Software Aging Analysis of the Android Mobile OS

Domenico Cotroneo, Francesco Fucci, Antonio Ken Iannillo, Roberto Natella, Roberto Pietrantuono
Università degli Studi di Napoli Federico II, Naples, Italy
{cotroneo, francesco.fucci, antonioken.iannillo, roberto.natella, roberto.pietrantuono}@unina.it

***Abstract*—Mobile devices are significantly complex, feature-rich, and heavily customized, thus they are prone to software reliability and performance issues. This paper considers the problem of software aging in Android mobile OS, which causes the device to gradually degrade in responsiveness, and to eventually fail. We present a methodology to identify factors (such as workloads and device configurations) and resource utilization metrics that are correlated with software aging. Moreover, we performed an empirical analysis of recent Android devices, finding that software aging actually affects them. The analysis pointed out processes and components of the Android OS affected by software aging, and metrics useful as indicators of software aging to schedule software rejuvenation actions.**

I. Introduction

Mobile devices (including smartphones, tablets and wearables) assist people in their personal activities, and are today a fundamental resource to communicate and to benefit from cloud services: mail, data storage, e-commerce, banking, and social networking are only few examples. In the near future, they will become digital wallets and keepers of digital identity. Moreover, mobile devices are used in business contexts to access to sensitive enterprise data and services.

As a result, users expect a reliable platform, which should be responsive and avoid smartphone crashes and data losses. Assuring the reliability of mobile devices is a challenge for smartphone vendors: devices have become significantly complex and feature-rich, are upgraded at a fast pace, and are heavily customized by vendors in order to differentiate their products from competitors.

The increase of software complexity inevitably leads to software bloat and reliability issues. The Android mobile OS, which currently dominates the smartphone market, grew up to more than 6 millions of lines of Java and C++ code[1]. Moreover, previous studies showed that software complexity and vendor customizations have a negative impact on Android reliability in terms of bug density and vulnerabilities [2]–[4]. In turn, this reflects in poor quality perceived by users, and affects the popularity of mobile products on the market.

This paper considers the problem of the **software aging** phenomenon in the Android mobile OS. Software aging can cause the device to slowly degrade its performance and to eventually fail, due to the accumulation of errors in the system state and to the incremental consumption of resources, such as physical memory. Software aging can be attributed to software bugs that manifest themselves as memory leakage and fragmentation, unreleased locks, stale threads, data corruption, and numerical error accumulation [5], [6]. We found evidence[2] that these bugs affect the Android OS, thus exposing commercial Android devices on the market to software aging issues.

Contributions. In this paper, we present an experimental methodology to analyze software aging issues in the Android mobile OS. The methodology uses statistical methods to identify which factors (such as workloads and device configurations) exacerbate performance degradation and resource consumption. Moreover, the methodology analyzes the correlation between software aging and resource utilization metrics, in order to pinpoint which subsystems are affected by aging and to support the design of software rejuvenation strategies. We used the methodology for an extensive empirical analysis of software aging in recent Android devices, pointing out that:

- Android devices are indeed affected by software aging. Our devices experienced a noticeable performance degradation (in terms of low responsiveness of the system) after few hours of stress testing. The magnitude of degradation is influenced by the workload (apps, events) and by the configuration (in particular, storage availability).
- Software aging issues can be attributed to specific processes inside the Android OS, including the *System Server*, the *System UI* and the *Surface Flinger*, which exhibit inflated memory consumption. Moreover, specific services inside them, such as the *Activity Manager* and the *Power Manager*, exhibit aging trends.
- Performance degradation trends are correlated with resource utilization metrics from the kernel, such as process PSS size and pages merged by KSM, and with the time spent on garbage collection. These metrics are useful as symptoms of software aging, and to schedule software rejuvenation actions in Android.

The paper is structured as follows. Section II discusses related work on software aging analysis and mobile reliability. Section III describes the proposed methodology for software aging analysis. Sections IV and V present the plan and the results of our empirical case study. Section VI closes the paper.

[1]LOCs computed with David A. Wheeler's SLOCCount [1] on the Android Open Source Project (AOSP), the baseline version of the Android OS, version 5.0, not including further LOCs from external open-source projects (such as the Linux kernel and SQLite) and from vendor customizations.

[2]At the time of writing (May 2016), we found 137 resource leak issues recorded by developers on the AOSP issue tracker [7], by querying for the word "*leak*" (https://goo.gl/5lY5u7) from a pool of 7585 total reports (https://goo.gl/VkLzXI), and represent 1.8% of all reports. This number is a lower bound for aging-related bugs, since aging issues are not limited to memory leaks, may have been reported using different keywords, or may have not been reported by users due to low repeatability. This result is consistent with other studies on bugs in open-source software [8]–[10].

II. Related Work

Software aging has been repeatedly reported both by scientific literature and by software practitioners [11], and it has been recognized as a chronic problem in many long-running software systems. In order to provide context to discuss our experimental methodology and analysis, we briefly review the most relevant results and techniques for the empirical analysis of software aging.

Garg et al. [12] presented an early study on software aging issues from systems in operation, by monitoring a network of UNIX workstations over a period of 53 days. This study adopted SNMP to collect data on resource consumption and OS activity, including memory, swap space, file, and process utilization metrics. The analysis found that the 33% of reported outages were related to resource exhaustion, and in particular to memory utilization (which exhibited the lowest *time-to-exhaustion* among the monitored resources).

Garg et al. [12], and later Grottke et al. [13], adopted statistical hypothesis testing and regression to identify *degradation trends* in resource consumption measurements (*i.e.,* if random fluctuations are excluded, the time series exhibits a gradual increase or decrease over time). The *Mann-Kendall test* and the *seasonal Kendall test* were adopted to confirm the presence of trends, respectively without and with periodic cycles, and the *Sen's procedure* and *autoregressive models* to forecast the time-to-exhaustion.

