negligible
	All	0.22	-0.18	small
SaveApp	String	0.06	-0.27	small
	Prog	0.01	0.36	medium
	All	0.97	< 0.01	negligible
RaspberryBusMalaysia	String	0.38	-0.12	negligible
	Prog	0.23	0.15	small
	All	0.13	-0.20	small
Subsonic-Android	Lint	< 0.01	0.99	large
	Prog	< 0.01	0.51	large
	All	< 0.01	0.99	large
HeartsSky	Lint	0.15	-0.20	small
	Prog	0.17	-0.19	small
	All	0.32	-0.14	negligible

software.

5.4. Threats to Validity

Threats to *internal validity* are related to factors, internal to our study, that can influence our results. In our context, these threats arise primarily from the technique that we used to measure CPU and memory consumption. Regarding the CPU measurements, although the values were collected with Android-specific tools running on the device, which increase the precision, the CPU sampling process does not ensure that we measure the CPU usage in the same location for each version of the apps. In the case of memory usage, we collected the average memory used during a whole execution. Thus, to reduce the impact of threats to internal validity (i) we executed the same scenario automatically 30 times for each APK of an app, and (ii) we used appropriate statistical tests to measure statistical significance.

Threats to *external validity* concern the generalizability of the results. In our study, since we analyzed a reduced set of optimizations (15) that can be refactored automatically, the results are not generalized to all the available micro-optimizations. Therefore, it is possible that some of the micro-optimizations that are not analyzed in this study could improve CPU and memory usage. However, compared to previous work on analyzing the impact of refactorings on energy consumption [46], our study analyzes almost two times as many refactorings than had been previously analyzed. Another concern in terms of generalizability of the results arises from the Android version and the device that we used in our experiments. Regarding the Android version, we used Android 4.4, since it was the most popular Android version at the time (according to statistics as of Sept. 2016 [75]); however, the results might not be generalizable to different Android versions, which could rely on different compiler and OS internal optimizations. For example, a different resource pre-loading policy (e.g., for drawables) could impact the effectiveness of specific micro-optimizations. Regarding the device, we used the Nexus 4 with stock Android. While we believe that results should not significantly vary across different devices, given the fact that most of the Android devices rely on ARM-based processors, this intuition

should be empirically validated. Future research will be conducted to empirically measure the impact of micro-optimizations on different Android versions and devices.

6. The Third Study: Surveying Android Developers' Micro-Optimization Practices

Studies 1 and 2 demonstrate from a mining and resource measurement perspectives that micro-optimizations are not widely used by Android developers, and the impact of the micro-optimizations was not significant in eight apps, except for the cases of removing unused resources (e.g., images, strings). However, results from Studies 1 and 2 are not enough to understand the rationale behind the state of practice in-the-wild. Therefore, in order to understand the current practices of Android developers towards micro-optimizing their apps, we designed an online survey to fill this gap. The *goal* of this third study is to identify whether developers use micro-optmizations in-the-wild, and their rationale. The *context* consists of 389 Android open source developers and their answers to our online survey. The *perspective* is of researchers interested in developer practices for improving performance of Android apps. The survey and analysis of the responses were focused to answer the following RQs:

RQ$_4$: *What practices are used by developers for detecting optimization opportunities in Android apps?*

RQ$_5$: *What micro-optimizations are recognized (and used) by developers as useful for improving the performance of apps?*

6.1. Analysis Method

The survey questions are listed in Table 12. Besides the survey specific questions, we asked also demographic background questions. Concerning the participants, we identified the developers that contributed changes to the 3,513

Table 12: Survey questions for the third study

Question (Type)
1. For how many years have you been programming? (Numeric)
2. For how many years have you been developing Android apps? (Numeric)
3. What is your academic level? (Single-choice: *High school—Bachelor—Master—PhD—PostDoc*)
4. How do you detect opportunities for performance optimizations in mobile Apps? If you use any commercial or open-source tools please mention them (Open)
5. Do you use micro-optimizations for improving the performance and resources consumption of mobile apps? (Boolean)
6. If your answer was "Yes" to question 5, please list the micro-optimizations, including the ones not mentioned in the links, that you have applied to successfully improve the performance and resources consumption of your app as well as the context of the specific changes (Open)
7. If you answer was "No" to question 5, please explain the reasons why not (Open)

apps in our dataset for Study 1, merging email addresses that occurred across multiple projects to avoid duplicates. Subsequently, we filtered invalid email addresses by removing instances of *[user]@localhost* and *[user]@none.**. Developers at Google were also removed from our study to avoid developers that work on Android's core operating system, which is beyond the scope of our study. Then, we emailed the survey to 24,340 emails addresses, from which we got 5K+ messages reporting "undelivered message" and 425 survey responses. For the analysis, we used descriptive statistics and open coding [76, 77] of the free-text questions. One of the authors with experience in Android development

and Android tools revised the responses of the open questions and categorized the results. The results for $\mathbf{RQ}_4$ and $\mathbf{RQ}_5$ are presented in Sections 6.2 and 6.3 correspondingly.

