

A Comprehensive Study on Software Aging across Android Versions and Vendors

Domenico Cotroneo · Antonio Ken
Iannillo · Roberto Natella · Roberto
Pietrantuono

Received: date / Accepted: date

Abstract This paper analyzes the phenomenon of software aging – namely, the gradual performance degradation and resource exhaustion in the long run – in the Android OS. The study intends to highlight if, and to what extent, devices from different vendors, under various usage conditions and configurations, are affected by software aging and which parts of the system are the main contributors. The results demonstrate that software aging systematically determines a gradual loss of responsiveness perceived by the user, and an unjustified depletion of physical memory. The analysis reveals differences in the aging trends due to the workload factors and to the type of running applications, as well as differences due to vendors’ customization. Moreover, we analyze several system-level metrics to trace back the software aging effects to their main causes. We show that bloated Java containers are a significant contributor to software aging, and that it is feasible to mitigate aging through a micro-rejuvenation solution at the container level.

Keywords Android · Operating Systems · Software Aging · Stress Testing

1 Introduction

With mobile devices becoming crucial for our everyday activities, the need for designing reliable and high-performance software for smartphones is well recognized. At the same time, the numerous new functions required to satisfy the emerging customers’ needs, along with the short time-to-market, greatly impact the size, complexity and, ultimately, the quality of the delivered software. This turns into frequent software-related failures, ranging from degraded performance to the device hang or even crash.

Domenico Cotroneo, Antonio Ken Iannillo, Roberto Natella, Roberto Pietrantuono
Università degli Studi di Napoli Federico II
via Claudio 21, 80125, Napoli, Italy
E-mail: {cotroneo, antonioken.iannillo, roberto.natella, roberto.pietrantuono}@unina.it

A common problem, whose impact on end-user quality perception is often underestimated by engineers, is **software aging** (Huang et al., 1995). Software affected by the so-called *aging-related bugs* (ARBs) suffers from the gradual accumulation of errors that induces to progressive performance degradation, and eventually to failure (Grottke and Trivedi, 2007; Machida et al., 2012). Due to such a “subtle” depletion, ARBs are difficult to diagnose during testing; they appear only after a long execution and under non-easily reproducible triggering and propagation conditions. Typical examples include memory leakages, fragmentation, unreleased locks, stale threads, data corruption, and numerical error accumulation, which gradually affect the state of the environment (e.g., by consuming physical memory unjustifiably). The typical solution is to figure out the temporal trend of the degradation, in order to act by preventive maintenance actions known as *rejuvenation*, i.e., solutions to clean and restore the degraded state of the environment (Grottke et al., 2016; Huang et al., 1995). The problem is known to affect many software systems, ranging from business-critical to even safety-critical systems (Araujo et al., 2014; Carrozza et al., 2010; Cotroneo et al., 2014; Garg et al., 1998; Grottke et al., 2006, 2010; Silva et al., 2006).

In this work, we focus on the *Android OS*. Software aging in the Android OS can potentially affect the user experience of millions of mobile products: this OS currently dominates the smartphone market, and has been approaching 50 millions of physical lines of code¹. We conducted an extensive experimental study to highlight potential aging phenomena, to understand the conditions when they occur more severely, and to diagnose their potential source. Indeed, we designed and performed a controlled experiment, grounding on a series of long-running tests, where devices from four different vendors (Samsung, Huawei, LG, and HTC) were stressed and monitored under various configurations.

Results revealed i) systematic trends of performance degradation, which manifest under all the tested Android versions, vendors, and configurations; ii) that software aging is exacerbated by the type of workload (as the degradation trends are more severe under applications from the Chinese market compared to the European one), and by vendor customizations; and iii) that the performance degradation trends are not improving across different Android Versions (from Android 5 to 7): this emphasizes the need to invest more on software aging problems.

By correlating the aging trends to resource utilization metrics, we found that memory consumption is the main issue. In all the cases, the main aging issues can be confined to key processes of the Android OS that exhibit a systematic inflated memory consumption. Specifically, a correlation analysis reveals that the *System Server* process plays a key role in determining the bad performance in terms of responsiveness, and thus should be the main target of a rejuvenation action. Such a result is corroborated by the further analysis

¹ Computed with David A. Wheeler’s SLOCCount (Wheeler, 2016) on the entire Android Open Source Project (AOSP) Nougat 24 (Android Open-Source Project, 2017).

conducted on Garbage Collection metrics, where we observed a significant and systematic increase of collection times of objects of the *System Server* process. Moreover, we present a further experiment to provide more insights about the root causes of the aging, by applying a simple micro-rejuvenation mechanism on selected Java containers. The experiment shows that bloated Java containers indeed are a significant contributor to software aging, and that it is feasible to mitigate aging through a micro-rejuvenation solution at the container level.

This work is an extension of a previous initial analysis of software aging in the Android OS (Cotroneo et al., 2016). Previous studies in this field, including ours (Cotroneo et al., 2016) and work from other researchers (Qiao et al., 2016; Weng et al., 2016), were limited to the analysis of an individual Android vendor and version of the OS; for example, our previous work has been focused on software aging in the Android OS version 5 (Lollipop) from the Huawei vendor. Android vendors introduce significant customizations (*e.g.*, to improve the user experience and stand out against competition), it becomes important to consider many of them, in order to quantify the impact of customizations and the extent of the problem. Moreover, the Android OS has been evolving over several years, with many major versions that revised and extended the architecture of this OS. Several empirical studies showed that these differences among versions and among vendor customizations have a significant impact on both feature-richness and reliability, such as, in terms of compatibility issues (Li et al., 2018; Scalabrino et al., 2019; Wei et al., 2016, 2018) and security vulnerabilities (Gallo et al., 2015; Iannillo et al., 2017; Wu et al., 2013).

For these reasons, it becomes important to consider how software aging impacts on different flavors of the Android OS. Therefore, this paper significantly extends the previous analysis by covering Android devices from four major *vendors* (Samsung, Huawei, HTC, LG), by investigating the differences across them, and extends the analysis to three major *versions* of the Android OS. Moreover, we here provide a more detailed analysis of the root causes of software aging, by tracing back the issue to bloated Java containers.

In Section 2, we first survey the related literature on software aging and mobile device reliability. Section 3 provides an overview of the research questions addressed in this paper. Section 4 describes our experimental methodology. Section 5 presents the experimental results, which are discussed in the conclusive Section 6.

2 Related work

Software aging has been extensively studied since the early '90s (Cotroneo et al., 2014; Huang et al., 1995). One important branch of research has been on *analytical modeling* of systems under software aging, in order to find the optimal time at which to schedule software rejuvenation (Dohi et al., 2000; Grottke et al., 2016; Pfening et al., 1996; Vaidyanathan and Trivedi, 2005), and to perform actions to extend the software lifetime, such as throttling

the workload and allocating more resources (Machida et al., 2016). Another branch of research has been on the *empirical analysis* of software aging, e.g., to measure the impact of software aging effects in real systems (Araujo et al., 2014; Carrozza et al., 2010; Garg et al., 1998; Grottke et al., 2006; Silva et al., 2006) and to get insights on aging-related bugs (Cotroneo et al., 2013a; Grottke and Trivedi, 2007; Grottke et al., 2010; Machida et al., 2012). We here briefly review the key contributions in the empirical branch, to provide background for the experimental methodology of this study.

Software aging metrics and memory consumption. In an early study on software aging in a network of UNIX workstations, Garg et al. (1998) collected resource utilization metrics using SNMP, which included memory, swap, filesystem, and process utilization metrics. They found that many of the failures (33%) were indeed caused by software aging (e.g., resource exhaustion failures), and that *memory* was the resource exhibiting the shortest *time-to-exhaustion* among all the monitored resources. This observation has also been consistently confirmed in subsequent empirical studies, such as in server applications (Grottke et al., 2006), middleware (Carrozza et al., 2010), and cloud management software (Araujo et al., 2014).

Trend analysis for software aging. Statistical hypothesis testing (in particular, the *Mann-Kendall test* and the *seasonal Kendall test*) have often been used (e.g., by Garg et al. (1998) and Grottke et al. (2006)) to point out *degradation trends* in resource utilization, in order to understand whether the time series of resource measurements show a gradual increase/decrease. Moreover, regression techniques, such as the *Sen's procedure* (Garg et al., 1998) and *autoregressive models* (Grottke et al., 2006), have been applied to forecast the time-to-exhaustion of resources. One important challenge is represented by the variability workload conditions and configuration, as opposed to laboratory tests in which these factors can be controlled. Thus, subsequent studies have adopted machine learning techniques (Alonso et al., 2010), such as *decision trees* and robust time series analysis techniques (Zheng et al., 2014), such as the *Cox-Stuart test* and the *Hodrick-Prescott filter*, to make the aging detection less dependent on assumptions about the types of aging trends (e.g., seasonal or non-linear trends) and about the workload (e.g., by establishing a relationship between workload and aging trends).

Stress testing. Stress testing is a potential means to accelerate the onset of software aging phenomena, by providing high volumes of workload (e.g., a high number of client requests) for a long amount of time. For example, this approach has been used by Silva et al. (2006) and Carrozza et al. (2010) respectively on a web server, a SOA system, and a CORBA middleware, showing that aging can lead to gradual performance degradation in terms of throughput, latency, and success rate of requests.

Aging-related bugs. Empirical research has looked into bugs that have been causing software aging issues in complex software systems. One early study has been conducted on embedded software used for space missions (Grottke et al.,

2010), which pointed out that *Mandelbugs* (i.e., “transient” software faults, which aging-related bugs belong to) are a significant part (higher than 30% on average). This finding has later been confirmed in open-source software projects for the LAMP stack (Cotroneo et al., 2013a), and cloud computing (Machida et al., 2012). Most of the aging-related bugs affect memory management (e.g., memory leaks). Several techniques have been developed to aid software engineers in finding software aging bugs. In particular, most of the research literature on debugging has been focused on memory management bugs. Popular tools for findings *memory leaks* (i.e., objects that have not been deallocated) in C and C++ software include *Valgrind* (Nethercote and Seward, 2007) and *AddressSanitizer* (Serebryany et al., 2012), which traverse at run-time the heap memory, starting from the program’s root set of references (e.g., pointers on the stack) and following the pointers inside the objects to find the unreachable (i.e., leaked) ones. For the Java programming language, which avoids memory leaks with automated garbage collection, research has been focused on *memory bloat*, i.e., unnecessary objects that are still referenced by the program but will not be used, thus increasing the load for memory management and impacting on the performance of the program (Ghanavati et al., 2019). Xu and Rountev (2013) and Jump and McKinley (2007) analyzed in detail memory bloat issues caused by the accumulation of unused objects in Java containers (e.g., the *HashMap* and *ArrayList* data structures), and proposed techniques for analyzing the heap growth over time to identify objects suspected of bloating memory, based on the age, the time-to-last-access, and the contribution to memory consumption. Sor et al. (2013) with the Plumb tool, and Andrzejak et al. (2017), used heap utilization metrics in conjunction with machine learning in order to rank the suspicious objects. To further support developers, Xu et al. (2011) developed *LeakChaser* to allow them to annotate a program with *transactions*, to infer and to check the scope of objects, and with *assertions*, to specify lifetime relationships between objects. Later on, the *PerfBlower* approach by Fang et al. (2015) introduced a domain-specific language, in which developers define *penalty counters* for objects in the program, which are then increased or zeroed at run-time (“amplify” and “deamplify”) when some developer-defined conditions occur.

Software aging in mobile systems. The initial studies on software aging in mobile devices were focused on *user applications*. Araujo et al. (2013) performed stress tests in Android, and pointed out aging trends in popular Android applications, such as Foursquare. Other studies proposed techniques to off-load tasks to the cloud and to perform local application restarts in order to prevent the performance degradation of mobile apps (Wang and Wolter, 2015; Wu and Wolter, 2015), and to profile and to diagnose performance bugs in UI-triggered asynchronous applications (Kang et al., 2015).

More recent studies have been investigating software aging and rejuvenation in the Android OS. The empirical study of bug reports by Qin et al. (2017) found that the percentage of aging-related bugs and Mandelbugs in the Android OS is comparable to other open-source projects of similar size and

complexity. Qiao et al. (2018b) and Xiang et al. (2018) presented model-based rejuvenation studies, respectively based on Markov chains and Stochastic Petri Nets, for app-level and OS-level rejuvenation, by taking into account the life-cycle of mobile applications. Other studies by Huo et al. (2018), Qiao et al. (2018a), and Weng et al. (2017) have instead developed measurement-based techniques for forecasting resource utilization and performance degradation, by applying machine learning techniques.

3 Research questions

In this study, we consider the following research questions:

(RQ1) Is the Android OS affected by software aging effects? Does software aging affect responsiveness and/or resource utilization, and to what extent?

This is the basis to explore software aging problems in Android, and to motivate further research on mitigating such possible problems through better testing or rejuvenation. To answer this question, we consider a wide set of experimental conditions, in terms of workloads and device configurations, that may influence the extent of software aging effects. Moreover, we measure the user-perceived responsiveness of the Android devices over time, to analyze whether software aging has an impact on the user experience. This research question is further divided into two questions, in order to assess the extent of software aging from different perspectives.

(RQ1.1) Is software aging widespread across Android devices from different Android vendors?

This stems from the extensive customizations that Android vendors introduce in their products (Iannillo et al., 2017). Therefore, the open-source version of Android (AOSP) is typically extended by vendors with new graphical interfaces and services to enhance the user experience. However, these customizations can also introduce new software defects, and exacerbate software aging issues. Thus, we consider Android devices from four different vendors, so as to figure out whether software aging is a common problem among them and whether vendor customization can worsen software aging issues. It is worth noting that the goal of the analysis is not to make claims about which device vendor is most prone to software aging, which would require a different experimental design than our one to cover multiple devices from the same vendor. Instead, we want to analyze whether the problem of software aging exists across more than one specific vendor, and whether there exists a variability among vendors due to their proprietary customizations.