Silva et al. [14] and Matias and Filho [15] studied software aging in SOA and web server environments by performing stress tests. They showed that aging can lead to gradual performance degradation in terms of throughput, latency, and success rate of web-service requests. A similar effect was observed by Carrozza et al. [16] on a CORBA-based middleware, in which the performance degradation of remote object invocations was attributed to memory leak issues, reducing the performance of memory allocators and bloating internal data structures.

Subsequent studies found that software aging issues can also affect the lower layers of the software stack, such as the Sun's Java Virtual Machine [17], the Linux kernel [18], and cloud management software [19]. In particular, the study on the JVM revealed that performance degradation trends were exacerbated by the inefficiency of the garbage collector.

Some empirical studies focused on the analysis of bugs behind software aging issues (*aging-related bugs*), both in several open-source software projects for the LAMP stack [9], [10], [20] and cloud computing [8], and in embedded software used for space missions [21]. These studies provided insights on the nature of aging-related bugs: they represent a minor share of all software defects but are quite subtle to identify and to fix; most of them affect memory consumption and, in many cases, application-specific logical resources (such as thread pools and I/O connections).

Recent research has been focused on monitoring techniques to detect software aging in deployed systems, which is especially challenging due to varying workload conditions and configuration. They include machine learning techniques [22], such as *decision trees* and robust time series analysis techniques [23], [24], *e.g.,* the *Cox-Stuart test* and the *Hodrick-Prescott filter*.

Research on software aging in mobile devices is still at an early stage. Araujo et al. [25] designed a testbed for stress testing of Android applications.However, their approach was not meant to study aging issues inside the Android OS, and their tests did not point out any software aging symptom at the lower layers of the Android OS. Other studies were focused on preventing performance degradation of mobile applications through off-loading of tasks to the cloud and local application restarts [26], [27], debugging apps for performance bugs [28], and on forecasting Android device failures with time series analysis techniques [29]. Finally, studies on reliability of mobile devices were focused on field failure data analysis [2] and robustness testing [3] of mobile devices but did not consider software aging issues. Our work differs from these studies since it focuses on the empirical analysis software aging issues inside the Android OS and provides a systematic methodology to identify these issues.

III. Methodology for Software Aging Analysis

Our methodology leverages *stress tests* to identify software aging issues. In general, stress tests exercise a system with an intense workload, in terms of volume of user requests per unit of time, for a long period: as showed by previous studies [13]–[16], an intense workload significantly increases the likelihood to bring out software aging effects, for example by triggering memory leaks and bloat. Stress testing requires to find a balance: on the one hand, it needs to perform diversified tests under several conditions in order to explore as much aging-prone scenarios as possible; on the other hand, the number of stress tests should be kept at a minimum, since each test may take a significant amount of time until the onset of noticeable aging effects. For these reasons, we adopt the *Design of Experiments* (DoE) approach [30] to achieve a trade-off between thoroughness and duration of stress tests. DoE is aimed at creating a minimal set of test scenarios (namely, a test plan) able to explain most of the output variability with few treatments, by separating out the impact of variables of interest from the (usually negligible) impact of multiple variables together. This way, depending on how many treatments can be done, the tester can systematically decide to ignore the less important variables.

In the following, we define *factors* and *response variables* tailored for applying DoE in the context of a mobile OS. The response variables represent the outcome of an experiment, *i.e.,* metrics that quantify the aging effects. We consider both *user-perceived* response metrics (subsection III-A), which reflect software aging as directly perceived by the user, and *system-related* response metrics (subsection III-B), which reflect the amount of stress imposed on resources and subsystems, and indirectly the possible causes of user-perceived aging. Appendix A lists the metrics adopted in our analysis. The factors (subsection III-C) are the parameters of a test, which can potentially affect the response variable; their value

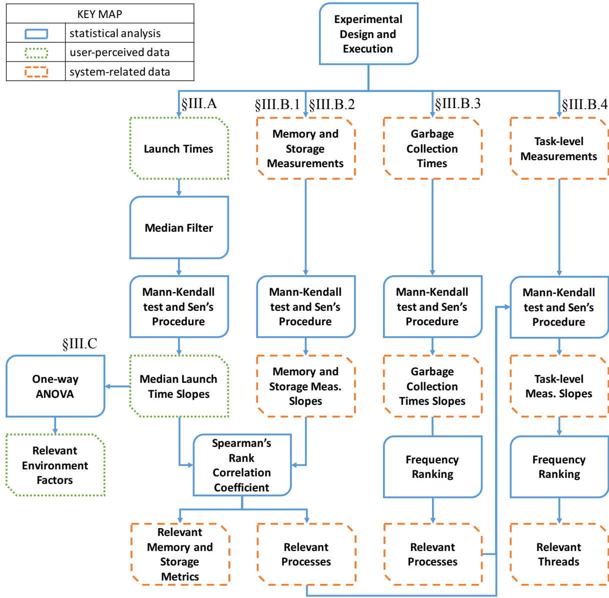

Fig. 1. Overview of the methodology for software aging analysis.

is chosen among a set of possible *levels*. Finally, experimental data are analyzed to identify software aging effects. Figure 1 provides an overview of the methodology for data analysis, which is further commented in the following subsections. This methodology is generic and not constrained to any specific Android device or vendor.

A. User-Perceived Response Variable

To evaluate the impact of software aging perceived by the user, we consider that one of the main design goals of Android is to achieve a good *responsiveness*. For example, one of the early requirements was to be able to cold-start a basic application, up to a responsive GUI, within 200 ms at most [31], [32]. For this reason, we measure user-perceived aging in terms of **Launch Time** (LT) of Android Activities (*i.e.*, an application component that generates and manages a GUI screen). The LT is the period between the request for an Activity, and the display of the contents of the Activity, including both background and foreground initialization.