6.2. Results for RQ_4: Practices for Detecting Optimization Opportunities

In order to answer $\mathbf{RQ}_4$, we analyzed survey responses from 425 participants. To filter out invalid answers, we removed responses of participants with less than one year and more than 10 years[16] of experience in mobile development. After the filtering, we obtained responses from **389** participants. On average, the participants had 12 years of experience in programming and three years of experience developing Android apps (Figure 6). Concerning the academic level, 44.22% of the participants only had undergraduate degree, and 40.36% had an M.S. degree (Figure 6). Also, Figure 7 depicts the geographic distribution of the survey participants. The complete details regarding programming experience and academic level of the 389 participants can be found in our online appendix [52].

Out of the 389 responses, 346 had answers to the question regarding practices for detecting optimization opportunities. However, in 43 instances to this answer, we were not able to decipher the practice; and 32 participants responded saying that they are not interested in optimizing code. Thus, discounting those responses, we have 271 informative answers. The practices declared in the 271 answers are codified in the taxonomy depicted in Table 13.

In general, the informative answers revealed the existence of a diverse set of practices and tools for detecting optimization opportunities. However, the responses show a bias in preference towards dynamic-based practices and tools provided by Google. Only 82 participants included in their answers static analysis-based practices, and only 15 participants indicated the usage of static analysis tools (e.g., *PMD*, *FindBugs*, *LINT*) for detecting optimization oppor-

[16]The development of Android started in 2003 and an Alpha version was available by 2005-2006

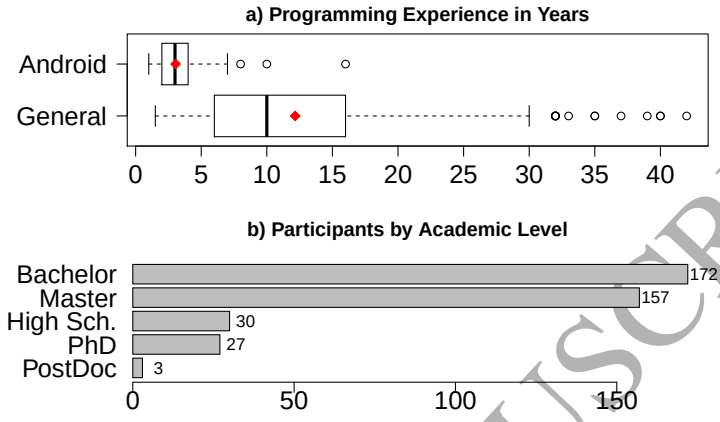


Figure 6: Programming experience of survey participants

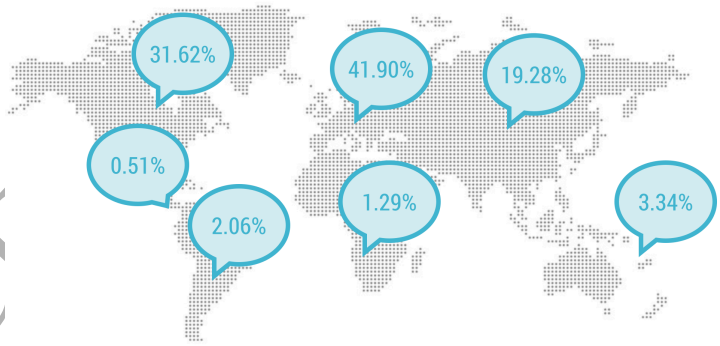


Figure 7: Geographic distribution of the 389 survey participants with valid responses. For the case of America the distribution is presented individually for each sub-continent (i.e., North, Central, and South America)

tunities.

Manual testing and profiling turned out to be the most preferred practices. Only two participants rely on automated unit testing tools. In the case of profiling, the most used suites and tools belong to the Android ecosystem, such as Eclipse MAT[4], DDMS [3], Android Device Monitor [78], traceview [79], systrace [80], and dmtracedump [79]. Only 20 answers included third-party tools such as Valgrind [81], LittleEye [82], AT&T ARO [83], Intel Parallel Studio [84], Adreno profiler [85], among the others.

Summary for RQ$_4$. Although the current practices and tools based on dynamic analysis do not support automatic detection of optimization opportunities, these techniques were preferred by the survey participants over static analysis tools for ad-hoc fixing of performance related issues or early detection of micro-optimization opportunities.

6.3. Results for RQ$_5$: Usage of Micro-optimizations

We mostly received negative answers concerning the usage of micro-optimizations; 246 out of 389 (63.24%) participants stated they do not use micro-optimizations for improving performance or resource consumption in Android apps. 224 participants provided rationale for this negative choice. The reasons are explained mostly by their lack of knowledge of available micro-optimization practices and tools (66 answers), the position that optimizations are not required because their apps were rather simply (62 answers), lack of confidence in premature optimization (50 answers), and lack of interest in optimizing the apps in general (24 answers). Other answers stated reasons as not enough time in the development cycle (10 answers) and bad user experience with the tools (3 answers). Finally, 8 answers were not clear; thus, we could not classify them.