(RQ1.2) Is software aging widespread across different versions of the Android OS?

This is motivated by the extensive architectural revisions that the Android OS has been undergoing in recent years. The Android OS has been introducing new services, new features for managing security (e.g., stricter permission management mechanisms), new user interfaces (e.g., to improve accessibility), and new mechanisms for managing memory and power utilization. Moreover, vendor customizations also vary across products. Therefore, we compare software aging across different Android versions, in order to investigate whether the evolution of Android had any influence on software aging issues.

(RQ2) Where are the software aging problems that affect the Android OS located?

In order to better understand software aging phenomena, we investigate resource utilization metrics at process- and at task-level, by correlating them with the user-perceived performance degradation. This information is useful to identify where to put more efforts in the Android OS in order to either prevent software aging (e.g., by more extensive tests and debugging) or to mitigate it through software rejuvenation (e.g., by cleaning-up or restarting selected components, rather than the whole Android device).

4 Experimental methodology

To analyze software aging issues in Android, we adopt an experimental methodology based on *stress testing*. In this paper, the term “test” is used to denote an experiment where the system is exercised with an intensive workload for a long period (typically, several hours), in order to increase the likelihood that software aging effects, such as memory leaks, accumulate over time (Carrozza et al., 2010; Grottke et al., 2006; Silva et al., 2006). The test execution “fails” when it exhibits software aging anomalies that lead to poor responsiveness and crash/hang of the system. Test results are then analyzed to identify which conditions exacerbate the occurrence of these failures.

We perform tests under several different conditions, as the extent of software aging effects (e.g., the rate at which the device experiences performance degradation or resource depletion) varies depending on how the system is configured and exercised (Carrozza et al., 2010). For example, in the context of Android OS, different user apps may have a different impact on software aging, as they may trigger different OS services; or different OS configurations (e.g., vendors or versions) may exhibit different software aging effects. However, considering all of the possible combinations of workloads and configurations leads to an extremely high number of long-running experiments, which would take an unfeasible amount of time to complete.

Thus, in order to determine the most influential factors (*e.g.*, workload, device vendor, OS version, etc.) on the software aging trend, we design a test plan for the Android OS. Specifically, we define a set of *factors* (*i.e.*, the parameters of a test) and their possible values (also called *levels* (Montgomery, 2008)). First, we identify the feasible combinations of Android devices and Android versions, since we can not install all the versions in all the devices. Then, we define an experimental design by investigating different combinations of these factors, according to the research questions. We introduce *response variables* to quantify the impact of a test on the target device in terms of software aging, and correlate the factors with the response variable to identify the most influential ones. We consider both *user-perceived* response variables (§ 4.2), and *system-related* response variables (§ 4.3).

The first objective of the data analysis is to assess whether the software aging is present in Android OS, by analyzing the *user-perceived* response variables (*cfr.* RQ1). Once we demonstrate the presence of software aging, we analyze the variance of these response variables with respect to the values of the factors. In particular, we focus on the variances between different Android vendors (*cfr.* RQ1.1) and between different versions of Android OS (*cfr.* RQ1.2). Finally, we analyzed the *system-related* response variables to unveil the underlying components where the software aging phenomenon is internally localized (*cfr.* RQ2).

4.1 Statistical Analysis

We analyze the experimental data using statistical techniques (further discussed in the next subsections), including:

- The Mann-Kendall (MK) test to statistically assess if there is a monotonic trend in a series of the variable of interest over time (Garg et al., 1998). Although it is the most used tests for aging analysis (Cotroneo et al., 2014), it may be subject, like any other hypothesis test, to false positives/negatives (Machida et al., 2013): in order to double-check the results, we apply the following further tests for trend detection (Önöz and Bayazit, 2003). Their effectiveness depends on the shape of the data distribution: *i)* the Cox-Stuart test, *ii)* the *t*-test, and *iii)* the Spearman’s rho test. We claim that there is a trend only if the MK test (which is the most used ones for aging analysis) plus two further tests (out of the three mentioned above) reject the null hypothesis of no trend.
- The Sen’s procedure to compute, in a non-parametric way, the slope of a trend (Sen, 1968; Theil, 1992);
- The Spearman’s rank correlation coefficient to analyze the statistical dependence between two variables of interest (Pirie, 1988).
- The Analysis of Variance (or ANOVA) and the Kruskal-Wallis/Wilcoxon hypothesis test to analyze whether the differences among two sets of experiments are *statistically-significant* (*i.e.*, not simply due to random variations) (Anscombe, 1948; Daniel, 1990).

Under fixed experimental conditions, which reflects our case, the system is expected to exhibit stable performance throughout the experiment: we apply a fixed workload, in which events are generated at a constant rate, and exercise the applications in the same way over and over, where the apps are periodically killed and their resources are freed-up. The experiments have a long duration and the trends are identified through statistical trend analysis in order to account for noise and transient fluctuation.

4.2 User-Perceived Response Variable

To quantify software aging as perceived by users, we focus on the *responsiveness* of the Android OS, as it is a key design goal of this mobile OS. In particular, we analyze the **Launch Time** (LT) of Android Activities (*i.e.*, an application component that provides a GUI screen), which is the period between the request to start an Activity, and the appearance of the Activity on the screen, including the initialization of background and foreground elements. The LT measures the end-to-end latency between the action of a user (starting a new Activity) and the response of the Android OS, across the whole Android software stack (*i.e.*, from the kernel at the lowest level, to the user interface at the highest level). This includes the update of the UI, the initialization of the Android run-time, the allocation of new processes and threads, the registration of a new Activity by the Activity Manager, permission checks by the Package Manager, *etc.*. Indeed, the Android project adopted this metric as an early design goal (“cold-start a basic application, up to a responsive GUI, within 200ms at most” (Tanenbaum and Bos, 2014, ch. 10.8), (Android, 2018b), and clearly correlates the user expectation with the Activities launch times (Android, 2018a). We start our analysis by focusing on the LT metric (rather than resource utilization metrics, that are discussed in the next subsection) since this metric directly impacts on the user, *i.e.*, it is the key performance indicator for the “quality of experience” from the perspective of the users. Resource consumption metrics (memory, CPU) may also exhibit aging trends, but these metrics have an indirect relationship with performance and failures, and have relevance only if there is some user-perceived performance degradation or failure (e.g., if a memory consumption trend does not affect responsiveness or cause failures, then it may be negligible for users and vendors). Thus, we analyze resource utilization metrics in a later stage, to get more insights about responsiveness degradation and the components and causes of software aging (*cfr.* 4.3).

We measure the LT by analyzing the logs from the *Activity Manager* of the Android OS, which is the service responsible for instantiating new activities and to switch among them by saving and restoring their state. We collect these logs using the Android *logcat* utility. As an example, the following line is the log message that shows the “MainActivity” Activity from the application “com.example.myapp”, which took 100 ms to complete its initialization:

I/ActivityManager(1097): Displayed com.example.myapp
--

/. MainActivity: +100ms

During an experiment, if an Activity is retrieved from a cache when the user switches an app, the LT cannot be measured. For this reason, periodical samples of LT are obtained by periodically terminating and restarting workload applications, with a frequency of 1 minute. This is necessary for preventing the OS to cache the Activities.

Moreover, by periodically restarting the apps, we avoid that software aging effects (such as leaked memory) could accumulate inside the apps since our focus is not to study aging of Android apps, but rather the software aging effects in the underlying Android OS.

For each experiment, we analyze the LT to identify any degradation of responsiveness. To this aim, we produce a time series for each experiment using the LT samples of all activities collected during the experiment, and we apply the non-parametric *Mann-Kendall* (MK) statistical test to check whether the time series exhibits a *trend* (Garg et al., 1998), along with the three further trend tests mentioned previously. All of them check the null hypothesis that there is no monotonic trend in the time series, and provides a level of significance (p-value) for the likelihood that the null hypothesis is actually true in the time series. If the p-value is lower than a given α , we can reject the null hypothesis with probability (namely *confidence*) greater than $(1 - \alpha)$, which points out that the LT has been affected by a trend. We require that the confidence should be higher than 95% ($\alpha = 0.05$) for the MK test and for at least two further tests out of the three ones mentioned above.

We mainly rely on the MK test, since it is a non-parametric test, it does not require the measurements to follow a specified distribution or the trend to be linear. However, it is required that measurements are not serially correlated over time. To apply the test, we preliminarily checked for auto-correlation of each data series by means of the *Durbin Watson* test (Durbin and Watson, 1951) at $\alpha = 0.05$: when the test statistic d and the value $(4 - d)$ are greater than the upper critical value, then there is no evidence of positive or negative auto-correlation: in such a case the conventional MK test is applied. In the other cases (*i.e.*, there is evidence of auto-correlation or the test is inconclusive) a modified version of the MK test is adopted, namely the *Hamed and Rao Variance Correction Approach* (Hamed and Rao, 1998). The approach removes the trend from the series and computes the effective sample size significant serial correlation coefficients. A corrected p-value is then provided as an outcome.

If the (modified) MK test indicates the presence of a trend in the LT, we then obtain the *slope* of such trend by applying the *Sen's procedure* (Sen, 1968; Theil, 1992), which is a non-parametric, robust technique that fits a linear model and computes the rate at which the samples increase over time. It simply computes the slope as the median of all slopes between paired values, and it is insensitive to outliers. This approach is often adopted in software aging studies where the system is stressed under fixed conditions, which is likely to lead to a fixed degradation rate, if any (Carrozza et al., 2010; Grottke

et al., 2006; Silva et al., 2006). It should be noted that the outcome of each experiment is not merely based on a single-point observation, in which case there is no clue of the variability of the error caused by possibly repeating the run. It is instead a *trend* computed over hundreds of observations, and its slope value, obtained by the Theil-Sen estimator on the data series of “response times”, is associated with a confidence interval under confidence level of 95% – thus its assessment accounts for the impact of the variability of response times. This is a good compromise between repeating more times the same 6-hours-long run for each of the 72 experiments (which would be more accurate, but too costly), or having only 72 single-point observations with no statistical significance associated with the response variable. The detailed experimental plan is presented in Subsection 4.5.

4.3 System-Related Response Variables

In our analysis, system-related metrics are important to point out which are the most stressed areas of the Android OS that might be causing software aging. System-related metrics include the *memory utilization* (which is the resource most exposed to software aging issue due to memory management bugs; and a scarce one for mobile devices); the *CPU utilization* (which is also exposed to software aging, e.g., due to algorithmic bugs that waste CPU time on bloated data structures); and the *garbage collection duration* (which is a critical activity for the efficient use of memory).

4.3.1 Memory

Many previous experiments on software aging effects indicated that this resource is the most affected one and it tends to have the shortest time-to-exhaustion (TTE) (Carrozza et al., 2010; Cotroneo et al., 2013a; Garg et al., 1998; Grottke et al., 2006; Silva et al., 2006). The Android OS uses complex mechanisms to manage memory, by automatically handling the lifecycle of apps (e.g., collecting resources once an app is not used for a long time, e.g., through the *Low Memory Killer* (LMK) mechanism (Tanenbaum and Bos, 2014)), by recycling processes (e.g., when starting a new Activity), and by managing memory inside applications based on the ART (*Android Runtime*) Java environment. Another potential cause of aging effects in memory utilization is represented by the complexity of the Android OS services, such as *Activity Manager* and *Package Manager*, that are persistent and may accumulate aging effects over time.

We analyze memory utilization through the Android *dumpsys* utility, which reports the memory consumption of the Android OS both in user-space (e.g., the memory used by Android apps and services) and in kernel-space (e.g., Android extensions to the Linux kernel such as the Kernel Samepage Merging, *KSM*, and virtual memory compression, *zram*). We analyze memory consumption of each process of the Android OS, by periodically launching the command

dumpsys every 30 seconds and extracting the *Proportional Set Size (PSS)* of the process, *i.e.*, the footprint of the process on the physical RAM (*e.g.*, not including parts of the process that do not consume memory, such as program code that has not been executed and that still resides on the storage). We focused on this metric because our previous results show that it is strongly correlated to performance degradation trends (Cotroneo et al., 2016).

We check again whether LT degradation is related to per-process PSS metrics. For each PSS series, we perform the following two steps: (i) we test the presence of a trend (and compute its slope) using the four trend tests, *i.e.*, the (modified) Mann-Kendall, Cox-Stuart, *t-test*, Spearman’s rho tests, and the Sen’s procedure; (ii) we compute a *correlation measure* between the slopes of the metric and the slopes of the median LT trend, across all experiments, using the non-parametric *Spearman’s rank correlation coefficient* (Pirie, 1988), since it is robust to outliers and does not make restrictive assumptions on data, contrarily to the parametric counterparts. The correlation points out whether a trend of the metric is systematically accompanied by a degradation trend of the LT.

4.3.2 Garbage Collection

Garbage collection (GC) is a key component of the modern programming environment, as it manages dynamic memory allocations in place of the programmer (*e.g.*, freeing unused area) in order to avoid memory management bugs. However, despite garbage collection, there can still be residual software aging effects: if unused objects are still referenced by the program (*e.g.*, due to poor object handling by programmers), the GC is not able to dispose of the objects, which can accumulate over time (Carrozza et al., 2010; Cotroneo et al., 2013b). This problem is exacerbated by memory fragmentation, which occurs when the program allocates a mix of small and large objects with different lifetimes, causing “holes” in the heap area and increasing the duration of garbage collections and of memory allocations for large objects; and by other bad memory management practices (*e.g.*, frequently re-allocating objects that could instead be reused). Because of these phenomena, GC can significantly degrade the performance perceived by users.