This metric is computed from the logs generated by the Activity Manager, which is a service inside the Android OS that instantiates and switches among Android Activities, saving and restoring their state when necessary. These logs are generated the first time that an Activity is created: they are denoted by the "ActivityManager" tag and the keyword "Displayed", and can be collected using the *logcat* Android tool. To regularly collect LT samples through the whole duration of an experiment, we periodically terminate (at a low frequency, every 60 seconds) workload applications, and let them restart once they are exercised again by a workload generator. The periodical termination avoids the caching of Activities by the Android OS, which would prevent us from measuring the responsiveness at instantiating new Activities; moreover, periodical termination prevents the accumulation of errors (such as memory leaks) inside user applications, since our focus is on software aging inside the Android OS. The following line is a 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
```

The LT is analyzed to point out a possible degradation of device responsiveness. Ideally, if no software aging were present, the average LT should be stable thorough the whole experiment, given that the workload and test conditions are kept fixed during the experiment. Instead, in case of software aging, we expect that the device will experience performance degradation, manifesting as a gradual degradation of LT.

To analyze Launch Times (Figure 1), we compute the median LT among all activities during every 30-seconds period, and check the presence of a *trend* in this time series, by means of the non-parametric *Mann-Kendall* (MK) test [12]. It tests the null hypothesis that there is no monotonic trend in the time series and provides a level of significance (*p-value*) indicating the likelihood of the null hypothesis according to the samples. If the p-value is lower than α, we reject the null hypothesis with $1-\alpha$ confidence, and we conclude that an aging trend occurred. In our study, we require a confidence higher than 90% ($\alpha = 0.1$). If the LT series exhibits a trend, the slope is computed by a non-parametric procedure known as *Sen's procedure*, providing a coefficient denoting the rate at which the data values increase following a linear model.

B. System-Related Response Variables

To examine in depth software aging issues, we also collect metrics on resource utilization and on events occurring inside the Android OS. System-related metrics include basic resources, such as memory and storage, and fundamental system operations, such as garbage collection and task management. We relate these metrics to the Launch Time, in order to point out the areas of the Android system that can be affected by aging. Moreover, these metrics can provide a valuable basis for developing software rejuvenation solutions since they can be monitored easily, with low intrusiveness and in a portable way across Android devices, by using standard interfaces provided by the Android framework and the Linux kernel. Thus, we investigate whether software aging effects can be detected by looking for deviations and trends in these metrics.

*1) **Memory**:* The physical memory is a critical resource for achieving good performance, but several studies showed that this resource is often affected by software aging and exhibits the lowest time-to-exhaustion [10], [12]–[16]. The efficiency of memory usage in Android can be influenced by several factors. The Android OS provides sophisticated mechanisms for memory management, both at the kernel-level and at the Android framework-level, including automated app lifecycle management, application process caching, and memory reclaim through the out-of-memory killer [31]. Moreover, the Android OS provides many complex services,

such as *Activity Manager* and *Package Manager*, that may be inefficient and cause software aging in the long term (*e.g.*, due to poor management of complex data structures, which may grow indefinitely over time). Therefore, we include metrics for analyzing memory usage in our experiments (Table VI in the appendix).

The *proc* file system of the Linux kernel is the primary source of information about memory usage. In particular, the virtual file */proc/meminfo* provides information on memory consumption. Furthermore, the Android utility *dumpsys* provides additional information about memory usage of the Android framework, such as the amount of cached Android apps and of Android kernel extensions, such as the Kernel Samepage Merging (*KSM*) and virtual memory compression (*zram*). Finally, memory consumption is also tracked for each individual process of the Android OS, by measuring the *Proportional Set Size* (*PSS*), which is the amount of physical RAM consumed by the process, where each page is weighted by the amount of processes that share that page (for example: if a 4kB page is shared by two process, each process is only accounted 2kB for that page in the PSS metric). All these metrics are periodically sampled every 30 seconds.

As showed in Figure 1, we check whether LT degradation is related to memory usage, by looking for trends in memory usage metrics and by checking whether these trends are correlated to LT degradation trends. For each memory usage metric, we perform the following two steps: (i) we test the presence of a trend (and compute its slope) in the time series of the metric (MK and 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*, 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 accompanied by a degradation trend of the LT in the same experiment. For example, given a memory usage metric, say *Total Free*, we compute correlation between the pairs of trends observed for the *Total Free* metric and for the median LT in each experiment. We conduct this analysis both on global memory consumption metrics and on individual per-process PSS metrics. This latter analysis also points out processes that are more critical from the aging point of view.

*2) **Storage**:* Storage management involves many components both at kernel-level and at Android framework level, such as the filesystem, the block I/O management, drivers, and storage frameworks such as *SQLite*. As a matter of fact, studies on filesystems showed that performance may degrade over time due to software aging problems [18], [33]. In turn, a delay on storage access is perceived by the users as poor responsiveness. Storage metrics analyzed in our experiments are listed in Table VII in the appendix.

The *proc* file system of the Linux kernel is the primary source of information about storage usage. In particular, the virtual file */proc/diskstats* provides information for each storage partition in terms of throughput and latency of I/O transfers. Many performance monitoring tools, such as *iostat*, gather information from this virtual file. Storage metrics are periodically collected every 30 seconds. Storage metrics are analyzed in the same way as memory usage metrics (Figure 1), by computing trends using the Mann-Kendall test and the Sen's procedure, and by evaluating the correlation between the trends of the metric and the trends of LT across all experiments, using the Spearman's correlation coefficient.

*3) **Garbage Collection**:* Garbage collection (GC) is a fundamental activity of the *Android Runtime* (*ART*), which is the execution environment of Android applications. The garbage collector relieves programmers from heap memory management, which is a cumbersome and error-prone task, by automatically reclaiming unused memory. However, GC can still be a cause of performance degradation in the case of memory fragmentation under intense workloads and of poor object lifetime management by programmers [6], [16], [17]. Thus, many GC algorithms are supported by the ART, and their efficiency influences the perceived performance of Android devices: if GC takes too long, the application may experience slowdowns and may even be temporarily paused during GC. However, it is tricky to assure the efficiency of GC under every possible scenario, and the ART may be affected by memory fragmentation in the long term. For these reasons, we analyze metrics on the performance of GC (Table VIII in the appendix).

The main source of information on GC are the Android logs 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
```