Examples of the answers representing the top reasons for not using micro-optimizations are the following: "*I haven't been aware of micro-optimizations or their impact before*"; "*unaware that they are helpful*"; "*not convinced they are worth the time*"; and "*Your survey just informed me of this. I did not know such tools existed, as I solved performance in my app by using common sense*".

Table 13: Taxonomy of practices for detecting optimization opportunities. Dynamic analysis-based techniques are marked with (D), and static analysis-based ones with (S). The percentage of participants using the technique is also listed.

Category	**Technique**	**Description**
Direct observation (D)	Testing (21.18%)	Manual/automatic execution aimed at detecting hot-spots by using common sense and personal judgment
	Debugging (2.12%)	Manual execution in step-by-step mode with breakpoints and detection of hot-spots by using common sense
	LogCat analysis (2.35%)	Manual/automatic execution and analysis of the information available in the Android LogCat such us garbage collection, dropped frames, etc.
	GUI analysis (1.65%)	Manual/automatic execution and analysis of inefficient draw calls, excessive nesting, unused components.
Data analysis-based (D)	Profiling (24.94%)	Resource-related measurements are collected using a profiler and analyzed in runtime
	Benchmarking (1.41%)	Performance of a target app is evaluated with a set of conditions or compared to a set of apps belonging to a benchmark
	Monitoring (0.94%)	Same than profiling, but the profiling is done online meanwhile the app is used by real users.
Policy-based (D)	Android StrictMode (0.71%)	Android specific instrumentation, in which policies are defined via API and the policies' violations are detected by the framework in runtime
Post-mortem (D)	Reviews/Bug reports (4.94%)	The optimization opportunities are detected by users and reported as bugs or requests for improvements via user reviews or bug reports
	Crash analysis (1.88%)	Crashes are collected online in runtime, then the reports are analyzed to detect performance issues
Code analysis (S)	Code reviews (8.95%)	Source code is inspected manually
	Lint (3.53%)	Source code is inspected by using the Android Lint tool
	Analysis (0.47%)	Analysis of algorithm techniques are used to estimate time and memory
Documents (S)	Crowd-based (2.59%)	Participants look for advice on forums, developer groups, documentation, etc.
Prevention (S)	Best practices (4.71%)	Good programming/design practices, acquired mostly by experience
	Native code (1.18%)	Usage of native code and reduction of the dependency in Android API
	Libraries (0.24%)	Participants rely on open source libraries to reduce risk of optimization issues
	Latest API (0.24%)	Usage of the latest version of the Android API is considered to prevent optimization issues

The following are some longer answers explaining the lack of confidence in micro-optimizations:

"I believe strongly in optimizing only things that meet a two-fold test: It has to be an observed performance problems, and it has to be the hotspot in that problem. (Usually not the hotspot in the program, just the hotspot in the phase of execution that has problems.) Micro-optimization is essentially orthogonal to that."

"I was surprised by the selection of micro-optimizations there. That kind of thing matters on servers, where there's a steady load caused by thousands or millions of users. But on a phone there's usually lots of spare CPU. Not always: When the user moves a finger you have to react very quickly. But most of the time the program has all the time in the world."

"The gains are usually very small compared to previously mentioned strategies (may depend on the code you write of course). So I think it only makes sense as a last effort and so far I have not needed it"

"Typically network requests are the bottle neck and micro-optimizations only provide minor improvements"

"Many of the micro optimizations mentioned on the previous page can be categorized as "be a good engineer who doesn't do write inefficient code". Many of the others are, unfortunately, "this may speed up the program by 0.03% but it also makes the code dramatically harder to read./ My instinct is that you can get 90% of the benefits with 10% of the drawbacks simply by being aware of efficiency pitfalls. That said, I've never actually evaluated the performance gains you might get from running a tool like PMD on an entire source base, so maybe I'm underestimating the cumulative effects."

*"we don't prematurely optimize, favoring code that's easy to read and maintain over efficient code, **until** there's a noticeable issue."*

From the 143 participants using micro-optimizations, we received answers mentioning specific optimizations and answers asserting the usage of *LINT*, but without details on the specific micro-optimizations. We analyzed only the answers detailing the optimizations and grouped them into categories. Figure 8 depicts the categories and the frequency an optimization in that category was mentioned by the participants. Most of the cases refer to micro-optimizations for memory management, operations with strings and collections, and loops.

Regarding memory management, the optimizations reported by participants consist of reusing existing objects, caching for JNI, memoization of cursors and views, using `recycle()` for Bitmaps and other objects, and tuning Garbage Collection. One of the participants wrote about the impact of the `Activity.findViewById` method, which has been described by an Android Framework Engineer as an expensive function [86], and the method was detected as an energy greedy API [35]:

> *"You'd be surprised how much of a difference memoizing the result of findViewById or finding the index of a column in cursor will do. It can dramatically affect list scrolling. These days I use constant column indexes for all cursors even though they require more maintenance because the performance improvement is so great."*

Another participant described that bitmaps recycling have beneficial impact on memory consumption:

> *"This might not be a micro-optimization: Recycling bitmaps and reducing what bitmaps are loaded in the first place, to reduce memory use. Using MAT to understand memory use with bitmaps helped significantly reduce memory footprint and fix crashes on some devices."*

Two participants pointed us to websites that (i) describe the usage of Hprof [87] for performance analysis on Android apps, and how the individual used it to detect/remove unnecessary object allocations, and (ii) lists good practices in Java — with examples— oriented to improve performance [88]. The former website also shows that avoiding getters and setters reduce the memory consumption.