In our experiments, where the device executes under fixed conditions, the GC collection times are expected to be stable throughout the experiment, and the presence of a trend definitely shows an anomaly. In particular, such trends can occur when numerous and scattered objects accumulate in the heap, which the GC will need to walk through object references to reclaim memory. Therefore, the increasing GC time is likely due to the growth and fragmentation of the heap memory. If GC takes more and more time, a process can be noticeably slowed down during GC. This phenomenon has been found in several studies on software aging, which showed that performance degradation in real systems was caused by the higher memory management overhead, which in turn was caused by memory bloat (Carrozza et al., 2010; Grottke et al., 2006).

For the above reasons, we include the duration of GC among system-related metrics. We collect information on GC from the logs of the Android OS, marked with the “art” tag. The Android RunTime (ART) is the Android equivalent for the JVM. The ART reports on GC only in the case that the GC takes much more than usual (in particular, when the *GC Pause Time* exceeds 5 ms or the *GC Duration* exceeds 100 ms). In such a case, the log includes the event that triggered the GC, the GC algorithm, the amount of time spent for the GC, the number of objects freed by the GC, and the available heap memory. We collect these logs as soon as they appear during the experiments. We expect that intensive workloads (such as the ones used by our stress tests) can highlight the effects of poor memory management in Android components, which in turn can result in degraded performance.

The main source of information on GC is the logs provided by the Android Runtime, which can be queried using the *logcat* tool. GC logs are denoted by the “art” tag, as in the following example:

```
I/art: Explicit concurrent mark sweep GC freed
      104710(7MB) AllocSpace objects , 21(416KB)
      LOS objects , 33%free , 25MB/38MB, paused
      1.230ms total  67.216ms
```

The GC metrics are analyzed for each process, by computing trends using the (modified) Mann-Kendall test and the Sen’s procedure. We count the number of cases in which the process exhibited an increase of GC times, which reveals a potential relationship between software aging (in particular, loss of responsiveness) and memory bloat or fragmentation.

4.3.3 CPU and memory utilization at the task level

The Android OS adopts a complex multi-process and multi-threaded architecture to run its several services and components (e.g., to manage a specific hardware resource or provide an API). Since the previous metrics do not provide specific information about individual threads inside a process, we introduce additional metrics to get more insights into the activity of individual services running inside threads.

In the context of the Linux kernel, both processes and threads are internally represented by a *task_struct* object (we use the term *task* in the following of this paper), where a thread is a special type of task that shares certain resources with other tasks (i.e., their *task_structs* point to the same file table, the same page table, etc.) (Love, 2010). Therefore, we analyze CPU and memory utilization metrics for individual tasks. These metrics point out which tasks are mostly active during the onset of software aging effects, and are a potential origin of software aging.

We obtain task-level metrics from the *proc* filesystem of the Linux kernel. In particular, we use the virtual files *schedstat* and *stat* files that are exposed by the Linux kernel (in the virtual folder */proc/TASK_PID/*) to provide information on CPU and memory activity of tasks. These metrics include the

number of *major page faults* (i.e., memory accesses that trigger storage accesses to retrieve the data) and *minor page faults* (i.e., memory accesses that are served by re-using data from the storage cache or shared from other processes), and CPU execution time spent in user-space and kernel-space, which respectively denote the CPU and I/O activity of the task. We periodically sample these task-level metrics every 30 seconds.

To identify critical tasks, we compute trends for each metric and for each task using the (modified) Mann-Kendall test and the Sen’s procedure. Then, we count the number of cases in which a metric exhibited a statistically-significant trend for the task, at a confidence level of 95%. The higher the count, the higher the likelihood that the metric evolves with software aging effects, thus revealing a potential relationship between a task and software aging of the device.

4.4 Factors and Levels

We consider several *factors* to cover different configurations and workloads in the experimental plan. In particular, we define 5 factors and obtain the test plan combining the *levels* of these factors. Factors and levels are summarized in Table 1.

▷ **Device (DEV)**. The Android devices in our experimental setup represent the levels for the DEV factor. We conducted experiments on high-end smartphones from four different, popular vendors. Although market data is highly variable and difficult to gauge, there is a consensus among market analysts that Samsung and Huawei are the two most popular Android vendors, and that their *S* and *P* series were sold at high volumes; for example, see the analysis by the Counterpoint market intelligence firm (GSMArena, 2019). LG and HTC have also been very popular brands. Both of them have been producers of smartphones for Google devices, including the Nexus LG analyzed in this work, and more recently HTC for the Pixel series. The Nexus device is an interesting target for experimentation, since these devices are designed by Google to be “reference” devices for the Android platform, with more timely updates from Google and more lightweight vendor customizations (Wikipedia, 2019). Moreover, the HTC One series has been quite popular until recently with the rise of the competition, and these devices have been designed to be top-quality Android products (Android Central, 2015). In total, we have four levels for the DEV factor, labeled as *HTCONEM9* (*HTC M9 One*), *HUAWEIP8* (*Huawei P8*), *LG NEXUS* (*LG Nexus*), and *SAMSUNGS6EDGE* (*Samsung S6 Edge*).

▷ **Version (VER)**. The Android OS versions available for a device determine the levels for the VER factor. Given the devices that we selected for experimentation, we have three levels for the VER factor: *ANDROID5*, *ANDROID6*, and *ANDROID7*.

According to recent statistics (Android, 2019a), these versions of the Android OS account for more than 50% of Android devices on the market up

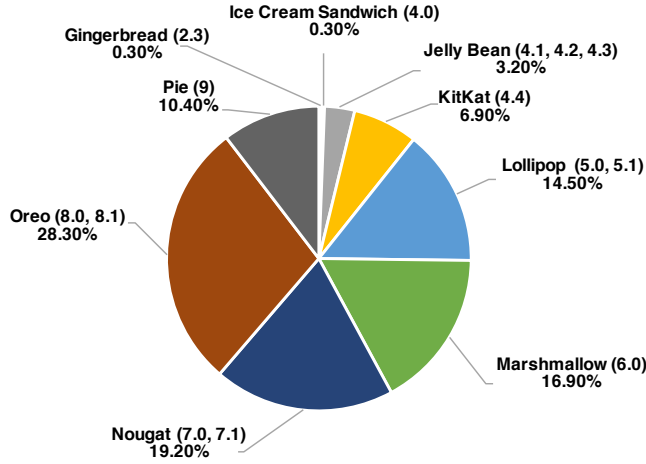


Fig. 1: Distribution of Android OS versions across users (source: <https://developer.android.com/about/dashboards> (Android, 2019a), May 2019).

to May 2019 (Fig. 1). Even if newer releases are available on the market, it is important to note that the adoption of new versions by users has always been quite slow, due to the reluctance of device vendors to provide Android OS updates for a long period, and it is typical for Android devices to receive updates only for just 2 years or even less. In turn, this is caused by the relatively high costs to align a new version of the Android OS with the proprietary customizations by the vendors, and the need for vendors to incentivize users to buy new Android devices. We focused our efforts on versions up to 7 as the Android OS reached a high degree of maturity with this version, after 10 years since the initial version. Therefore, even if the Android OS has continued to be developed so far, there have been no major revisions of the fundamental OS architecture compared to past versions (e.g., no major component has been rewritten from scratch as it happened between for the Dalvik VM that was used by Android 4 and lower versions).

It is worth noting that not every level in the *DEV* factor can be combined with every level in the *VER* factor. In our experiments, we only use official Android releases, by flashing factory images publicly released by the device vendors (i.e., the binary blob of the OS to be written on storage memory). Since the vendors only release updates for a limited period of time, not all of the versions 5, 6, and 7 were available for all devices. To force a specific version of Android, we would have needed to install the open-source Android, but that would have not reflected the actual Android running on real devices. If we used different devices to compare the different Android versions, we would also vary the underlying hardware, thus adding uncertainty on whether

the differences are due to the different hardware. Despite the lack of some combinations of versions/devices, the available versions were still enough to allow us to compare multiple devices running the same Android version, and to evaluate more than one Android versions on the same device (see Section 4.5).

▷ **Application Set (APP).** In our experiments, we use different sets of applications as the workload to exercise the Android OS. These apps are selected to be representative of typical usage scenarios (including browsing, making photos, dialing, chatting), and include both stock apps and third-party apps. We include popular Android applications, listed in Table 1. We organized applications in two groups, which represent the two levels of the APP factor: European applications (*EU*), and Chinese applications (*CHINA*), which are obtained respectively from the European version of the Google app market, and from Chinese app markets.²

▷ **Workload Events (EVENTS).** Our workload generator produces a set of UI events to interact with the apps and the Android device. The events include application switch, touch, motion, trackball, and navigation events. This approach is similar to tools for testing UIs in Android, such as *AndroidRipper* (Amalfitano et al., 2012), *Dynodroid* (Machiry et al., 2013), and *Sapienz* (Mao et al., 2016). Our goal is not to find bugs in a specific app, but to stimulate a collection of apps in order to stress the Android OS at managing the apps. We adopt the popular Google’s Android *Monkey* tool (Android, 2019b), which is integrated with the Android OS. The events are generated randomly, and their probability of occurrence is configured by us. The EVENTS factor varies across three levels: *MIXED1*, where half of the events are application switches; *MIXED2*, where half of the events are touches; *MIXED3*, where half of the events are navigation events. In every level, the other half of the events are of the remaining types and are selected according to a uniform random distribution. The events are generated at a high frequency ($500ms$) to stress the device and to trigger aging phenomena within the duration of the experiment (6 hours). Setting higher values (lower frequency) would have prolonged the duration of the experiments, which would have required an excessive experimental cost. Instead, setting a lower value (higher frequency) would have caused too many events that the devices could not be able to handle, since the creation of new activities often can take several hundreds of milliseconds.

▷ **Storage Space Usage (STO).** We execute experiments either with or without available storage (*i.e.*, free space for storing data), as this aspect can impact on some of the services of the Android OS (*e.g.*, by storing photos and videos from the camera). This factor varies between two levels: *FULL*, where 90% of the storage is occupied by filling it with videos and images; and *NORMAL*, where the default amount of storage space is used (*i.e.*, the storage is occupied only by system files and application packages).

² For those applications requiring the user to login, we have manually configured them before the experiment, by signing up and activating the automatic login.

We adopt the one-way *Analysis of Variance* (ANOVA) technique (Anscombe, 1948) to assess which factors impact the response variable in a statistically-significant way. To figure out if (and which type of) ANOVA can be applied, we verified the following assumptions: *i*) independence of treatments; *ii*) normality of residuals; *iii*) homoscedasticity of variances. Independence is assured by resetting the environment at each treatment execution and avoiding human biases by making the entire procedure automatic. As for normality, we apply the *Shapiro-Wilk* test; the null hypothesis states that data came from a normal distribution. Rejecting the hypothesis means that the assumption is not verified. To verify homoscedasticity, we performed the Levene's test. The null hypothesis is that variances of levels of variables are homogeneous; rejecting the hypothesis means that we are dealing with unequal variances. We then apply the following ANOVA tests accordingly: *i*) if the residuals are normally distributed and homoscedastic, then the conventional parametric ANOVA Fisher test is used; *ii*) if the residuals are normally distributed but heteroscedastic, then the Welch ANOVA test is used, which admits unequal variances; *iii*) if residuals are not normally distributed, then the non-parametric Kruskal-Wallis/Wilcoxon test is adopted (Daniel, 1990).

In all the cases, the null hypothesis is that the factor does not impact the response variable, and the p-value indicates again the confidence in rejecting this hypothesis. We conclude that a factor impacts the response variable if the level of confidence is higher than 95%, i.e., the p-value is less than 0.05.

4.5 Experimental plan

We defined an ad-hoc experimental plan by identifying different combinations of the factors to answer our research questions. The experimental plan includes 72 experiments, for a total of 18 days of experimentation. The experiments are listed in Table 2 (grouped by research question) and in Table 3 (sorted from the oldest to the newest Android version).

The experimental plan contains three overlapping sets, where each set can keep out one factor and combines the remaining factors in a full-factorial design. The first set (EXP13~EXP60) covers all of the *DEV* levels, and locks the *VER* factor to *ANDROID6*, since Android 6 Marshmallow is the only version that can be installed on all the devices, allowing us to study the impact of software aging across devices from different vendors (and all other factors with the same level). The second set of experiments (EXP1~EXP24) fixes the *DEV* factor to *HUAWEIP8*, and varies the *VER* factor between *ANDROID5* and *ANDROID6*. The third set (EXP49~EXP72), instead, fixes *DEV* to *SAMSUNG6EDGE* and the *VER* to either *ANDROID6* or *ANDROID7*. These last two sets are used to study the impact of software aging across different versions of the Android OS. In each set, with the only exception of the fixed factor, we consider every possible combination of the levels.