A log line records a GC occurrence, along with other information[34] The ART produces a GC log when the *GC Pause Time* exceeds 5 ms, or the *GC Duration* exceeds 100 ms. These cases of slow garbage collections are especially important for our analysis: we are interested in collections that are performed when the system is under memory pressure (which is the case of our aging stress tests) and that may result in performance degradation and resource exhaustion. We collect GC log records continuously during the experiments.

Similarly to memory and storage metrics, the GC metrics are analyzed for each individual process, by computing trends using the Mann-Kendall test and the Sen's procedure (Figure 1). We count the number of cases in which the process exhibited a increase of GC occurrences, which reveals a possible relationship between software aging (in particular, loss of responsiveness) and memory bloat or fragmentation.

*4) **Tasks**:* In Linux, a *task* is the basic scheduling unit of the kernel. If a set of tasks shares OS resources (such as the virtual address space), they form a multi-threaded process. Multi-threading is extensively used in the Android OS to run several Android services, where each service manages a specific hardware resource (*e.g.,* sensors, audio, and camera) or serves a specific Android framework API. In order to get fine-grained information about the effects of software aging,

we analyze CPU and memory utilization metrics for individual tasks (Table IX in the appendix). These metrics can point out the tasks that caused most stress on the Android system during the experiments, and that may have triggered aging issues and should be considered for software rejuvenation.

The main source of task-level metrics is the *proc* filesystem. In particular, for each task, the kernel exposes the virtual files */proc/TASK_PID/schedstat* and */proc/TASK_PID/stat* to provide information on scheduling and memory usage of that task. Task-level metrics include minor and major page fault counts, and execution time spent in user-space and kernel-space (which denote higher CPU and I/O activity). Task-level metrics are periodically collected every 30 seconds.

To identify critical tasks (as in Figure 1), we compute trends for each metric and for each task using the 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 90%. 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.

C. Factors and Levels

To extensively stress the Android OS and to trigger latent software aging issues, we run tests under several different configurations and workload applications, which represent the *factors* of the experimental plan. We consider 5 factors, and derive the test plan by varying the combinations of *levels* of these factors according to DoE.

1) ***Application Set (APP):*** Experiments use different sets of applications as workload to stimulate the system, aiming to reflect common usage scenarios. These sets should include popular Android applications, that need to be installed on all Android devices used in the experiments.

2) ***Device (DEV):*** Experiments may run on different Android physical devices, with its own hardware and software configuration, and its own Android version. Every available Android device in the experimental setup establishes a level for the DEV factor.

3) ***Workload Launch&Kill Frequency (L&K):*** During experiments, applications are terminated and re-launched continuously with frequency based on two levels: *HIGH*, every 5 seconds leading the system to a high stress; and *LOW*, every 60 seconds just to collect Launch Time samples (see subsection III-A).

4) ***Workload Events Configuration (EVENTS):*** A set of "events" is used by the workload generator (the *monkey* tool) to interact with the Android device. These include: application switch, touch, motion, trackball, and navigation events. EVENTS varies on three levels: *SWITCHES*, where the experiments allows only application switch events; *MIXED*, where the application allows all events (uniformly distributed); *NONE*, where the application does not allow any of these events (in this case, the L&K factor is forced to *HIGH*).

5) ***Storage Space Usage (STO):*** Experiments may run with or without storage availability (in terms of free space), which may impact on the performance of the storage subsystem. It varies on two levels: *FULL*, where 90% of the storage is occupied by filling it with multimedia files (such as 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).

We check whether the 5 factors contribute to the severity of LT degradation, in order to provide context about which conditions lead to Android OS aging. We apply the one-way *Analysis of Variance* (*ANOVA*) technique to assess which factors impact the response variable in a statistically-significant way. In order to use a non-parametric ANOVA and be robust to potential non-normal distribution of errors, we use the *Kruskal-Wallis/Wilcoxon hypothesis test*. The null hypothesis, in this case, is that the factor does not impact the response variable. We conclude that a factor impacts the response variable if the level of confidence is higher than 90%, *i.e.,* the p-value is less than 0.1.

IV. Experimental Plan

We applied the general methodology from Section III to conduct an empirical case study on software aging phenomena in Android devices, focusing on the Android OS version 5.0. We conducted experiments on two Android smartphones, respectively a *high-end* device and a *low-end* device from Huawei, in the context of a joint R&D project with this Android vendor. The devices belong to the same product line, and differ with respect to the hardware equipment; the most notable differences are with respect to the type and amount of CPUs, RAM, and storage. Thus, we have two levels for the DEV factor labeled as *HIGH-END* and *LOW-END*. Devices are equipped both with stock applications provided by the vendor, and with third-party applications. We organized applications in three sets, which represent the three levels of the APP factor. They are: *STOCK-1*, including mainly stock applications by Android and Google; *STOCK-2*, including mainly stock applications by Huawei; and *3PARTY*, including popular third-party applications downloaded from the Google Play app store. Factors and levels are summarized in Table I, with a full list of the application packages used for the APP levels. 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), and user inputs are provided with the *monkey* tool.