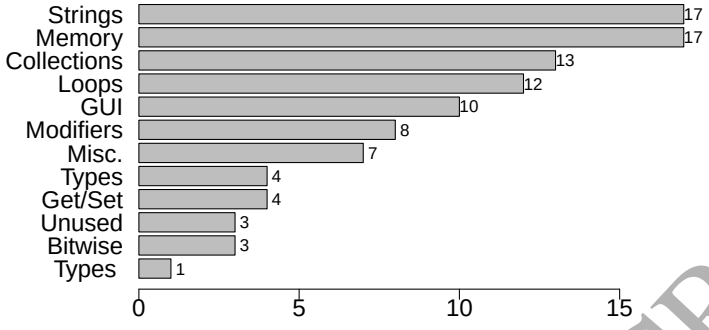


Figure 8: Number of participants with successful applications of micro-optimizations to improve performance

Another example of the importance of reusing objects is described with the following answer:

> *"Sometimes, particularly for view drawing, it's helpful to reuse objects rather than creating new ones to avoid garbage collection."*

The second group of optimizations reported by the participants is related to string operations, such as use `StringBuffer` instead of `String`, avoid the usage of `String.format()`, and avoid instantiating `String` objects. These types of optimizations are considered as "classic" example of Java best practices that do not have a significant impact when the strings or the number of operations are small. Unfortunately, the participants did not provide specific contexts (i.e., type and number of strings) for getting benefit from those optimizations. And in Study 2 we did not find significant memory improvements when applying String micro-optimizations.

In the case of loops, the optimizations consist of avoiding instantiating objects in loops, using the break statement in loops, replacing loops with native code, avoiding overly nested loops, and unrolling loops. Interesting responses were "*Select the right collection for the type of content and looping mode*" and "*avoid foreach*". Java provides four different loop constructs, and the existence

of the `Iterator` class for collections introduces another way for interacting with loops. The question here is whether the choice of looping mode significantly impacts the performance of Android apps. Further work will be devoted to analyzing that question. In the case of collections, the optimizations are the same suggested by Java and Android best practices: use array list instead of vector, use sparse arrays instead of hash map, use arrays instead of collections, use uni-dimensional arrays instead of matrices.

Regarding the GUI, the optimizations reported in the survey are mostly related to reducing unnecessary layout objects, using the ViewHolder pattern [89], using `TextView` with compound drawables instead of a `TextView` and an `ImageView`, and optimizing the images in the GUI. Using static fields, static methods, and final fields (for constants) were mentioned in eight responses. Using `int` instead of other boxed and primitive numerical types, and using no mutable types were mentioned by four of the participants. In spite of avoiding internal getters and setters being promoted by Google as one of the performance tips for Android [1], only four participants mentioned this practice explicitly. Removal of dead/unused code was suggested by only three participants. Finally, bitwise operations were suggested by three participants for long arithmetic operations and games.

We grouped seven optimizations in the "Miscellaneous" category: remove reflection, use custom thread synchronization primitives, use native code when possible, use logical expression shortcuts in conditionals, optimize resources loading, promote method arguments to local variables, and "avoid retaining Activity when application context is enough". For the last one, the context of the answer is not clear; however, we assume it is related to performance, when keeping additional references to Activities as described in SO questions 21644932 [90] and 24186597 [91]. Surprisingly, the micro-optimization we found that improves memory consumption significantly in Section 5 (i.e., removing unused resources) was not mentioned by the participants.

Summary for RQ$_5$. The practice of detecting and implementing micro-optimizations is not prevalent among Android developers who responded to

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

6.4. Threats to Validity

A threat to external validity of the results is the number of participants answering the survey. The number of participants is rather small as compared to the population of Android developers. Therefore, we can not claim the results can be generalized to non-open source developers. It is possible that non-open source developers follow different practices. However, our sample is diverse regarding (i) the programming experience (Fig. 6) in general and specific to Android, and (ii) geographic distribution of the developers (Fig. 7). Moreover, our sample (389) is large compared to mobile developers (with industrial experience) surveyed/interviewed in many previous studies investigating other software engineering phenomenon: 3 in [43], 9 in [92], 45 in [93], 83 in [94], 200 (188 developers + 12 experts) in [95], and 485 in [20].

Concerning the internal validity, one threat is that the coding was done on the open questions by one of the authors. To deal with this threat, the coding was performed using an open coding approach with several coding phases until saturation was achieved; moreover, the coding was done by the author with experience in Android development and usage of Android tools for performance optimization.