Due to the long duration of the experiments, we adopted an experimental design without replications, namely one long-running test for each of the 72

Table 1: Factors and levels

Factor	Level	Description
DEV	HTCONEM9	HTC One M9 device
	HUAWEIP8	Huawei P8 device
	LGNEXUS	LG Nexus device
	SAMSUNGS6EDGE	Samsung S6 Edge device
VER	ANDROID5	Android 5 (Lollipop)
	ANDROID6	Android 6 (Marshmallow)
	ANDROID7	Android 7 (Nougat)
APP	EU	com.google.android.videos, com.*.camera, com.android.browser, com.android.email, com.android.contacts, com.google.android.apps.maps, com.android.chrome, com.google.android.play.games, com.android.calendar, com.google.android.music, com.google.android.youtube
	CHINA	com.tencent.mm, com.sina.weibo, com.qiyi.video, com.youku.phone, com.taobao.taobao, com.tencent.mobileqqi, com.baidu.searchbox, com.baidu.BaiduMap, com.UCMobile, com.moji.mjweather
EVENTS	MIXED1	mostly switch events
	MIXED2	mostly touch events
	MIXED3	mostly navigation events
STO	FULL	90% of storage space usage
	NORMAL	default storage space usage

Table 2: Experiments Analyzed to Answer the Research Questions

RESEARCH QUESTIONS	EXPERIMENTS
(RQ1) Is the Android OS affected by software aging effects? Does software aging affect responsiveness and/or resource utilization, and to what extent?	EXP1~EXP72
(RQ1.1) Is software aging widespread across Android devices from different Android vendors?	EXP13~EXP60
(RQ1.2) Is software aging widespread across different versions of the Android OS?	EXP1~EXP24, EXP49~EXP72
(RQ2) Where are the software aging problems that affect the Android OS located?	EXP1~EXP72

experiments. It should be noted that the “response variable” of our experimental design, namely the outcome of a single run, is not a point-value of the metric of interest (e.g., a value of launch time, or memory PSS), but it is a *trend value* (e.g., “launch time trend”), obtained after a long-running test by the Sen’s estimator on the data series of “launch times”, and is itself associated with a confidence interval under confidence level of 95%. The fluctuations of

the metric under analysis (e.g., launch time) within a single run are accounted for, from the statistical point of view, by the trend detection test. When we reject the null hypothesis of “no trend in data” with 95% of confidence, we are saying that the fluctuations of the metric under analysis (e.g., the variance of the metric in that single repetition) are not enough large to make us accept the null hypothesis. The repetitions would allow having a number of *trend* values for that configuration (since repeating the run would provide a different trend), thus accounting for possible fluctuation of the trend (namely, the variance of the trend values across multiple repetitions). We compensate for the replications by applying the Sen’s procedure to estimate the slope of the trend. The procedure provides a probability distribution for the slope, and we consider the median value: this value represents the likely value that would be obtained if we repeat the test multiple times. The longer the time series is when applying the Sen’s procedure, the more reliable is the resulting trend, with a tighter confidence interval, hence the more likely a new repetition will yield a similar trend.

Thus, we calibrated the duration of a test so as to obtain very stable trends (namely tight confidence intervals) as follows. We have analyzed the experimental data from our initial work on software aging in Android (Cotroneo et al., 2016). In that analysis, we conservatively set the duration of tests to 20 hours, since this duration sufficed to make software aging to surface in other systems analyzed by previous experimental studies (Bovenzi et al., 2011; Carrozza et al., 2010). It turned out that, in the cases where we found aging trends, the trend was apparent much before the end of the 20 hours, as 6 hours of data would have been sufficient to detect all trends, and to estimate the trend slope with less than 1% error. Therefore, since in this work we are performing a much larger number of experiments than the past work (Cotroneo et al., 2016), we re-calibrate the duration of tests for this work to 6 hours.

When we investigate each research question (e.g., the impact of the Android OS version), we split our dataset into two partitions, where each partition has the same size with the same experimental configurations, but differs pairwise with respect to the investigated factor. For example, the configuration with device=LG NEXUS, workload=MIXED1, storage=NORMAL, and app=EU, appears in both partitions, differing only with respect to OS versions. Under the null hypothesis that the factor does not influence software aging, the results in the two subsets of experiments should not be significantly different. Otherwise, we report that the factor had a statistically-significant impact.

The devices are controlled and monitored using the *adb* (*Android Debug Bridge*) utility, which is a non-intrusive, dedicated channel through the USB port for debugging purposes. The experimental testbed is shown in Fig. 2.

Table 3: Experimental plan of the case study

ID	DEV	VER	APP	EVENTS	STO
EXP1	HUAWEIP8	ANDROID5	EU	MIXED1	NORMAL
EXP2	HUAWEIP8	ANDROID5	EU	MIXED1	FULL
EXP3	HUAWEIP8	ANDROID5	EU	MIXED2	NORMAL
EXP4	HUAWEIP8	ANDROID5	EU	MIXED2	FULL
EXP5	HUAWEIP8	ANDROID5	EU	MIXED3	NORMAL
EXP6	HUAWEIP8	ANDROID5	EU	MIXED3	FULL
EXP7	HUAWEIP8	ANDROID5	CHINA	MIXED1	NORMAL
EXP8	HUAWEIP8	ANDROID5	CHINA	MIXED1	FULL
EXP9	HUAWEIP8	ANDROID5	CHINA	MIXED2	NORMAL
EXP10	HUAWEIP8	ANDROID5	CHINA	MIXED2	FULL
EXP11	HUAWEIP8	ANDROID5	CHINA	MIXED3	NORMAL
EXP12	HUAWEIP8	ANDROID5	CHINA	MIXED3	FULL
EXP13	HUAWEIP8	ANDROID6	EU	MIXED1	NORMAL
EXP14	HUAWEIP8	ANDROID6	EU	MIXED1	FULL
EXP15	HUAWEIP8	ANDROID6	EU	MIXED2	NORMAL
EXP16	HUAWEIP8	ANDROID6	EU	MIXED2	FULL
EXP17	HUAWEIP8	ANDROID6	EU	MIXED3	NORMAL
EXP18	HUAWEIP8	ANDROID6	EU	MIXED3	FULL
EXP19	HUAWEIP8	ANDROID6	CHINA	MIXED1	NORMAL
EXP20	HUAWEIP8	ANDROID6	CHINA	MIXED1	FULL
EXP21	HUAWEIP8	ANDROID6	CHINA	MIXED2	NORMAL
EXP22	HUAWEIP8	ANDROID6	CHINA	MIXED2	FULL
EXP23	HUAWEIP8	ANDROID6	CHINA	MIXED3	NORMAL
EXP24	HUAWEIP8	ANDROID6	CHINA	MIXED3	FULL
EXP25	HTCONEM9	ANDROID6	EU	MIXED1	NORMAL
EXP26	HTCONEM9	ANDROID6	EU	MIXED1	FULL
EXP27	HTCONEM9	ANDROID6	EU	MIXED2	NORMAL
EXP28	HTCONEM9	ANDROID6	EU	MIXED2	FULL
EXP29	HTCONEM9	ANDROID6	EU	MIXED3	NORMAL
EXP30	HTCONEM9	ANDROID6	EU	MIXED3	FULL
EXP31	HTCONEM9	ANDROID6	CHINA	MIXED1	NORMAL
EXP32	HTCONEM9	ANDROID6	CHINA	MIXED1	FULL
EXP33	HTCONEM9	ANDROID6	CHINA	MIXED2	NORMAL
EXP34	HTCONEM9	ANDROID6	CHINA	MIXED2	FULL
EXP35	HTCONEM9	ANDROID6	CHINA	MIXED3	NORMAL
EXP36	HTCONEM9	ANDROID6	CHINA	MIXED3	FULL
EXP37	LGNEXUS	ANDROID6	EU	MIXED1	NORMAL
EXP38	LGNEXUS	ANDROID6	EU	MIXED1	FULL
EXP39	LGNEXUS	ANDROID6	EU	MIXED2	NORMAL
EXP40	LGNEXUS	ANDROID6	EU	MIXED2	FULL
EXP41	LGNEXUS	ANDROID6	EU	MIXED3	NORMAL
EXP42	LGNEXUS	ANDROID6	EU	MIXED3	FULL
EXP43	LGNEXUS	ANDROID6	CHINA	MIXED1	NORMAL
EXP44	LGNEXUS	ANDROID6	CHINA	MIXED1	FULL
EXP45	LGNEXUS	ANDROID6	CHINA	MIXED2	NORMAL
EXP46	LGNEXUS	ANDROID6	CHINA	MIXED2	FULL
EXP47	LGNEXUS	ANDROID6	CHINA	MIXED3	NORMAL
EXP48	LGNEXUS	ANDROID6	CHINA	MIXED3	FULL
EXP49	SAMSUNGS6EDGE	ANDROID6	EU	MIXED1	NORMAL
EXP50	SAMSUNGS6EDGE	ANDROID6	EU	MIXED1	FULL
EXP51	SAMSUNGS6EDGE	ANDROID6	EU	MIXED2	NORMAL
EXP52	SAMSUNGS6EDGE	ANDROID6	EU	MIXED2	FULL
EXP53	SAMSUNGS6EDGE	ANDROID6	EU	MIXED3	NORMAL
EXP54	SAMSUNGS6EDGE	ANDROID6	EU	MIXED3	FULL
EXP55	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED1	NORMAL
EXP56	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED1	FULL
EXP57	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED2	NORMAL
EXP58	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED2	FULL
EXP59	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED3	NORMAL
EXP60	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED3	FULL
EXP61	SAMSUNGS6EDGE	ANDROID7	EU	MIXED1	NORMAL
EXP62	SAMSUNGS6EDGE	ANDROID7	EU	MIXED1	FULL
EXP63	SAMSUNGS6EDGE	ANDROID7	EU	MIXED2	NORMAL
EXP64	SAMSUNGS6EDGE	ANDROID7	EU	MIXED2	FULL
EXP65	SAMSUNGS6EDGE	ANDROID7	EU	MIXED3	NORMAL
EXP66	SAMSUNGS6EDGE	ANDROID7	EU	MIXED3	FULL
EXP67	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED1	NORMAL
EXP68	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED1	FULL
EXP69	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED2	NORMAL
EXP70	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED2	FULL
EXP71	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED3	NORMAL
EXP72	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED3	FULL

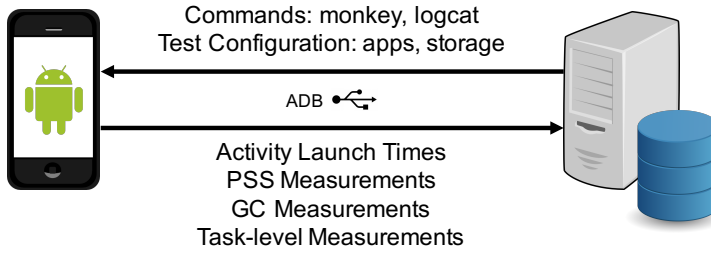


Fig. 2: The Experimental Android Testbed

5 Experimental results

We analyze software aging phenomena with respect to the research questions of Section 3, using the metrics and the experimental plan presented in Section 4. We conclude the analysis with a more detailed study of software aging symptoms in Section 5.3. The dataset and results are available at <https://doi.org/10.6084/m9.figshare.12136134.v1>.

5.1 Software aging across Android vendors

The following experiments aim to assess whether software aging exists in commercial Android devices, and whether aging symptoms vary across different devices. We consider devices from several Android vendors, and we fix the version of the Android OS to 6.

5.1.1 Analysis of Launch Time

To provide insights about the software aging problems found by the experiments, we show in Fig. 3 examples of LT measurements and trends. The figure focuses on a subset of activities from the experiment *EXP39*, where the activities are divided among “browser”, “camera”, “calendar”, and “dialer”. In the figure, the data points are the LT measurements; the lines that cross the data points represent the median slope estimated by the Sen’s procedure; the colored bands around the lines represent the 95% confidence interval for the slope according to the Sen’s procedure. Since the aging phenomenon takes some hours to manifest, and since there are unavoidable random fluctuations and sporadic outliers in the performance measurements, these trends are not immediately apparent from visual inspection, which is the reason why statistical analysis is needed to address the research questions, such as the Mann-Kendall (MK) trend detection test (see subsection 4.2).

We first analyze the experiments to address our first basic research question, that is, whether software aging phenomena can be found in the Android OS. Indeed, this was the case for all of the tested devices: in the majority of

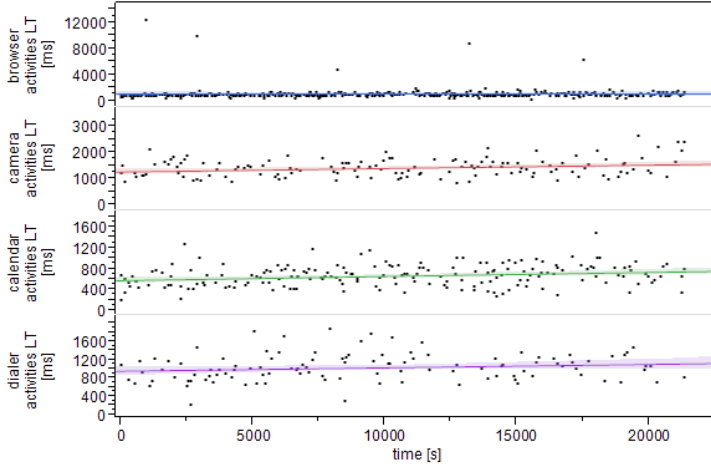


Fig. 3: Groups Activities Launch Time for *EXP39*

the experiments (31 out of 48), the Mann-Kendall test confirmed the presence of a statistically-significant increasing trend in the LT series across all the activities launched during the experiment, which implies that the launch times become longer and longer over time. For 29 out of these 31 cases, at least two further tests among the Cox-Stuart, t -test and Spearman's rho, confirmed the presence of a trend. These trends are summarized in Fig. 4. The data points in the figure represent the slopes of the performance degradation trends obtained by the Sen's procedure, in terms of milliseconds of launch time lost per second. The values greater than zero represent the cases where the series exhibited an increasing trend. In total, there are 48 samples, one for each combination of the factors (EXPs 13 to 60 in Table 3, where the OS version is Android 6). All of the sub-plots show the same data samples from different perspectives, where we split the data samples with respect to different factors (device, application set, workload events, storage space usage). On average, there was an estimated degradation of 380ms of the launch time after 6 hours of testing (i.e., the additional delay for launching apps after that the device has been stress-tested); in the worst case, there was an estimated degradation of 2.5 seconds of the launch time after 6 hours. At the end of some experiments, the devices were so bloated to be unusable, as they reacted to user inputs with very long delays (e.g., more than 10 seconds).