The duration of a single experiment is given by the number of events injected with the *monkey* tool. We opted for 150,000 events with a throttle (*i.e.,* time between events) of 500 ms, thus obtaining an expected duration of about 20 hours, in accordance with previous studies in which 20 hours of stress testing were sufficient to make software aging to surface [16], [35]. In most cases, our experiments lasted even less than 20 hours since the high-stressing load and the subsequent aging phenomena lead to system failures and poor responsiveness.

We defined an experimental plan (*i.e,* a set of experiments) by considering different combination of levels of the factors

TABLE I
FACTORS AND LEVELS

FACTOR	LEVEL	DESCRIPTION
DEV	HIGH-END	Android 5 high-end device
	LOW-END	Android 5 low-end device
APP	STOCK-1	com.google.android.videos, com.huawei.camera, com.android.browser, com.huawei.phoneservice, com.android.email, com.android.contacts, com.google.android.apps.maps, com.google.android.marvin.talkback, com.android.chrome, com.google.android.play.games, com.android.calendar, com.google.android.music, com.google.android.youtube
	STOCK-2	com.android.systemui, com.huawei.powergenie, com.android.browser, com.huawei.vassistant, com.android.contacts, com.android.gallery3d, com.huawei.appmarket, com.huawei.gamebox, com.android.phone, com.android.settings, com.huawei.android.totemweather
	3PARTY	com.tencent.mm, com.sina.weibo, com.qiyi.video, com.youku.phone, com.taobao.taobao, com.tencent.mobileqq, com.baidu.searchbox, com.baidu.BaiduMap, com.UCMobile, com.moji.mjweather
L&K	HIGH	kill and re-launch applications every 5 seconds
	LOW	kill and re-launch applications every 60 seconds
EVENTS	SWITCHES	monkey tool allows only for switch events
	MIXED	monkey tool allows for switch, touch, motion, trackball, and navigation events
	NONE	monkey tool is not used (this level forces L&K to HIGH)
STO	FULL	90% of storage space usage
	NORMAL	default storage space usage

presented in subsection III-C. A full factorial design, using every possible combination at all levels of all factors, is typically impractical because of many experiments (72 in our case, *i.e.,* approximately 60 days of experiments) with little gain in terms of explanation of the response variability[3].

A two-stage fractional factorial plan is adopted. Specifically, we first considered the minimum number of experiments to analyze only the *main effects* of each factor on the launch time without accounting for the experimental error (*i.e.*, a "saturated" design). The plan created is a customized design

TABLE II
EXPERIMENTAL PLAN OF THE CASE STUDY

ID	DEV	APP	L&K	EVENTS	STO
EXP1	HIGH-END	STOCK-1	HIGH	NONE	NORMAL
EXP2	LOW-END	STOCK-2	HIGH	SWITCHES	FULL
EXP3	HIGH-END	STOCK-1	LOW	SWITCHES	NORMAL
EXP4	LOW-END	STOCK-2	HIGH	SWITCHES	NORMAL
EXP5	LOW-END	STOCK-2	LOW	MIXED	FULL
EXP6	LOW-END	3PARTY	HIGH	NONE	FULL
EXP7	HIGH-END	3PARTY	LOW	MIXED	NORMAL
EXP8	HIGH-END	3PARTY	LOW	MIXED	FULL
EXP9	HIGH-END	STOCK-1	LOW	MIXED	NORMAL
EXP10	HIGH-END	STOCK-1	HIGH	MIXED	NORMAL
EXP11	HIGH-END	STOCK-1	HIGH	MIXED	NORMAL
EXP12	HIGH-END	STOCK-1	HIGH	MIXED	NORMAL
EXP13	HIGH-END	STOCK-1	HIGH	NONE	NORMAL
EXP14	LOW-END	STOCK-2	HIGH	MIXED	NORMAL
EXP15	LOW-END	STOCK-2	HIGH	MIXED	FULL
EXP16	LOW-END	STOCK-2	HIGH	SWITCHES	FULL
EXP17	LOW-END	STOCK-2	LOW	SWITCHES	FULL
EXP18	LOW-END	3PARTY	HIGH	SWITCHES	FULL
EXP19	LOW-END	3PARTY	LOW	SWITCHES	FULL
EXP20	LOW-END	3PARTY	LOW	MIXED	FULL
EXP21	LOW-END	3PARTY	HIGH	MIXED	FULL
EXP22	LOW-END	3PARTY	LOW	MIXED	NORMAL
EXP23	LOW-END	3PARTY	HIGH	MIXED	NORMAL
EXP24	LOW-END	3PARTY	HIGH	NONE	NORMAL
EXP25	HIGH-END	STOCK-1	LOW	SWITCHES	NORMAL
EXP26	HIGH-END	3PARTY	HIGH	MIXED	NORMAL
EXP27	HIGH-END	3PARTY	HIGH	MIXED	FULL
EXP28	HIGH-END	3PARTY	HIGH	MIXED	FULL
EXP29	LOW-END	3PARTY	HIGH	MIXED	FULL

of 8 experiments. We performed an initial ANOVA on these experiments, and identified factors and levels with the highest influence on software aging trends. In the second stage, based on the feedback of the first stage, we added experiments with the aim of accounting also for the experimental error (having at least two observations for each level), and biasing the design in favor of the high-impact factors, through an unbalanced design. The final plan of 29 experiments is in Table II. In the end, unbalancing allowed us highlighting more severe aging trends, while keeping the ability of comparing different factors and two-way interactions with a confidence of 95%. As a matter of fact, the ANOVA test to verify the model confidence, computed after the experiments execution, yielded a p–value of 0.0225, *i.e.,* a confidence of $(1 - 0.0225)\% = 97.75\%$.