7. Conclusions

In this paper, we presented three empirical studies aimed at identifying the state of practice of micro-optimizations in Android apps. The results of the

mining-based study show that implementing micro-optimizations is not a common practice in Android open source projects; we found a significant number of micro-optimization opportunities that are not implemented by the developers in the analyzed projects. Moreover, we found empirically, in a sample of eight apps with a large number of micro-optimization opportunities, that one (out of 15) micro-optimization (remove unused resources) can significantly reduce the memory consumed by Android apps. This micro-optimization opportunity is also the third most prevalent in the current versions of the 3.5K+ analyzed apps, with around 35K+ detected instances, therefore, Android developers should consider to "remove unused resources" as a good practice that has significant impact on Android apps performance. This second study suggests that only one micro-optimization (out of 15 empirically evaluated) has significant impact on the memory consumption of Android apps. In particular, we found evidence that developers can reduce the memory consumption of Android apps, by removing unused resources, which can be automatically detected by the Android LINT tool. On the other hand, this result demonstrates that the remaining 14 micro-optimizations do not appear to significantly improve CPU or memory consumption of Android apps (even in those apps containing a high number of warnings).

Additionally, we surveyed 389 Android developers and their answers confirmed the results of the other two studies; the participants responded against implementing micro-optimizations in their apps. Some of the reasons behind this phenomenon are lack of knowledge of the micro-optimizations and support tools; some apps are considered by developers too simple to require micro-optimizations; and lack of confidence in a practice of "premature optimization" that do not have enough impact on the app's performance. In fact, developers prefer dynamic analysis and observation-based techniques to detect opportunities for improving the performance.

However, some of the surveyed developers reported successful stories, when using a specific set of optimizations depicted in Figure 8. This highlights and suggests that the impact of micro-optimizations is noticeable under certain

"load" conditions (e.g., impact of String micro-optimizations is noticeable when a considerable number of String operations are implemented by a system) that might appear only on specific types of apps, e.g., games that require a lot of objects and floating-point operations to render the GUI.

This paper sheds some light on the impact of the micro-optimizations and opens a research path to empirically verify and understand the benefits of the optimization practices in Android apps. Consequently, future work will be devoted to verifying the impact of a larger set of micro-optimizations and design tools that provide Android developers with better support for detecting useful optimization opportunities and performance bottlenecks.

References

[1] Android performance tips. `http://developer.android.com/training/articles/perf-tips.html`.

[2] Best practices for performance. `http://developer.android.com/training/best-performance.html` [online].

[3] Google, Using ddms. `http://developer.android.com/tools/debugging/ddms.html`.

[4] Eclipse memory analyzer (mat). `https://eclipse.org/mat/` [online].

[5] Lint. `http://developer.android.com/tools/help/lint.html`.

[6] Pmd. `http://pmd.sourceforge.net`.

[7] Findbugs. `http://findbugs.sourceforge.net/`.

[8] Y. Liu, C. Xu, S.-C. Cheung, Characterizing and detecting performance bugs for smartphone applications, in: 36th International Conference on Software Engineering (ICSE'14), 2014, pp. 1013–1024.

[9] C. Guo, J. Zhang, J. Yan, Z. Zhang, Y. Zhang, Characterizing and detecting resource leaks in android applications, in: Automated Software Engineering

(ASE), 2013 IEEE/ACM 28th International Conference on, 2013, pp. 389–398.

[10] N. Ayewah, W. Pugh, The google findbugs fixit, in: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA '10, ACM, New York, NY, USA, 2010, pp. 241–252. doi:10.1145/1831708.1831738.
URL http://doi.acm.org/10.1145/1831708.1831738

[11] S. Wagner, J. Jürjens, C. Koller, P. Trischberger, Comparing bug finding tools with reviews and tests, in: F. Khendek, R. Dssouli (Eds.), Testing of Communicating Systems, Vol. 3502 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 40–55.

[12] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, M. Vouk, On the value of static analysis for fault detection in software, IEEE Transactions on Software Engineering 32 (4) (2006) 240–253.

[13] A. Vetro, M. Torchiano, M. Morisio, Assessing the precision of findbugs by mining java projects developed at a university, in: 7th IEEE Working Conference on Mining Software Repositories (MSR'10), 2010, pp. 110–113.

[14] A. Vetro, M. Morisio, M. Torchiano, An empirical validation of findbugs issues related to defects, in: 15th Annual Conference on Evaluation Assessment in Software Engineering (EASE 2011), 2011, pp. 144–153.

[15] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, W. Pugh, Using static analysis to find bugs, IEEE Software 25 (5) (2008) 22–29.

[16] S. Zaman, B. Adams, A. E. Hassan, Security versus performance bugs: A case study on firefox, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, ACM, New York, NY, USA, 2011, pp. 93–102. doi:10.1145/1985441.1985457.
URL http://doi.acm.org/10.1145/1985441.1985457

[17] S. Zaman, B. Adams, A. Hassan, A qualitative study on performance bugs, in: Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on, 2012, pp. 199–208. doi:10.1109/MSR.2012.6224281.