Moreover, we found that the extent of software aging varies across devices from different vendors. Even if the devices all run Android 6, the vendors introduce their own changes to the open-source version, and these changes can have an impact in terms of software aging. We applied the one-way ANOVA technique to assess whether the experimental factors (see Table 1) had an influence on the extent of the performance degradation trends. Table 4 reports the detailed results of both the assumptions verification and the p-value for all the ANOVA tests for this work. According to the ANOVA, the device vendor

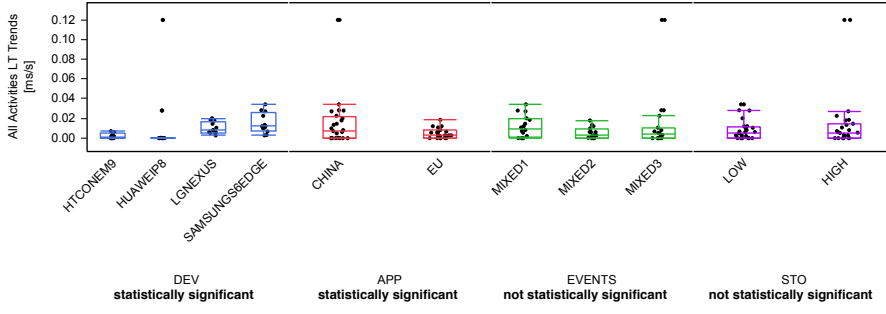


Fig. 4: Distribution of the Launch Time Trends, with all vendors and fixed to Android 6 (EXP13~EXP60)

(*DEV*) is a factor that determines statistically-significant differences in the Launch Time, with a high confidence level ($p\text{-value} = 0.0460$). By means of visual analysis of the boxplots, we can determine that the experiments with *HUAWEI P8* yielded the lowest trends, while the *SAMSUNG S6 EDGE* yielded the highest ones. More importantly, another statistically-significant difference is in the *APP* factor, with a confidence greater than 99% ($p\text{-value} = 0.0044$), where the *CHINA* applications have a worse impact than *EU* in terms of LT. This result suggests that software aging in the Android OS depends on the workload, which can stress different services and subsystems of the Android OS depending on user applications. Moreover, we found that the extent of software aging varies across different devices, as the vendors adopt different flavors of the Android OS with their own proprietary customizations.

5.1.2 Analysis of Memory Usage

Based on the results of our preliminary work (Cotroneo et al., 2016), we focus the analysis on the *PSS* metric collected for four key processes of the Android OS, namely the *System Server*, *Media Server*, *System UI*, and *Surface Flinger*. These processes play an important role in the Android OS:

- The *System Server* is the first Java process that starts at Android OS boot and initializes the rest of the Android Framework. It runs the majority of system services, such as the *Activity Manager*, which manages the life cycle of applications and their activities, and the *Package Manager*, which manages installed packages and security permissions.
- The *Media Server* is the process that hosts most of the media related services, e.g. *Audio Flinger*, *Media Player Service*, *Camera Service*, and *Audio Policy Service*.
- The *System UI* is the process that composes notifications, device status, and device navigation buttons as system bar elements in specific screen areas.

Table 4: Analysis of Variance w.r.t. Normality and Homoscedasticity

H_0	all groups have the same effect on the response variable.						
H_{alt}	the differences in observed effects between groups are unlikely to be due to random chance.						
α	0.05 (i.e., we reject H_0 and accept H_{alt} when p-value is lower than 0.05)						
NORM.	normality (YES if the samples came from a normally distributed population, NO otherwise)						
HOM.	homoscedasticity (YES if the population variances are equal, NO otherwise)						

		SHAPIRO-WILK		LEVENE		ANOVA	p-value
grouped by		P<W	NORM.	P>F	HOM.		
ANDROID 6 LT	DEVICE	<.0001	NO	0.0078	NO	K-W	0.0460
	APP	<.0001	NO	0.0861	YES	K-W	0.0044
	WL	<.0001	NO	0.3325	YES	K-W	0.1428
	FS	<.0001	NO	0.3905	YES	K-W	0.8744
ANDROID 6 PSS systemserver	DEVICE	<.0001	NO	<.0001	NO	K-W	0.0002
	APP	<.0001	NO	0.0142	NO	K-W	0.0033
	WL	<.0001	NO	0.9672	YES	K-W	0.1953
	FS	<.0001	NO	0.8444	YES	K-W	0.8510
ANDROID 6 PSS surfaceflinger	DEVICE	<.0001	NO	0.01346	NO	K-W	0.0003
	APP	<.0001	NO	0.0363	NO	K-W	0.0114
	WL	<.0001	NO	0.6004	YES	K-W	0.6274
	FS	<.0001	NO	0.2157	YES	K-W	0.6966
HUAWEI LT	VERSION	0.0057	NO	0.0562	YES	K-W	0.8961
	APP	0.0005	NO	0.1679	YES	K-W	0.1325
	WL	0.0037	NO	0.6114	YES	K-W	0.9227
	FS	0.0025	NO	0.3738	YES	K-W	0.1649
HUAWEI PSS systemserver	VERSION	0.2046	YES	0.0110	NO	WELCH	0.0040
	APP	0.1623	YES	0.0079	NO	WELCH	0.1714
	WL	0.0754	YES	0.6465	YES	FISHER	0.5436
	FS	0.0089	NO	0.7904	YES	K-W	0.7055
HUAWEI PSS surfaceflinger	VERSION	0.1095	YES	0.5162	YES	FISHER	0.0144
	APP	0.6426	YES	0.8697	YES	FISHER	0.2736
	WL	0.1107	YES	0.8754	YES	FISHER	0.8222
	FS	0.1884	YES	0.6679	YES	FISHER	0.7893
SAMSUNG LT	VERSION	<.0001	NO	0.1500	YES	K-W	0.5089
	APP	<.0001	NO	0.4255	YES	K-W	0.3218
	WL	<.0001	NO	0.6222	YES	K-W	0.4657
	FS	<.0001	NO	0.4526	YES	K-W	0.2207
SAMSUNG PSS systemserver	VERSION	0.1368	YES	0.0016	NO	WELCH	0.0220
	APP	0.2010	YES	<.0001	NO	WELCH	0.0023
	WL	0.0003	NO	0.9196	YES	K-W	0.8692
	FS	0.0005	NO	0.5976	YES	K-W	0.9439
SAMSUNG PSS surfaceflinger	VERSION	0.0224	NO	0.0039	NO	K-W	0.2353
	APP	0.1938	YES	0.0634	YES	FISHER	0.0330
	WL	0.0018	NO	0.9644	YES	K-W	0.2437
	FS	0.0027	NO	0.5378	YES	K-W	0.8696

- The *Surface Flinger* process receives window layers (surfaces) from multiple sources (*System UI* included), combines them, and displays them on the screen.

We again performed the one-way ANOVA, using the PSS of these processes. Fig. 5 shows the distribution of PSS trends for the *System Server* and the *Surface Flinger*. These two processes exhibited increasing memory consumption trends. In particular, the *System Server* is the process with the highest trends. Moreover, the *DEV* and the *APP* factors have a statistically-significant influence on the *System Server*, with very high confidence (p-values of 0.0002 and 0.0033, respectively): these trends are especially high in the case of the

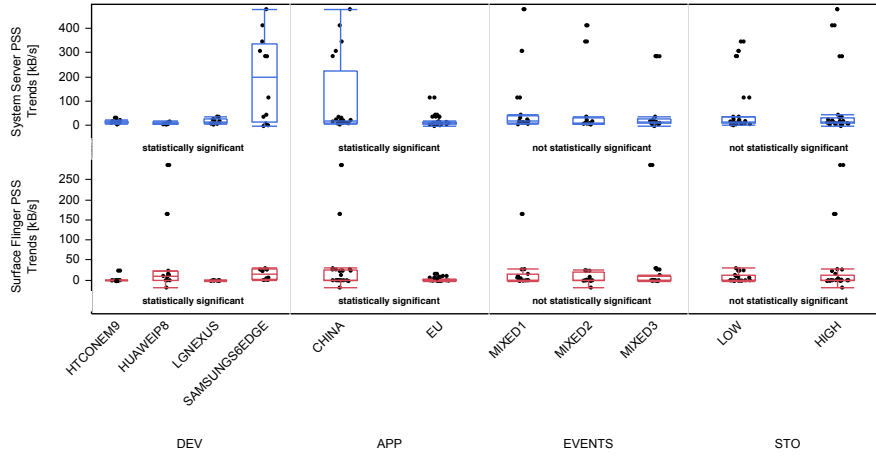


Fig. 5: PSS Trends Distributions: EXP13~EXP60 (Android 6)

SAMSUNGS6EDGE, and of the group of *CHINA* apps. The *EVENTS* and *STO* factors do not have a statistically-significant influence.

The remaining processes (*Media Server* and *System UI*) exhibit negative trends (meaning that, in some cases, memory consumption has been decreasing). Therefore, the memory consumption of these processes does not seem to be related to the performance degradation that we found in the previous analysis. We cross-check this interpretation of the results by jointly analyzing the *PSS* and the *LT* metrics. We compared the memory consumption trend of the processes with the corresponding *LT* trends of the experiments, by correlating these two metrics using the *Spearman's rank correlation*.

Table 5 shows the results of the correlation. Indeed, the memory consumption of the *System Server* and the *Surface Flinger* exhibit a noticeable (and statistically significant) positive correlation with the *LT*, meaning that high *LT* trends (i.e., quicker performance degradation) occur at the same time of high *PSS* trends (i.e., quicker inflation of the memory consumption). Instead, the other processes exhibit a less significant correlation, and even negative. Thus, the memory consumption of the *Media Server* and *System UI* does not seem a possible cause of the performance degradation (i.e., the *LT* trends). Instead, the increasing memory consumption of the *System Server* (which has an important role in starting and managing activities through the *Activity Manager* and *Package Manager*) is a potential symptom of software aging, that we further investigate later in this section (§ 5.3).

5.2 Software aging across Android versions

We analyze software aging across different versions of the Android OS, by looking for differences both between the Android versions 5 and 6 (by locking

Table 5: Spearman correlation coefficients between All Activities LT trends and PSS trends of Android system processes.

PROCESS	SPEARMAN COEFFICIENT	P-VALUE
system (System Server)	0.6467	0.0001
mediaserver (Media Server)	-0.5054	0.0052
com.android.systemui (System UI)	-0.2135	0.91
surfaceflinger (Surface Flinger)	0.6409	0.0002

the *DEV* factor to *HUAWEIP8*), and between the Android versions 6 and 7 (by locking the *DEV* factor to *SAMSUNGS6EDGE*).

5.2.1 Analysis of Launch Time

In the case of the *SAMSUNGS6EDGE* device, we again observed aging trends also for *ANDROID7*, which are shown in Fig. 6. On average, there was an estimated degradation of 650.89ms of the launch time after 6 hours of testing; in the worst case, this estimated performance degradation was up to 5.1 seconds. In the case of the *HUAWEIP8* device, we also notice aging trends in both the versions, as shown in Fig. 7. In this case, the estimated performance degradation at the end of testing has been 291.62ms on average, and 2.6 seconds in the worst case.

Comparing *ANDROID6* with *ANDROID7* (Fig. 6), the LT trends show only small differences in the mean values, and slightly higher variability of the trends for *ANDROID7*. The differences between *ANDROID5* and *ANDROID6* (Fig. 7) are apparently more noticeable, with a reduction of the LT trends in favor of *ANDROID6*. We performed the one-way ANOVA on these two experiments, to assess whether the Android version has a statistically-significant influence. According to the ANOVA, the Android version had no influence in both *HUAWEIP8* and *SAMSUNGS6EDGE*. This result suggests that the revisions to the Android OS are not addressing the areas of the OS that are affected by software aging, and that Android vendors need to invest more effort to address this neglected problem.

5.2.2 Analysis of Memory Usage

Figure 8a and Figure 8b show the PSS trends for the *System Server* and *Surface Flinger* processes, respectively in the case of *HUAWEIP8* and *SAMSUNGS6EDGE*. According to the ANOVA, there were statistically-significant

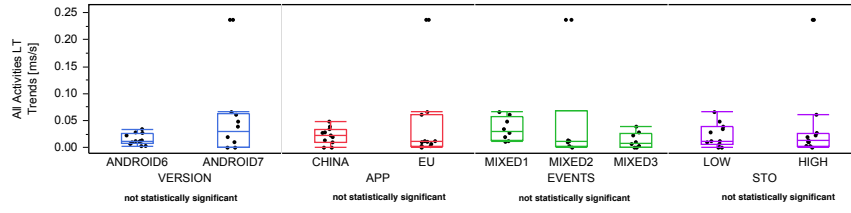


Fig. 6: Launch Time Trends Distributions: EXP49~EXP72 (Samsung S6 Edge)

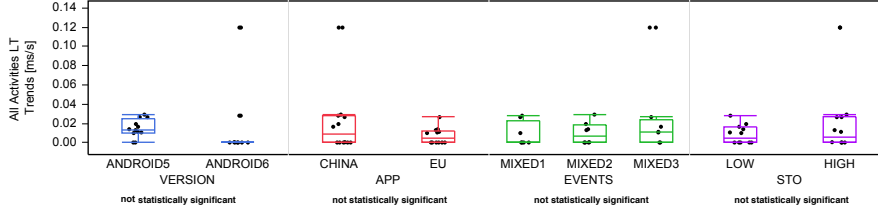


Fig. 7: Launch Time Trends Distributions: EXP1~EXP24 (Huawei P8)

differences between the trends of different Android OS versions. In particular, in the case of *HUAWEIP8* (i.e., the transition from *ANDROID5* to *ANDROID6*), the PSS trends for the *System Server* process gets worse; instead, in the case of the *SAMSUNGS6EDGE* (i.e., the transition from *ANDROID6* to *ANDROID7*), the PSS trends for the *System Server* exhibit an improvement. Considering the results of the previous analysis on LT trends, it seems that the magnitude of LT trends is not impacted by these variations of the PSS trends (i.e., the LT trends are steady even if the PSS trends are different). This result suggests that, probably, it is not only the quantity of memory consumption that influences the performance degradation, but also the way the memory is managed, in terms of fragmentation and lifespan of the objects (see also § 4.3.2). We analyze this aspect in more detail in the next sections.