V. EXPERIMENTAL RESULTS

After executing the experimental plan of Section IV, we analyzed software aging phenomena using the metrics and the techniques presented in Section III.

A. Analysis of Launch Time

The Launch Time of Android activities provides a direct indicator of software aging as experienced by the user. The analysis showed that, in all the experiments that we executed, there is a statistically-significant positive trend in the LT series. On average, the LT trend across all the experiments has been 9.15E-03 ms/s (with an estimated degradation of 659ms, on average, of the launch time after 20 hours of testing), with a maximum of 6.39E-02 ms/s in the worst case (estimated degradation of 4.6 seconds after 20 hours). Indeed, at the end

[3]Full designs allow assessing the impact of the main factors and of contemporary variation (*i.e.,* of the interaction) of all factors together on the response (*i.e.*, in our case, of two-, three-, four-, and five-way interactions), with a very high degree of redundancy: usually, just the main factors and, possibly, some two-way interactions contribute to explain the response variability, while three-way (and higher order) interactions are negligible [30].

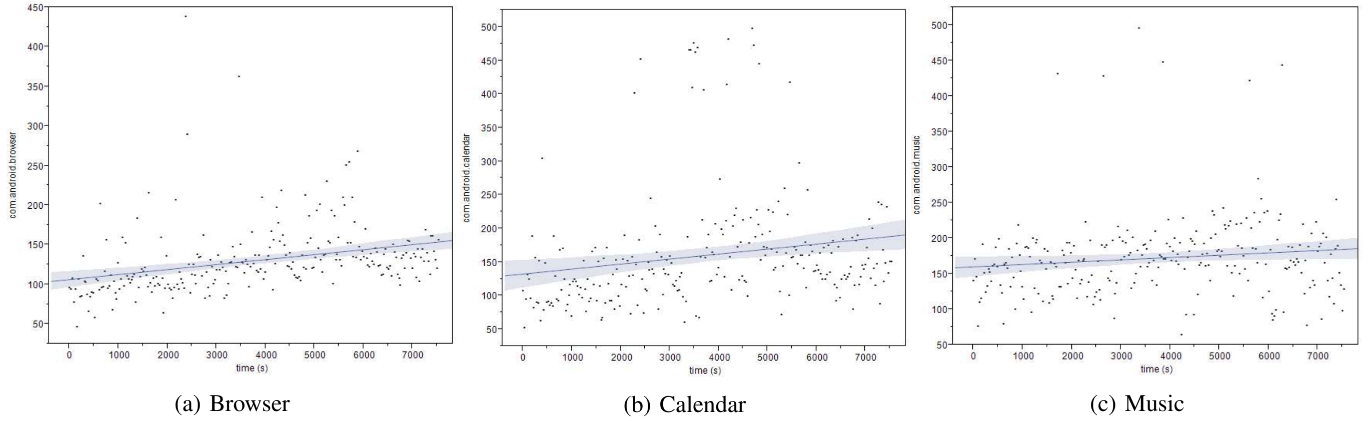

Fig. 2. Examples of Launch Time trends in EXP10.

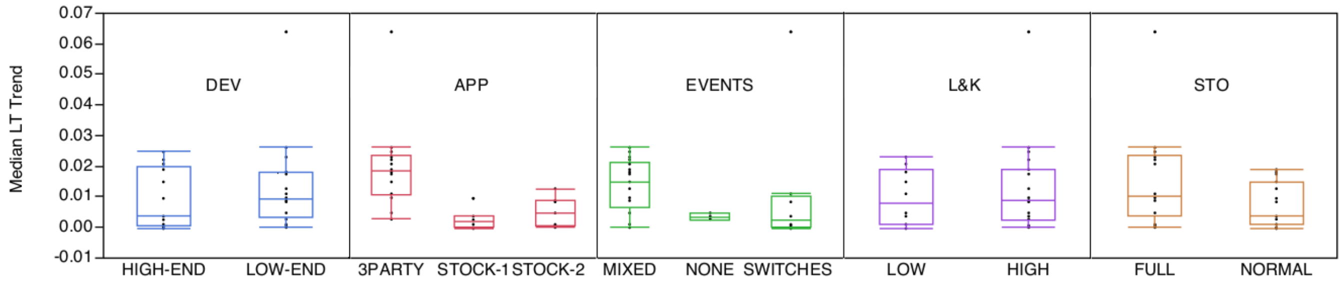

Fig. 3. Median Launch Time trends distribution for each factor.

of the experiments, we found the device in an unusable state, in which the switch between apps required seconds to provide feedback. Figure 2 provides examples of LT trends from EXP10 in basic applications, namely the Android Browser, Calendar, and Music applications.

We performed the one-way ANOVA after the first stage of experiments, to assess whether the differences between the samples is statistically significant, and thus the factors that impact the observed *Launch Time* trend. According to ANOVA, the only factor that determines statistically-significant differences in the Launch Time is **APP**, with a confidence of 99%, where the experiments with *3PARTY* level yielded trends higher than the experiments under the other levels. This result suggests that software aging in Android depends on the workload, which can stress different services and subsystems of the Android OS depending on user applications.

Figure 3 shows the *distribution of Launch Time trends* from both the first and the second stage of experiments, by dividing the samples with respect to different levels of each factor of the experimental plan. The distributions do now show relevant differences of *Launch Time* trends for the **DEV** and **L&K** factors. While the **EVENTS** and **STO** factors do not have a statistically-significant impact, their distributions show a weak influence of these factors on the launch time.