[18] G. Jin, L. Song, X. Shi, J. Scherpelz, S. Lu, Understanding and detecting real-world performance bugs, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, ACM, New York, NY, USA, 2012, pp. 77–88. doi:10.1145/2254064.2254075.
URL http://doi.acm.org/10.1145/2254064.2254075

[19] A. Nistor, T. Jiang, L. Tan, Discovering, reporting, and fixing performance bugs, in: 10th Working Conference on Mining Software Repositories (MSR'13), 2013, pp. 237–246.

[20] M. Linares-Vásquez, C. Vendome, Q. Luo, D. Poshyvanyk, How developers detect and fix performance bottlenecks in android apps, in: Proc. ICSME, IEEE, 2015, pp. 352–361.

[21] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, M. Schwalb, An evaluation of two bug pattern tools for java, in: 1st International Conference on Software Testing, Verification, and Validation, 2008, pp. 248–257.

[22] Qj pro. http://qjpro.sourceforge.net.

[23] J. Araújo, S. Souza, M. Valente, Study on the relevance of the warnings reported by java bug-finding tools, IET Journal 5 (4) (2011) 366–374.

[24] T. Bakota, P. Hegedus, P. Kortvelyesi, R. Ferenc, T. Gyimothy, A probabilistic software quality model, in: 27th IEEE International Conference on Software Maintenance (ICSM'11), 2011, pp. 243–252.

[25] P. Hegedus, Revealing the effect of coding practices on software maintainability, in: 29th IEEE International Conference on Software Maintenance (ICSM'13), 2013, pp. 578–581.

[26] T. Bakota, P. Hegedus, I. Siket, G. Ladanyi, R. Ferenc, Software maintenance, reengineering and reverse engineering (csmr-wcre), 2014 software evolution week - ieee conference on, in: Qualitygate SourceAudit: A tool for assessing the technical quality of software, 2014, pp. 440–445.

[27] H. Khalid, M. Nagappan, A. Hassan, Examining the relationship between findbugs warnings and end user ratings: A case study on 10,000 android apps, IEEE Software PP (99).

[28] S. Jeong, K. Lee, S. Lee, S. Son, Y. Won, I/o stack optimization for smartphones, in: USENIX Annual Technical Conference, 2013, pp. 309–320.

[29] H. Kim, D. Shin, Optimizing storage performance of android smartphone, in: 7th International Conference on Ubiquitous Information Management and Communication, 2013.

[30] A. Pathak, Y. Hu, M. Zhang, Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices, in: 10th ACM Workshop on Hot Topics in Networks (Hotnets'11), 2011, p. Article No 5.

[31] P. Vekris, R. Jhala, S. Lerner, Y. Agarwal, Towards verifying Android apps for the absence of no-sleep energy bugs, in: 2012 USENIX conference on Power-Aware Computing and Systems (HotPower'12), 2012.

[32] A. Pathak, A. Jindal, Y. Hu, S. P. Midkiff, What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps, in: 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12), 2012, pp. 267–280.

[33] J. Zang, A. Musa, W. Le, A comparison of energy bugs for smartphone platforms, in: 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS'13), 2013.

[34] S. Hao, D. Li, W. G. J. Halfond, R. Govindan, Estimating Android applications' CPU energy usage via Bytecode profiling, in: First International

Workshop on Green and Sustainable Software (GREENS'12), 2012, pp. 1–7.

[35] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, D. Poshyvanyk, Mining energy-greedy api usage patterns in android apps: an empirical study, in: 11th IEEE Working Conference on Mining Software Repositories (MSR'14), 2014, pp. 2–11.

[36] L. Zhang, M. S. Gordon, R. P. Dick, Z. Morley, P. Dinda, L. Yang, Adel: An automatic detector of energy leaks for smartphone applications, in: Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'12), 2012, pp. 363–372.

[37] G. Hecht, N. Moha, R. Rouvoy, An empirical study of the performance impacts of android code smells, in: Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, ACM, New York, NY, USA, 2016, pp. 59–69. doi:10.1145/2897073.2897100. URL http://doi.acm.org/10.1145/2897073.2897100

[38] Y.-F. Chung, Ch-Y.Lin, C.-T. King, Aneprof: Energy profiling for android java virtual machine and applications, in: IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS'11), 2011, pp. 372–379.

[39] D. Li, S. Hao, W. G. J. Halfond, R. Govindan, Calculating source line level energy information for android applications, in: International Symposium on Software Testing and Analysis (ISSTA'13), 2013, pp. 78–89.

[40] K. Kapetanakis, S. Panagiotakis, Efficient energy consumption's measurement on Android devices, in: 16th Panhellenic Conference on Informatics (PCI'12), 2012, pp. 351–356.

[41] Y. Liu, C. Xu, S. C. Cheung, Where has my battery gone? finding sensor related energy black holes in smartphone applications, in: IEEE International Conference on Pervasive Computing and Communications (PerCom), 2013, pp. 2–10.

[42] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, S. Romansky, Greenminer: A hardware based mining software repositories software energy consumption framework, in: 11th Working Conference on Mining Software Repositories (MSR'14), 2014, pp. 12–21.

[43] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, R. Oliveto, M. Di Penta, D. Poshyvanyk, Optimizing energy consumption of guis in android apps: A multi-objective approach, in: 10th Joint Meeting of the European Software Engineering Conference and the 23rd ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'15), 2015, pp. 143–154.