5.3 Analysis of process internals

In this section, we analyze the Garbage Collection, and task-related events, which provide more information about the internal behavior of Android processes, to get more insights about the reasons of the aging trends discussed in the previous sections.

5.3.1 Analysis of Garbage Collection

As discussed in subsection 4.3.2, we check how many experiments exhibited an anomalous garbage collection activity, as it is an indicator that the software

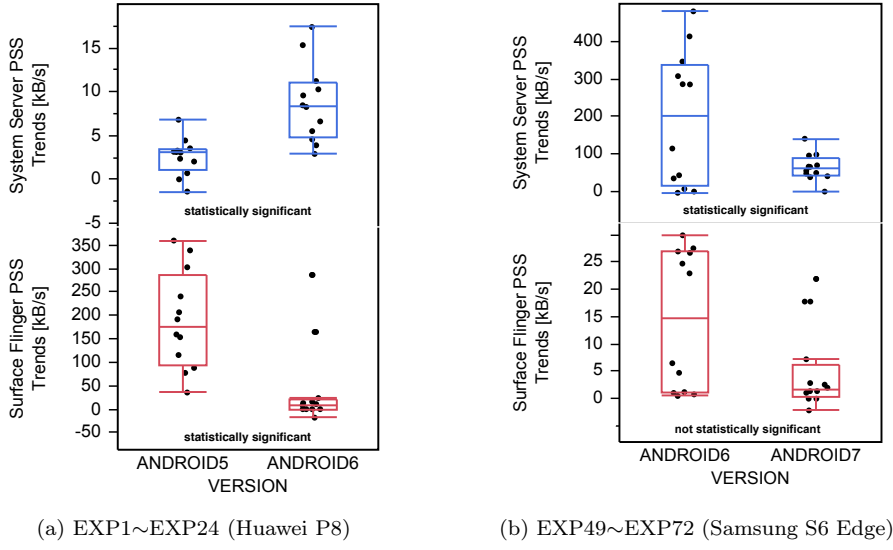


Fig. 8: PSS Trends Distributions:

aging problems are related to memory management problems, such as memory bloat (e.g., stale objects that are still referenced and are not collectable) and memory fragmentation.

As metrics for GC activity, we consider the *GC Pause Time* (i.e., the period that the process is suspended during the GC), and the *GC Duration* (i.e., the total duration, including both the GC that executes when the process is suspended, and the GC that executes in parallel with the program). We separately analyze the different types of GC that are performed by the Android Run-Time (Android, 2016): in particular, in our experiments, only two types of GC produced more than 100 samples and exhibited some trend with confidence higher than 95%, namely *Concurrent* GC (which is performed by a separate thread in background) and *Explicit* GC (where GC is performed on explicit request of a process).

We separately analyze each process of the Android OS, and each of the four GC metrics (GC Duration/Pause Time, and Explicit/Background Collection). We counted the number of experiments in which the garbage collection exhibited a statistically-significant increasing trend. These counts are showed in Fig. 9. In the figure, we ranked the processes with respect to these values, and showed the 5 processes that most frequently exhibited a trend in the GC pause times and GC durations. From the experiments, we noted that such trends occurred most often in the System Server process (which is internally labeled as *system* in the Android OS). This result suggests that this process heavily uses heap memory, and that it is exposed to performance degradation due to the inflation and fragmentation of the heap, which increases the overhead of garbage collections and slows down or freezes its threads. The Android

OS is sensitive to slow-downs of this process: the System Server provides key services for managing the applications' lifecycle (such as the Activity Manager and the Package Manager, which both run as threads inside the System Server, and which are invoked every time that an application is started, disposed of, etc.). Since garbage collection delays the System Server (by freezing or slowing down the process for a short amount of time), these delays propagate to other Android components and affect the responsiveness of the device. Moreover, this result suggests that monitoring the GC times of these processes is another useful indicator to detect software aging in the Android OS, and can be leveraged to trigger software rejuvenation actions when GC becomes too slow or too frequent.

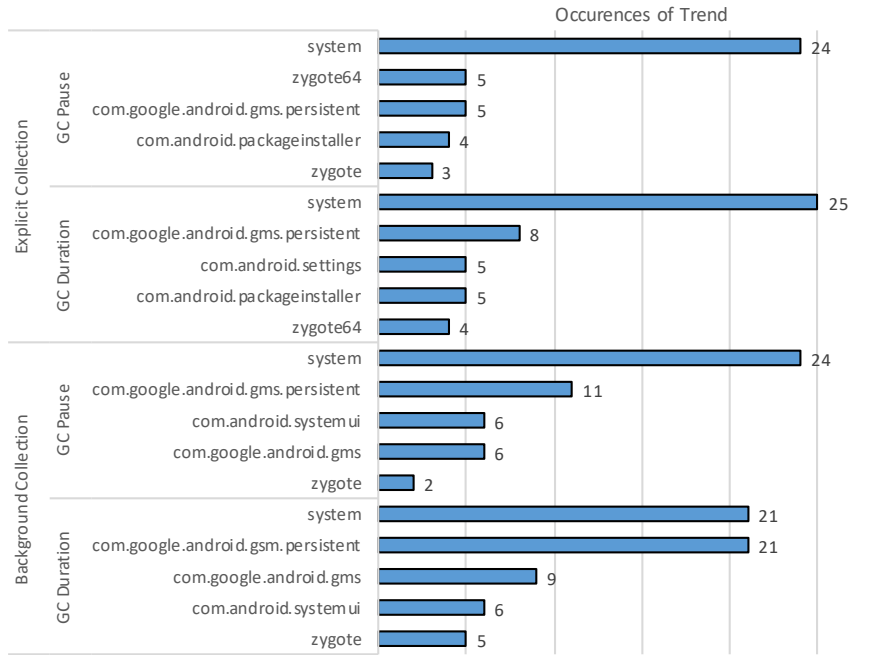


Fig. 9: Top five processes in EXP1~EXP72 that exhibit the greatest number of trend occurrences in each of the 4 garbage collection metrics (GC Duration, GC Pause Time, Explicit Collection, and Background collection times).

5.3.2 Analysis of task-level metrics

Following up on our previous analysis of Android OS processes, we perform a finer-grain analysis of software aging at *task* level (see also 4.3.3). This analysis is useful since the processes of the Android OS contain up to hundreds of tasks

to manage multiple services. For example, both the Activity Manager and the Package Manager run within the System Server process, together with tens of other services. Since process-level metrics (such as the PSS) cannot provide information about individual tasks, we analyze task-level metrics for specific Android services.

For each experiment, and for each group of tasks, we apply the MK trend detection test on the time series of *major faults*, *minor faults*, *kernel time*, and *user time*, as defined in subsection 4.3.3. Then, we counted the number of experiments in which the task group (*i.e.*, at least one task in the group) exhibited an increasing utilization of CPU and memory, and ranked the groups according to the count.

According to the previous results, we focus on task-level metrics for the *System Server* and the *Surface Flinger*. To relate the tasks to Android subsystems, we grouped the tasks with respect to the Android service or subsystem they belong to, according to their names and to our analysis of the Android AOSP source code. For example, the *ACTIVITY* group in *System Server* consists of four threads related to the Activity Manager, namely *ActivityManager*, *ActivityManager_2*, *ActivityManager_3*, and *HwActivityManag*. The value of each group is computed by averaging the trends count of each task of the group. Fig. 10 shows the 10 groups with the highest ranking.

In the case of the *System Server*, the task groups that most frequently exhibit trends include:

- *ALARM*: the tasks that execute the *Alarm Manager* service, that is in charge of setting up timers for the rest of the system.
- *BACKUP*: the tasks that execute the *Backup Manager*, which is notified each time there is new data to be saved persistently (*e.g.*, new contacts in the dialer).
- *ACTIVITY*: the tasks that execute the *Activity Manager* service, which handles requests for managing the lifecycle of Android activities.
- *PACKAGE*: the tasks that execute the *Package Manager* service, which handles requests for forwarding intents and checking permissions.
- *INPUT*: the tasks that read and dispatch user inputs from the hardware devices to the higher layers.
- *NETWORK*: the tasks for scanning and for handling the connection to WiFi networks.

These groups and the others in Fig. 10 represent Android OS tasks that consumed increasing amounts of CPU time (*e.g.*, due to the increasing amount of time spent on traversing bloated data structures, or on garbage collection). For example, the “Activity” and “Alarm” groups show an increasing trend of task-level metrics, which we attribute to memory bloat that develops in these services due to the very large number of activities and notifications generated during the experiments. These tasks are suspected to be causing software aging, and we further analyze them in the next subsection. Moreover, this information points out areas of the Android OS that may be targeted by software rejuvenation: in particular, it is advisable to focus software rejuvenation in the

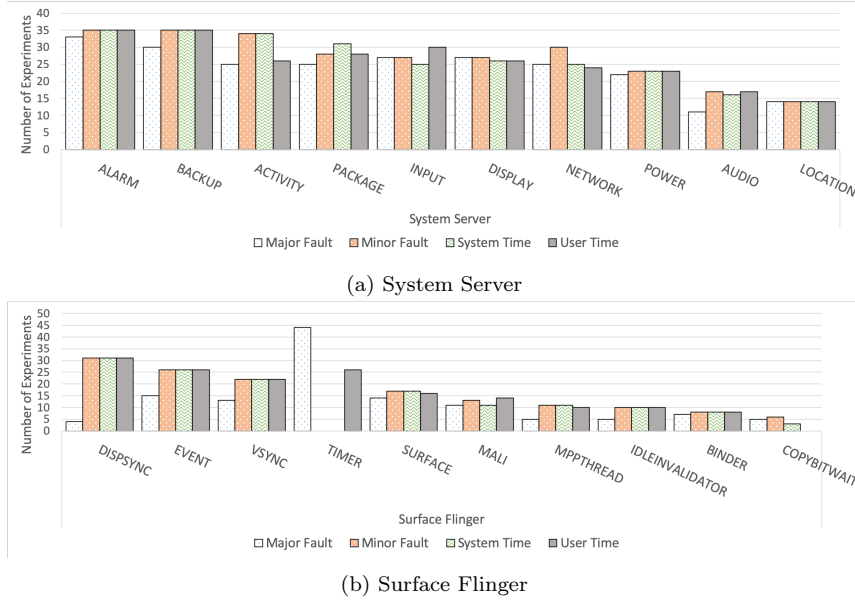


Fig. 10: Top 10 task groups that exhibit the highest number of task-level metric trends in EXP1~EXP72, respectively in the System Server (a) and Surface Flinger (b) processes. Major and minor faults are memory accesses served respectively from the storage, and from the page cache or other processes. User Time and System Time are the CPU time spent by a task running application- (e.g., for CPU-bound activity) or kernel-code (e.g., for I/O activity).

System Server, either at process-level (in order to rejuvenate all of the tasks inside the process) or at task-level (by re-initializing the top-most services in the ranks, in order not to disrupt other services inside the process).

5.4 An experiment with software rejuvenation to counteract memory bloat

According to the previous analyses, the performance degradation of the Android devices was accompanied by higher memory consumption of key Android OS processes (in particular, the *system server* process), and the increasing of garbage collection activity (i.e., longer duration of garbage collections, and longer suspensions of Android OS processes). Moreover, the analysis pointed out that specific Android OS tasks were consuming more and more CPU time. From these experimental facts, we hypothesize that software aging has been mostly caused by bloated Java containers, which is a recurring issue in large Java projects (Ghanavati et al., 2019; Xu and Rountev, 2008, 2010). We performed an additional experiment to verify this hypothesis, in order to gain further insights into the root causes of software aging. In this experiment, we first instrumented the open-source version of the Android OS with a mech-

Table 6: Java containers selected for rejuvenation

Name	Type	Service
mLog	LinkedList<String>	WifiService
mLogRecVector	Vector<LogRec>	WifiService
mPendingProcessChanges	ArrayList<ProcessChangeItem>	ActivityManager
mAvailProcessChanges	ArrayList<ProcessChangeItem>	ActivityManager
mPendingUidChanges	ArrayList<UidRecord.ChangeItem>	ActivityManager
mAvailUidChanges	ArrayList<UidRecord.ChangeItem>	ActivityManager

anism to periodically rejuvenate bloated Java containers; then, we stressed again an Android device, in order to evaluate whether the lower memory bloat has an impact on software aging of the device.

The software rejuvenation mechanism mitigates memory bloat by forcefully removing the contents of selected Java containers, such as by invoking the `.clear()` method of the `ArrayList` class. We focus rejuvenation on the *system server* process, as the previous analysis showed that this process was the one most affected by aging symptoms, and on its tasks that exhibited anomalous CPU and memory activity. For each of these tasks, we inspected the source code of the Android services that are executed by the tasks. For example, in the case of the *Activity Manager* service (which is executed by the tasks labeled as *ACTIVITY* in the previous analysis), we inspected the source code of the `ActivityManagerService` Java class, as this class provides concrete implementations of APIs of the *Activity Manager* service. These classes are instantiated by a single, long-lived object since the initialization of the *system service*, during the boot of the device. In our inspection, we looked for Java containers among member variables of these classes and allocated at object initialization, as these containers are also long-lived and can be potentially affected by memory bloat. Instead, we do not consider short-lived containers (e.g., only limited to the scope of an individual method).