B. Analysis of Memory Usage

While launch time gives an indication of software aging directly perceived by users, memory usage represents a potential underlying cause of this issue, since it often suffers from leaks, fragmentation, and thrashing [6]. Thus, our approach looks for a correlation between memory usage metrics and the LT, both to provide more information on aging, and to suggest metrics to use for detection of software aging at run-time and to support rejuvenation actions.

The correlation analysis of **global memory metrics** computes the Spearman's rank correlation between the series of memory usage trends (at system level) across experiments, and the series of trends for the median LT across experiments. The resulting correlation coefficients for each metric are shown in Table III. Positive or negative correlation means that high LT trends occur along with high trends of the metric. Considering a significance level of 90%, only the *Free Cached PSS* and the *KSM Saved* indicators are correlated to LT trends in a statistically-significant way, with moderate correlation degrees slightly greater than 0.4 in absolute value. These metrics are indicator of software aging, as process app caching and same-page merging often exhibit a trend when the device is aging. This behavior can provide indication at run-time that the responsiveness of the device is degrading. However, this correlation does not allow to accurately identify the root cause or to estimate the magnitude of the LT trend, since these metrics are coarse-grained and they are an indirect effect of the aging behavior of the device.

Therefore, to obtain more insights, we analyzed individual processes. In particular, the correlation analysis of **per-process memory usage** focuses on the *PSS* metric collected for each

TABLE III
SPEARMAN CORRELATION COEFFICIENTS BETWEEN LAUNCH TIME TRENDS AND GLOBAL MEMORY USAGE TRENDS.

MEMORY INDICATOR	SPEARMAN COEFFICIENT	P-VALUE
Free Cached PSS	-0.4084	0.0278
KSM Saved	0.4389	0.0890
Total Used	-0.2820	0.1383
Total Free	0.2344	0.2210
Lost Ram	0.2138	0.2653
KSM Volatile	0.2726	0.3070
ZRAM In SWAP	0.1874	0.3302
Used PSS	-0.1712	0.3747
Used Buffers	-0.1686	0.3819
Free Cached	0.1247	0.5192
ZRAM Physical Used	0.1082	0.5764
KSM Unshared	0.1357	0.6163
KSM Shared	0.0649	0.8113
Used Shared Memory	-0.0111	0.9543
Used Slab	0.0002	0.9990

process of the Android OS. Since most of the Android processes (about 60 out of 80) have a very small memory footprint that can be considered negligible, the analysis focused on processes whose PSS exceeded 20 MB during an experiment.

The results point out that three processes have a statistically-significant, high correlation (in absolute value) with the LT trend: they are the *System Server*, *System UI*, and *Surface Flinger*. Figure 4 provides an example, which shows the trend of the PSS of the *System Server* process in EXP10. These processes play an important role in the Android OS:

- The *System Server* is the first Java process that starts at Android OS boot, which initializes the whole Android Framework. It hosts 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. This process also regulates the access to system resources.
- The *System UI* is the process that composes the screen areas to display notifications, communication of device status, and device navigation buttons using system bars.
- The *Surface Flinger* process receives window layers (surfaces) from multiple sources (*System UI* included), combines them, and displays them on the screen.

The correlation measures for the PSS of these processes are reported in Table IV. Experiments showed that these processes experience a significant increase of the PSS, respectively up to +60%, +486% and +222% compared to the initial PSS, reaching 114MB, 409MB, and 296MB. Thus, these processes are further analyzed in the following of this section.

C. Analysis of Garbage Collection

We further analyze Android processes from the point of view of memory management, by considering the time spent for garbage collections, namely the *GC Pause Time* and *GC Duration*. We performed a trend analysis on these GC metrics for each process. The results were grouped by different

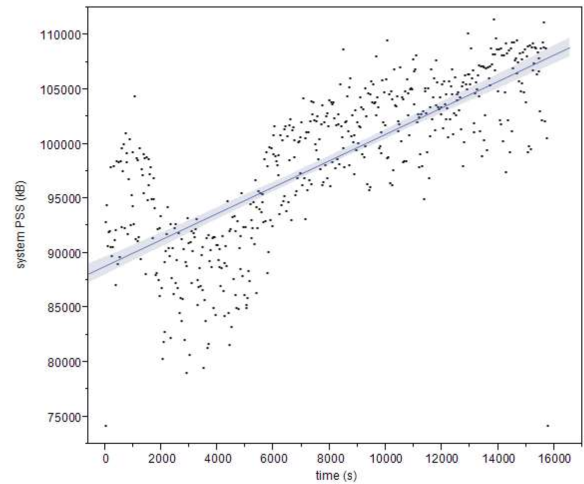

Fig. 4. System Server PSS trend in EXP10.

TABLE IV
SPEARMAN CORRELATION COEFFICIENTS BETWEEN LAUNCH TIME TRENDS AND PSS TRENDS OF ANDROID SYSTEM PROCESSES.

PROCESS	SPEARMAN COEFFICIENT	P-VALUE
system (System Server)	0.5323	0.003
surfaceflinger (Surface Flinger)	0.4828	0.008
com.android.systemui (System UI)	-0.3967	0.0331

collection types [34]: in particular, in our experiments only two GC types produced more than 100 samples and exhibited a trend with confidence higher than 90%, namely:

- *Concurrent* GC, in which threads are not suspended and not prevented from making more allocations, but a separate thread performs GC concurrently in background;
- *Explicit* GC, where a thread makes an explicit request for GC and it is blocked during this operation.

To investigate how much these processes are affected by a degeneration of GC activity, we analyzed how often a trend occurs in the GC times of each process. Then, we rank the processes according to the number of experiments in which the process exhibited a trend. Figure 5 reports these ranks, focusing on the topmost 5 processes.