[44] A. Hindle, Green mining: Investigating power consumption across versions, in: 34th International Conference on Software Engineering (ICSE'12), 2012, pp. 1301–1304.

[45] A. Hindle, Green mining: A methodology of relating software change to power consumption, in: 9th IEEE Working Conference on Mining Software Repositories (MSR'12), 2012, pp. 78–87.

[46] C. Sahin, L. Pollock, J. Clause, How do code refactorings affect energy usage?, in: 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'14), 2014.

[47] M. Gómez, R. Rouvoy, B. Adams, L. Seinturier, Mining test repositories for automatic detection of ui performance regressions in android apps, in: Proceedings of the 13th International Conference on Mining Software Repositories, MSR '16, ACM, New York, NY, USA, 2016, pp. 13–24. doi:10.1145/2901739.2901747.
URL http://doi.acm.org/10.1145/2901739.2901747

[48] C. Sahin, L. Pollock, J. Clause, From benchmarks to real apps: Exploring the energy impacts of performance-directed changes, Journal of Systems and Software 117 (2016) 307 – 316. doi:http:

//dx.doi.org/10.1016/j.jss.2016.03.031.
URL http://www.sciencedirect.com/science/article/pii/S0164121216000893

[49] Y. Lin, S. Okur, D. Dig, Study and refactoring of android asynchronous programming, in: 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), 2015, pp. 224–235.

[50] C. Sahin, P. Tornquist, R. McKenna, Z. Pearson, J. Clause, How does code obfuscation impact energy usage?, in: IEEE International Conference on Software Maintenance and Evolution (ICSME'14), IEEE, 2014, pp. 131–140.

[51] J. Gui, S. Mcilroy, M. Nagappan, W. G. Halfond, Truth in advertising: The hidden cost of mobile ads for software developers, in: 37th International Conference on Software Engineering (ICSE'15), 2015, p. to appear.

[52] How developers micro-optimize android apps – online appendix. http://www.cs.wm.edu/semeru/data/JSS-Android-opt.

[53] Improve your code with lint https://developer.android.com/studio/write/lint.html.

[54] Lint checks. http://tools.android.com/tips/lint-checks.

[55] Pmd rules. http://pmd.sourceforge.net/pmd-4.3.0/rules/.

[56] M. Linares-Vásquez, A. Holtzhauer, C. Bernal-Cárdenas, D. Poshyvanyk, Revisiting android reuse studies in the context of code obfuscation and library usages, in: Working Conference on Mining Software Repositories (MSR'14), 2014, pp. 242–251.

[57] Does use of final keyword in java improve the performance?. http://stackoverflow.com/questions/4279420/does-use-of-final-keyword-in-java-improve-the-performance.

[58] Google, Accesing resources http://developer.android.com/guide/topics/resources/accessing-resources.html.

[59] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, D. Damian, The promises and perils of mining github, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, ACM, New York, NY, USA, 2014, pp. 92–101. doi:10.1145/2597073.2597074.
URL http://doi.acm.org/10.1145/2597073.2597074

[60] Proguard. http://proguard.sourceforge.net/.

[61] T. Smith, Mastering proguard for building lightweight android code. http://www.crashlytics.com/blog/mastering-proguard-for-building-lightweight-android-code/.

[62] L. Gomez, I. Neamtiu, T. Azim, T. Millstein, Reran: Timing- and touch-sensitive record and replay for android, in: International Conference on Software Engineering (ICSE'13), 2013, pp. 72–81.

[63] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, D. Poshyvanyk, Mining android app usages for generating actionable gui-based execution scenarios, in: MSR 15, 2015, p. to appear.

[64] M. Linares-Vásquez, Enabling testing of android apps, in: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 763–765.
URL http://dl.acm.org/citation.cfm?id=2819009.2819160

[65] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, D. Poshyvanyk, Fusion: A tool for facilitating and augmenting android bug reporting, in: ICSE'16, 2016, p. to appear.

[66] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, D. Poshyvanyk, Automatically discovering, reporting and reproducing android application crashes, in: ICST'16, 2016, pp. 33–44.

[67] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, D. Poshyvanyk, Auto-completing bug reports for android applications, in: ESEC/FSE'15, 2015, pp. 673–686.

[68] Google, Android kitkat: Tools for analyzing memory use. http://developer.android.com/about/versions/kitkat.html#44-tools.

[69] Android memory usage. http://elinux.org/Android_Memory_Usage.

[70] D. Sheskin, Handbook of Parametric and Nonparametric Statistical Procedures., second edition Edition, Chapman & Hall/CRC, 2000.

[71] R. J. Grissom, J. J. Kim, Effect sizes for research: A broad practical approach, 2nd Edition, Lawrence Earlbaum Associates, 2005.

[72] StackOverflow, Resources pre-load. http://stackoverflow.com/questions/18341126/does-android-load-all-resources-when-app-starts.

[73] StackOverflow, Question. http://stackoverflow.com/questions/14025468/in-an-android-app-when-are-resources-loaded-to-memory.