In order to avoid that flushing the container leads to failure of the *system server* process, we also analyze Java containers with disposable data, such as message logs, resource utilization statistics, and information on recently-executed apps. Therefore, the Java containers we selected for rejuvenation (listed in Table 6) are a conservative subset of the containers used by the service. For example, in the case of the *WifiService*, we identified three `ArrayList` containers where the service accumulates message logs about scans WiFi networks and other protocol events. Since these data may cause memory fragmentation and are only useful for debugging purposes, we remove them from the container in order to mitigate software aging.

We remark that we do not aim to develop a fully-fledged software rejuvenation solution, but only to perform a controlled experiment: therefore, we focus rejuvenation on containers that could be rejuvenated with very low risk of side effects on the usability of the device. If the device exhibits less software aging when rejuvenating this subset of containers (i.e., lower performance degradation trends), then the experiment provides additional evidence

that the software aging effects are caused by bloated Java containers. Moreover, our measured performance improvement is a lower bound of the potential improvement that could be achieved by actual Android OS developers, which have the expertise to identify more opportunities for safe rejuvenation of containers with more selective rejuvenation (e.g., only removing stale objects related to apps that were terminated since a long time, or objects that could be easily recreated if necessary in a later moment).

We implemented and evaluated this software rejuvenation mechanism on the open-source version of Android OS (AOSP), as we needed access to source code to introduce the changes. We ran the experiment on the LG Nexus device, which was the only device that could execute the AOSP among the devices in our testbed, since the other devices require proprietary customizations to the Android OS (e.g., device drivers). We ran the experiment on the AOSP version 6.0.1, using the Chinese set of Android apps, with a uniform distribution of application switches, touch, motions, trackballs, and navigation events, and with default storage space usage. The experiment lasted 7 hours, and rejuvenation was triggered periodically every hour.

As in the previous experiments, we periodically restarted the apps to take measurements of the launch time. To evaluate the impact of rejuvenation on performance, we compare the performance degradation trend slopes respectively with rejuvenation (LT^r) and without it (LT):

$$\text{Gain}_{LT}\% = \frac{LT - LT^r}{LT} \cdot 100 \quad (1)$$

For instance, a gain of 40% means that the Launch Time after 6 hours is 40% lower in the with-rejuvenation case compared to the without-rejuvenation case. Moreover, we evaluate software rejuvenation with respect to the *Time to Aging Failure* (TTAF), that is, the period of time after which the launch time of an activity exceeds a limit, over which performance is considered degraded; we set the limit at +200ms with respect to the initial launch time of the activity, as 200ms is the performance goal of basic apps of the Android OS (Tanenbaum and Bos, 2014, ch. 10.8), (Android, 2018b). We compute the expected TTAF with and without rejuvenation ($TTAF^r$, $TTAF$) based on the estimated slopes (LT^r , LT), and evaluate the gain with the following metric:

$$\text{Gain}_{TTAF}\% = \frac{TTAF^r - TTAF}{TTAF} \cdot 100 \quad (2)$$

For instance, if the TTAF without rejuvenation for a given activity was 6 hours, and the TTAF with rejuvenation is prolonged to 8 hours, then the gain is $2/6 \cdot 100 = 33.33\%$.

The experiment was repeated three times. Table 7 reports the averaged results without and with rejuvenation, and the gain metrics in terms of launch time increase and TTAF. Across all activities that exhibited aging trends, the average gain was 46% in terms of launch time increase, and 87% in terms of TTAF. These results point out that flushing the Java containers relieves

heap memory management and improves the responsiveness of services running in the *system server* process. Thus, the rejuvenated Java containers were indeed affected by memory bloat. Moreover, our rejuvenation mechanism did not cause any side effect, without any high-severity error logged either by the *system server* or by the apps. Thus, this experiment points out that software rejuvenation is a viable strategy, which brings a measurable benefit by postponing the effects of software aging.

6 Key findings

We analyzed software aging issues in the Android OS, by performing a large experimental study across several devices, versions, and test configurations. We obtained a large amount of data that provided us information to address the research questions of this work.

(RQ1) Is the Android OS affected by software aging effects? We observed software aging effects in the vast majority of our tests, which thus confirms that the Android OS is indeed affected by software aging effects. We found that aging impacted the responsiveness of the device, as confirmed by the increasing, statistically-significant trends of the launch time of Android activities (*cfr.* Fig. 4, 6, 7). Moreover, we also observed increases in memory consumption of key processes of the Android OS (*cfr.* Fig. 5, 8a, 8b).

(RQ1.1) Is software aging widespread across Android devices from different Android vendors? In our analysis, we considered devices from four leading Android vendors (Samsung, Huawei, LG, and HTC) to address this question. We found that software aging occurs consistently across these four vendors, thus highlighting that software aging is not limited to specific Android devices (*cfr.* Table 4, Fig. 4). We have found that software aging effects are exacerbated by the specific Android vendors, as they apply customizations to the basic Android OS. Moreover, the workload is another factor that significantly contributes to the extent of software aging (*e.g.*, by stressing the parts of the Android OS that are affected by aging-related bugs), as in our tests the Chinese applications revealed higher software aging trends of the Android OS.

(RQ1.2) Is software aging widespread across different versions of the Android OS? Similarly, we analyzed the presence and the variability of software aging across different versions of the Android OS, by considering the three most recent Android releases at the time of writing. We found that all the considered Android versions are affected by software aging, thus pointing out that this problem is not limited to specific versions, but that the problem permeates the Android OS. Moreover, our tests did not show an improvement of the Android OS over time, as the most recent Android release shows aging trends that are comparable to the previous one (*i.e.*, there are no statistically-significant differences, *cfr.* Fig. 6, 7). This finding remarks the need for more extensive tests to fix aging-related bugs, and for software rejuvenation solutions

to mitigate the effects of the (unavoidable) aging-related bugs that get shipped with the products.

(RQ2) Where are the software aging problems that affect the Android OS located? We point out that the software aging effects can be traced back to the most basic elements of the Android OS. The extent of software aging varies with the version of the Android OS, the workload and other conditions; however, the fact that software aging manifested consistently across the experiments suggests that the aging issues are not limited to components exercised by specific use cases, but are part of the fundamental components of the OS that are always exercised by the user. We analyzed a set of metrics inside the Android OS, in order to gain insights on where and how to mitigate software aging. We found that the software aging trends are accompanied by a statistically-significant increase in memory consumption of key Android processes (*cfr.* Fig. 5, 8a, 8b). In particular, we found that the memory consumption of the *System Server* (an Android OS process that runs many of the basic services of this system, including the *Activity Manager* for starting user applications) is significantly correlated with the performance degradation trends (*cfr.* Table 5).

We then analyzed more in detail these processes, by looking at garbage collection and task-level metrics. The analysis pointed out that the *System Server* spends more and more time on garbage collection during the experiments, which is an indicator that the memory utilization becomes more fragmented and “bloated” (i.e., burdened by unnecessary objects) over time (*cfr.* Fig. 9). Moreover, the task-level analysis identified the subsystems of the Android framework that exhibit increasing trends in terms of CPU utilization and virtual memory, which points out that these components are the ones most exercised in our experiments (*cfr.* Fig. 10) and represent candidates for further investigation by developers (*e.g.*, to mitigate aging effects through software rejuvenation).

We applied the insights from this analysis to perform an additional experiment, in order to assess the hypothesis that aging was influenced by memory bloat in Java containers, and to get more insights about the root causes of the aging. We applied a rejuvenation mechanism in the *System Server*, by flushing the contents of selected Java containers. We inspected the source code of the services identified by our previous analysis, in order to locate Java containers that could benefit from software rejuvenation. The experimental results confirmed that the containers were indeed affected by aging, and that the analysis was useful to identify good targets for software rejuvenation. In principle, the same analysis could be performed on new releases of Android to trace back software aging issues, and to introduce software rejuvenation where necessary.

7 Threats to validity

Selection of Android versions and device vendors. As for any empirical study, the experiments can only cover a limited number of devices. We focused

on three major versions of the Android OS and four popular Android devices, as discussed in Section 4.4. Since the Android OS running on commercial devices is not the same of the open-source Android, but it is modified to include proprietary customizations, we could not perform a diversity analysis as proposed by (Nagappan et al., 2013), which would require detailed information about the number of developers, the number of changes, the amount of source code, etc., which is not publicly available. Therefore, in order to achieve a reasonable diversity of the analyzed devices, we relied on devices from different competing vendors on the market.

Number of experimental runs. We clarify that our results are based on the analysis of a single time series, *i.e.*, a single experiment for each considered combination. To have more reliable values, the trends have been checked by a robust trend detection procedure. Beside the MK test, we applied three further statistical tests for trend detection, claiming a trend only if the MK test and two further tests rejected the null hypothesis of no trend. In the average, in 69 out of 72 experiments at least three out of four tests provided the same outcome, confirming the reliability of the trend detection procedure against false positives/negatives. The experiments had a 6-hours duration, with samples collected every 30 seconds. We used the Sen’s procedure on these data to estimate the probability distribution of the trend value, to mitigate the threat of not having repetitions; this estimate represents a theoretical derivation valid under the assumptions of the Sen’s procedure. Although the 95%-confidence intervals for the estimated trends are small, replications would provide more actual values for the trend, and would empirically confirm/reject the theoretical trend’s probability distribution. Due to the high cost of repetitions, we opted to spend experimental time to cover more configurations (our design has 72 experiments) and to have stable trends (by 6-hours duration). Even a few repetitions would require a significant amount of additional resources, not only in terms of time but also in terms of hardware, without significantly impacting the result.

Duration of the experiments. The limited duration of the stress tests is an inevitable limitation of any experimental work. We calibrated the duration of our experiments to 6 hours according to data from our previous work (Cotroneo et al., 2016) since, in retrospect, this duration would have been sufficient to detect all of the aging trends. There is still some risk that this duration has not been enough to point out aging trends for some of the new conditions tested in this work. However, this duration still gives us confidence that we covered the most severe trends in terms of high performance degradation rate and short time-to-aging-failure, since any missed trend would have a much longer time-of-aging-failure and smaller performance degradation.

Workload definition and generation. The workload makes use of a number of applications widely used in both the Chinese and European markets. These include information and content applications (e.g., videos, music, camera, gallery), communication applications (e.g., browsers, email, IM), and basic applications (e.g., contacts, calendar, phone). Not including some types of apps, such as gaming ones, limits the generality of the conclusions: the reader

should be aware of this aspect when generalizing the reported numerical trends if their workload is different from ours.

Moreover, the workload is characterized by fixed configurations of the *EVENTS* factor, where one out of the three types of event is predominant (e.g., motions). Indeed, the types of events can be mixed in several other ways, such as making a workload dominated by two types of events, possibly obtaining different results. On the other hand, we can also conjecture, based on the obtained results, that changing the type of events is expected to have a little impact: in fact, the factor *EVENTS* turned out to be not statistically significant in our analysis, hence we expect that changing the percentages would have little impact.

To generate the workload, we adopted the *Monkey* tool, which generates pseudo-random streams of user events. Hence the final result of a single run could be affected by the *seed* selected for the pseudo-random event generation, as the seed determines the sequence of events. On the other hand, it should be noted that the possible occurrence of aging symptoms in a given configuration (that is, an experiment) does not depend on a single UI event, but on the cumulative effect of highly numerous events: the duration (6 hours) and the number of events (an event each 500ms) are so large that, given a fixed percentage of events of each type, the impact of the ordering of their execution becomes negligible. Indeed, a sensitivity analysis aimed at investigating the impact of the time between two consecutive events (500ms) and/or of the experiment duration on the extent of aging manifestation is an interesting future direction to study the phenomenon accounting for these factors too.

About the events generation, it should be noted that the effect of the randomly generated events could also interact with the effect caused by the other studied factors. The ANOVA we conducted on the controlled factors is a “one-way” ANOVA: this means that the possible effects of the interactions among multiple factors are not accounted for. This is a typical approach, since interactions usually explain a small part of variation of results. Conducting further experiments to run a two-way or n -way ANOVA could point out such interactions.

We also warn the reader against considering the reported aging trends as the actual values expected in operation. The adopted workload (*i.e.*, the chosen apps, the time between events, etc.) is not meant to be representative of some user behaviour, because, in order to observe and clearly distinguish the possible aging effects (from the workload), a fixed and stressful workload is preferred. This has the goal of highlighting possible aging problems, so as to perform a root cause analysis and debugging. Hence, results highlight that there are indeed potential aging problems, but the real trends will depend on real workload, hence on the user behaviour.

8 Lessons learned and conclusion

According to the results of our experimental analysis, we derive the following lessons learned and recommendations on mitigating aging for researchers and developers.

Performing stress tests, as in our experiments, is useful for developers as a starting point to **localize and fix software aging issues in the source code of the Android OS**. For example, the processes and services with anomalous GC times and CPU/virtual memory utilization should be scrutinized with more emphasis, by performing a detailed memory profiling of these services to identify stale objects and other bad memory management patterns. Since profiling is a heavy activity and since the Android OS has an extensive code base (several millions of lines of code), it is important to restrict profiling to the components that are more prone to aging-related issues.

Another approach is to apply software rejuvenation strategies, since the underlying software aging bugs may be too costly to be fixed (e.g., due to the long time that is needed to reproduce them). Our stress tests show that **experimental data can point out aging-prone components** that are causing performance degradation and resource utilization, and that are candidates for rejuvenation actions. Since software rejuvenation comes with a cost (e.g., in terms of periods of time during which the device is not usable, or slowed-down, due to rejuvenation), it is important to keep low this cost in order to improve the quality of experience of Android users, by targeting rejuvenation on selected components. Knowing the Android processes most prone to software aging (such as the System Server) allows developers to focus engineering efforts to implement rejuvenation actions at the process-level, such as to introduce mechanisms to preserve any critical state during the restart of a process, such that no Android applications or other system processes are affected.

In our analysis, we presented an experiment with a simple micro-rejuvenation mechanism that periodically flushes selected Java containers to counteract memory bloat, without restarting neither the Android device nor the process that runs the service. The experimental results showed that the Android OS can indeed benefit from rejuvenation in terms of less degradation of performance. Moreover, the analysis of software aging symptoms (e.g., CPU and memory utilization by process and by task, garbage collection) was proven to be useful to identify potential candidates for software rejuvenation. Such rejuvenation mechanisms are an interesting direction for future research, as we only limited to a very small subset of containers that could be safely rejuvenated with low programming efforts.

Rejuvenation can focus on the services that are most used and/or that exhibit a higher utilization CPU and memory, such as the services of the System Server and Surface Flinger that were pointed out by our analysis. The **rejuvenation actions should be triggered with a measurement-based approach**, by monitoring the memory utilization and garbage collection as

we did in our tests (e.g., by sampling the PSS and collecting logs from the ART) in order to detect the onset of software aging effects.

References

- Alonso J, Torres J, Berral JL, Gavalda R (2010) Adaptive On-Line Software Aging Prediction based on Machine Learning. In: Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on, IEEE, pp 507–516
- Amalfitano D, Fasolino AR, Tramontana P, De Carmine S, Memon AM (2012) Using GUI ripping for automated testing of Android applications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, ACM, pp 258–261
- Android (2016) Developers - investigating your ram usage. URL <https://developer.android.com/studio/profile/investigate-ram.html#LogMessages>
- Android (2018a) App startup time. URL <https://developer.android.com/topic/performance/vitals/launch-time>
- Android (2018b) Developers - keeping your app responsive. URL <https://developer.android.com/training/articles/perf-anr.html#Reinforcing>
- Android (2019a) Distribution dashboard - android developers. URL <https://developer.android.com/about/dashboards>
- Android (2019b) UI/Application Exerciser Monkey. URL <https://developer.android.com/studio/test/monkey>
- Android Central (2015) The HTC One series specs. URL <https://www.androidcentral.com/htc-one-series-specs>
- Android Open-Source Project (2017) Android Open Source Project. URL <https://source.android.com/>
- Andrzejak A, Eichler F, Ghanavati M (2017) Detection of memory leaks in C/C++ code via machine learning. In: 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, pp 252–258
- Anscombe FJ (1948) The validity of comparative experiments. *Journal of the royal statistical society series A (General)* 111(3):181–211
- Araujo J, Alves V, Oliveira D, Dias P, Silva B, Maciel P (2013) An Investigative Approach to Software Aging in Android Applications. In: Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on, IEEE, pp 1229–1234
- Araujo J, Matos R, Alves V, Maciel P, Souza F, Trivedi KS, et al. (2014) Software Aging in the Eucalyptus Cloud Computing Infrastructure: Characterization and Rejuvenation. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 10(1):11

- Bovenzi A, Cotroneo D, Pietrantuono R, Russo S (2011) Workload Characterization for Software Aging Analysis. In: Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on, IEEE, pp 240–249
- Carrozza G, Cotroneo D, Natella R, Pecchia A, Russo S (2010) Memory leak analysis of mission-critical middleware. *Journal of Systems and Software* 83(9):1556–1567
- Cotroneo D, Grottke M, Natella R, Pietrantuono R, Trivedi KS (2013a) Fault triggers in open-source software: An experience report. In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on, IEEE, pp 178–187
- Cotroneo D, Orlando S, Pietrantuono R, Russo S (2013b) A measurement-based ageing analysis of the JVM. *Software Testing, Verification and Reliability* 23(3):199–239
- Cotroneo D, Natella R, Pietrantuono R, Russo S (2014) A Survey of Software Aging and Rejuvenation Studies. *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 10(1):8
- Cotroneo D, Fucci F, Iannillo AK, Natella R, Pietrantuono R (2016) Software Aging Analysis of the Android Mobile OS. In: Proc. 27th IEEE Intl. Symp. Software Reliability Engineering (ISSRE)
- Daniel WW (1990) Kruskal-wallis one-way analysis of variance by ranks. *Applied Nonparametric Statistics* pp 226–234
- Dohi T, Goseva-Popstojanova K, Trivedi KS (2000) Statistical non-parametric algorithms to estimate the optimal software rejuvenation schedule. In: Dependable Computing, 2000. Proceedings. 2000 Pacific Rim International Symposium on, IEEE, pp 77–84
- Durbin J, Watson GS (1951) Testing for serial correlation in least squares regression. ii. *Biometrika* 38(1/2):159–177
- Fang L, Dou L, Xu G (2015) Perfblower: Quickly detecting memory-related performance problems via amplification. In: 29th European Conference on Object-Oriented Programming (ECOOP 2015)
- Gallo R, Hongo P, Dahab R, Navarro LC, Kawakami H, Galvão K, Junqueira G, Ribeiro L (2015) Security and system architecture: Comparison of Android customizations. In: ACM WiSec, ACM, p 12
- Garg S, Moorsel AV, Vaidyanathan K, Trivedi KS (1998) A Methodology for Detection and Estimation of Software Aging. In: Proc. of the 9th Intl. Symp. on Software Reliability Engineering (ISSRE)
- Ghanavati M, Costa D, Seboek J, Lo D, Andrzejak A (2019) Memory and resource leak defects and their repairs in java projects. *Empirical Software Engineering* DOI 10.1007/s10664-019-09731-8, URL <https://doi.org/10.1007/s10664-019-09731-8>
- Grottke M, Trivedi K (2007) Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate. *IEEE Computer* 40(2):107–109
- Grottke M, Li L, Vaidyanathan K, Trivedi KS (2006) Analysis of Software Aging in a Web Server. *IEEE Trans Reliability* 55(3):480–491
- Grottke M, Nikora AP, Trivedi KS (2010) An empirical investigation of fault types in space mission system software. In: Dependable Systems and Net-

- works (DSN), 2010 IEEE/IFIP International Conference on, IEEE, pp 447–456
- Grottke M, Kim DS, Mansharamani R, Nambiar M, Natella R, Trivedi KS (2016) Recovery From Software Failures Caused by Mandelbugs. *IEEE Transactions on Reliability* 65(1):70–87
- GSMarena (2019) Counterpoint: Huawei reaches all-time high market share in Q1 2019. URL https://www.gsmarena.com/counterpoint_huawei_reaches_alltime_high_market_share_in_q1_2019-news-36845.php
- Hamed KH, Rao AR (1998) A modified mann-kendall trend test for autocorrelated data. *Journal of Hydrology* 204(1-4):182–196
- Huang Y, Kintala C, Kolettis N, Fulton ND (1995) Software Rejuvenation: Analysis, Module and Applications. In: *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on, IEEE*, pp 381–390
- Huo S, Zhao D, Liu X, Xiang J, Zhong Y, Yu H (2018) Using machine learning for software aging detection in Android system. In: *2018 Tenth International Conference on Advanced Computational Intelligence (ICACI), IEEE*, pp 741–746
- Iannillo AK, Natella R, Cotroneo D, Nita-Rotaru C (2017) Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations. In: *Proc. 28th IEEE Intl. Symp. Software Reliability Engineering (ISSRE)*
- Jump M, McKinley KS (2007) Cork: dynamic memory leak detection for garbage-collected languages. In: *Acm Sigplan Notices, ACM*, vol 42, pp 31–38
- Kang Y, Zhou Y, Xu H, Lyu MR (2015) PersisDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions. *arXiv preprint arXiv:151207950*
- Li L, Bissyandé TF, Wang H, Klein J (2018) CiD: Automating the detection of API-related compatibility issues in Android apps. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp 153–163
- Love R (2010) *Linux Kernel Development*. Pearson Education
- Machida F, Xiang J, Tadano K, Maeno Y (2012) Aging-related bugs in cloud computing software. In: *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on, IEEE*, pp 287–292
- Machida F, Andrzejak A, Matias R, Vicente E (2013) On the effectiveness of Mann-Kendall test for detection of software aging. In: *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*
- Machida F, Xiang J, Tadano K, Maeno Y (2016) Lifetime extension of software execution subject to aging. *IEEE Transactions on Reliability* 66(1):123–134
- Machiry A, Tahiliani R, Naik M (2013) Dynodroid: An input generation system for Android apps. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM*, pp 224–234
- Mao K, Harman M, Jia Y (2016) Sapienz: Multi-objective automated testing for Android applications. In: *Proceedings of the 25th International Sympo-*

- sium on Software Testing and Analysis, ACM, pp 94–105
- Montgomery DC (2008) Design and analysis of experiments. John Wiley & Sons
- Nagappan M, Zimmermann T, Bird C (2013) Diversity in software engineering research. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ACM, pp 466–476
- Nethercote N, Seward J (2007) Valgrind: A framework for heavyweight dynamic binary instrumentation. In: ACM Sigplan notices, ACM, vol 42, pp 89–100
- Önöz B, Bayazit M (2003) The power of statistical tests for trend detection. Turkish Journal of Engineering and Environmental Sciences 27(4):247–251
- Pfening A, Garg S, Puliafito A, Telek M, Trivedi KS (1996) Optimal software rejuvenation for tolerating soft failures. Performance Evaluation 27:491–506
- Pirie W (1988) Spearman rank correlation coefficient. Encyclopedia of statistical sciences
- Qiao Y, Zheng Z, Qin F (2016) An empirical study of software aging manifestations in Android. In: Software Reliability Engineering Workshops (IS-SREW), 2016 IEEE International Symposium on
- Qiao Y, Zheng Z, Fang Y (2018a) An empirical study on software aging indicators prediction in Android mobile. In: IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE, pp 271–277
- Qiao Y, Zheng Z, Fang Y, Qin F, Trivedi KS, Cai KY (2018b) Two-level rejuvenation for Android smartphones and its optimization. IEEE Transactions on Reliability 68(2):633–652
- Qin F, Zheng Z, Li X, Qiao Y, Trivedi KS (2017) An empirical investigation of fault triggers in Android operating system. In: 2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC), IEEE, pp 135–144
- Scalabrino S, Bavota G, Linares-Vásquez M, Lanza M, Oliveto R (2019) Data-driven solutions to detect API compatibility issues in Android: An empirical study. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), IEEE, pp 288–298
- Sen PK (1968) Estimates of the regression coefficient based on kendall’s tau. Journal of the American Statistical Association 63(324):1379–1389
- Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) AddressSanitizer: A fast address sanity checker. In: Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12), pp 309–318
- Silva L, Madeira H, Silva J (2006) Software Aging and Rejuvenation in a SOAP-based Server. In: Proc. of the 5th IEEE Intl. Symp. on Network Computing and Applications (NCA), pp 56–65
- Sor V, Treier T, Srirama SN (2013) Improving statistical approach for memory leak detection using machine learning. In: 2013 IEEE International Conference on Software Maintenance, IEEE, pp 544–547
- Tanenbaum AS, Bos H (2014) Modern Operating Systems, Prentice Hall Press
- Theil H (1992) A rank-invariant method of linear and polynomial regression analysis. In: Henri Theil’s Contributions to Economics and Econometrics,

- Springer, pp 345–381
- Vaidyanathan K, Trivedi KS (2005) A comprehensive model for software rejuvenation. *IEEE Transactions on Dependable and Secure Computing* 2(2):124–137
- Wang Q, Wolter K (2015) Reducing task completion time in mobile offloading systems through online adaptive local restart. In: *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*, ACM, pp 3–13
- Wei L, Liu Y, Cheung SC (2016) Taming Android fragmentation: Characterizing and detecting compatibility issues for Android apps. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp 226–237
- Wei L, Liu Y, Cheung SC, Huang H, Lu X, Liu X (2018) Understanding and detecting fragmentation-induced compatibility issues for Android apps. *IEEE Transactions on Software Engineering*
- Weng C, Xiang J, Xiong S, Zhao D, Yang C (2016) Analysis of Software Aging in Android. In: *Software Reliability Engineering Workshops (ISSREW), 2016 IEEE International Symposium on*
- Weng C, Zhao D, Lu L, Xiang J, Yang C, Li D (2017) A rejuvenation strategy in Android. In: *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, pp 273–279
- Wheeler D (2016) Sloccount. URL <http://www.dwheeler.com/sloccount/>
- Wikipedia (2019) Google Nexus. URL https://en.wikipedia.org/wiki/Google_Nexus
- Wu H, Wolter K (2015) Software aging in mobile devices: Partial computation offloading as a solution. In: *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*, IEEE, pp 125–131
- Wu L, Grace M, Zhou Y, Wu C, Jiang X (2013) The impact of vendor customizations on Android security. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, ACM, pp 623–634
- Xiang J, Weng C, Zhao D, Tian J, Xiong S, Li L, Andrzejakb A (2018) A new software rejuvenation model for Android. In: *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, IEEE, pp 293–299
- Xu G, Rountev A (2008) Precise memory leak detection for Java software using container profiling. In: *Proceedings of the 30th international conference on Software engineering*, ACM, pp 151–160
- Xu G, Rountev A (2010) Detecting inefficiently-used containers to avoid bloat. In: *ACM Sigplan Notices*, ACM, vol 45, pp 160–173
- Xu G, Rountev A (2013) Precise memory leak detection for Java software using container profiling. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22(3):17
- Xu G, Bond MD, Qin F, Rountev A (2011) LeakChaser: Helping programmers narrow down causes of memory leaks. In: *ACM SIGPLAN Notices*, ACM, vol 46, pp 270–282

Zheng P, Qi Y, Zhou Y, Chen P, Zhan J, Lyu MR (2014) An Automatic Framework for Detecting and Characterizing Performance Degradation of Software Systems. *Reliability, IEEE Transactions on* 63(4):927–943