[74] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, A. Hindle, Energy profiles of java collections classes, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 225–236. doi:10.1145/2884781.2884869.
URL http://doi.acm.org/10.1145/2884781.2884869

[75] Android version market share distribution among smartphone owners as of september 2016. https://www.statista.com/statistics/271774/share-of-android-platforms-on-mobile-devices-with-android-os.

[76] J. A. Maxwell, Qualitative research design : an interactive approach, Sage Publications Thousand Oaks, Calif, 1996.

[77] J. Corbin, A. Strauss, Grounded theory research: Procedures, canons, and evaluative criteria, Qualitative Sociology 13 (1) (1990) 3–21.

[78] Google, Device monitor. http://developer.android.com/tools/help/monitor.html.

[79] Google, Profiling with traceview and dmtracedump. http://developer.android.com/tools/debugging/debugging-tracing.html.

[80] Google, Systrace. http://developer.android.com/tools/help/systrace.html.

[81] Valgrind.http://valgrind.org/.

[82] Littleeye. http://www.littleeye.co/.

[83] At&t application resource optimizer. http://developer.att.com/application-resource-optimizer.

[84] Intel parallel studio. https://software.intel.com/en-us/intel-parallel-studio-xe.

[85] Mobile gaming & graphics (adreno) tools and resources.https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno/tools-and-resources.

[86] How expensive findViewById ??. https://groups.google.com/forum/?fromgroups=#!topic/android-developers/_22Z90dshoM.

[87] Hprof: A heap/cpu profiling tool. http://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html.

[88] Java antipatterns. http://www.odi.ch/prog/design/newbies.php.

[89] Google., Making listview scrolling smooth. http://developer.android.com/training/improving-layouts/smooth-scrolling.html.

[90] Holding context, activity or views as member of a class is bad performance?.http://stackoverflow.com/questions/21644932/holding-context-activity-or-views-as-member-of-a-class-is-bad-performance.

[91] How to hold reference to context object in android library. http://stackoverflow.com/questions/24186597/how-to-hold-reference-to-context-object-in-android-library.

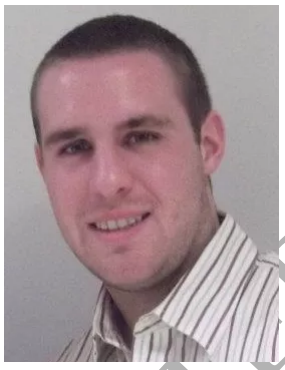
[92] M. Miranda, R. Ferreira, C. R. B. de Souza, F. Figueira Filho, L. Singer, An exploratory study of the adoption of mobile development platforms by software engineers, in: Proceedings of the 1st International Conference on Mobile Software Engineering and Systems, MOBILESoft 2014, ACM, New York, NY, USA, 2014, pp. 50–53. doi:10.1145/2593902.2593915. URL http://doi.acm.org/10.1145/2593902.2593915

[93] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, D. Poshyvanyk, The impact of api change- and fault-proneness on the user ratings of android apps, IEEE Transactions on Software Engineering 41 (4) (2015) 384–407. doi:10.1109/TSE.2014.2367027.

[94] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, D. Lo, Understanding the test automation culture of app developers, in: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–10.

[95] M. E. Joorabchi, A. Mesbah, P. Kruchten, Real challenges in mobile apps, in: ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13), 2013, pp. 15–24.

Mario Linares-Vásquez is an Assistant Professor at Universidad de los Andes in Colombia. He received his Ph.D. degree in Computer Science from the College of William and Mary in 2016. He received his B.S. in Systems Engineering from Universidad Nacional de Colombia in 2005, and his M.S. in Systems Engineering and Computing from Universidad Nacional de Colombia in 2009. His research interests include software evolution and maintenance, software architecture, mining software repositories, application of data mining and machine learning techniques to support software engineering tasks, and mobile development. He is member of the IEEE and ACM.



Christopher Vendome is a fourth year Ph.D. student at the College of William & Mary. He is a member of the SEMERU Research Group and is advised by Dr. Denys Poshyvanyk. He received a B.S. in Computer Science from Emory University in 2012 and he received his M.S. in Computer Science from The College of William & Mary in 2014. His main research areas are software maintenance and evolution, mining software repositories, software provenance, and software licensing.

Michele Tufano received the master's degree in Computer Science from the University of Salerno, Italy. He is currently working toward the PhD degree at the College of William and Mary, Virgina, USA, under the supervision of Prof. Denys Poshyvanyk. His research interests include software engineering, mining software repositories, software quality, software maintenance and evolution, and empirical software engineering. He is a student member of the ACM.

Denys Poshyvanyk is an Associate Professor at the College of William and Mary in Virginia. He received his Ph.D. degree in Computer Science from Wayne State University in 2008. He also obtained his M.S. and M.A. degrees in Computer Science from the National University of Kyiv-Mohyla Academy, Ukraine and Wayne State University in 2003 and 2006, respectively. His research interests are in software engineering, software maintenance and evolution, program comprehension, reverse engineering, software repository mining, source code analysis and metrics. He is a member of the IEEE and ACM.