This analysis highlights that the *System Server*, the *com.huawei.systemmanager*, and the *System UI* are the processes that most frequently showed a GC times trend. In particular, the System Server experienced trends in almost all experiments. We can also notice that trends in *Explicit* collections are much more frequent than *Concurrent* collections. This can be explained by observing that system processes in the Android OS often explicitly invoke GC to take advantage of the opportunities to reduce the memory footprint. In the Android AOSP version 5, we found respectively 54 and 26 explicit invocations in the *framework* and in the *libcore* subtrees of the source code.

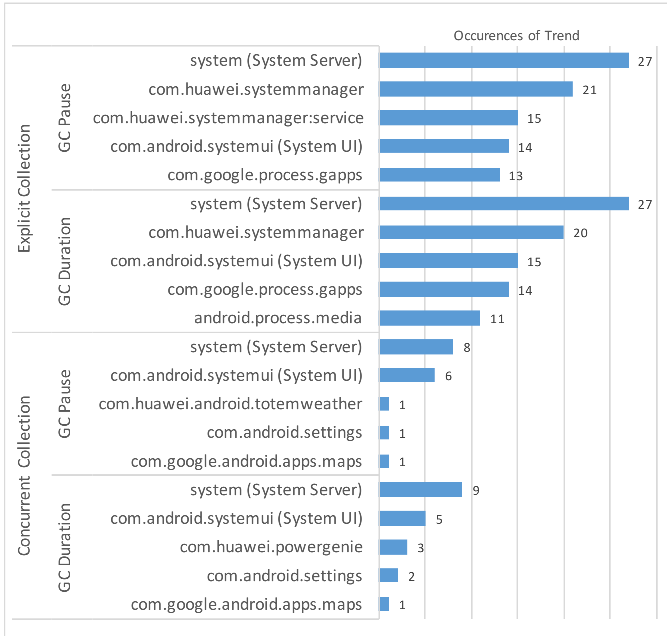

Fig. 5. Android processes that most often exhibited a statistically-significant trend of the time spent for garbage collection (*GC Pause Time* and *GC Duration*, for both *Explicit* and *Concurrent* collection).

These results suggest that these processes heavily use heap memory, and that they are exposed to performance degradation due to the inflation and fragmentation of the heap, increasing the overhead of garbage collections and slowing down or suspending the threads. Moreover, monitoring the GC times of these processes is another useful indicator to detect performance degradation.

D. Analysis of Storage

The metrics about storage usage provide information on the volume and duration of read/write operations, and can provide hints on degradation of the I/O throughput in the long term. Moreover, these metrics provide another opportunity for run-time monitoring and detection of software aging in Android. Similarly to the analysis of memory metrics, we analyze the correlation between storage metrics and the LT trends.

Table V shows the Spearman's rank correlation coefficient between the trends of each metric, and the trends of the median LT. However, none of the metrics exhibit a statistically-significant correlation (*i.e.*, *p-value* is always greater than 0.1). Thus, the storage usage does not seem to be related to the performance degradation caused by software aging.

This result can be explained observing that storage I/O activity by both the applications and by the Android framework is small and sporadic. Thus, software aging trends in the storage stack are more rare and smaller than trends in memory usage, and would become noticeable only after a longer period of time, or under a different workload that stresses the storage.

It is worth noting that in the analysis of factors with ANOVA (subsection V-A) we found that storage utilization has a weak influence on user-perceived aging. This can be

TABLE V
SPEARMAN CORRELATION COEFFICIENTS BETWEEN STORAGE INDICATORS TRENDS AND MEDIAN LT TRENDS

STORAGE INDICATOR	SPEARMAN COEFFICIENT	P-VALUE
Reads Completed	-3.13E-02	0.8768
Reads Merged	-2.30E-01	0.2447
Sectors Read	9.49E-03	0.9626
Reading Time	-7.58E-02	0.7072
Writes Completed	1.30E-01	0.5030
Writes Merged	-7.09E-02	0.7247
Sectors Written	-6.80E-02	0.7356
Writing Time	4.97E-02	0.8053
Read Completion Time	9.72E-02	0.6291
Write Completion Time	9.74E-02	0.6290
I/O Time	1.18E-01	0.5500
Weighted I/O Time	-8.39E-03	0.9669

explained considering that the Android framework scans the storage for indexing media files. The involved process, namely *android.process.media*, is busy when the amount of data to scan is high, which has an influence on the performance degradation. The higher activity of the *android.process.media* is confirmed by the trend analysis of GC times (subsection V-C, Figure 5): this process exhibited a statistically-significant trend in 11 out of 14 cases in which the STO factor was set to HIGH.

E. Analysis of Tasks

In the previous subsections, we identified three relevant Android processes that showed effects of software aging, *i.e., System Server*, *System UI*, and *Surface Flinger*. We performed a more detailed, task-level analysis by separately considering threads of these three processes, with the goal of identifying aging-related components inside them, which are potential candidates for monitoring and rejuvenation actions.

The analysis of task-related metrics showed us that some specific tasks present statistically-significant trends across the experiments (see [31] for more information on the internal Android tasks). They are:

- *Binder_** tasks: the thread pool used for managing inter-component communication on the *Android Binder*.
- *GCDaemon* and *Heap_thread_pool** tasks: the threads in charge of executing the garbage collection by the Android Runtime.
- *ActivityManager*: a thread of the System Server for executing the *Activity Manager* service, which handles requests for managing the lifecycle of Android activities.
- *PowerManager*: a thread of the System Server for executing the *Power Manager* service, which exposes API for managing the power state of the device (*e.g.,* to wake-up the device from power saving).

Figure 6 shows the first 5 tasks for each of the three processes, ordered by the number of experiments (showed on top of the bars) in which the task experienced a trend in at least one task-level metric. In particular, the tasks inside the System Server experienced trends in the highest number of cases. These data point out that the CPU and virtual memory

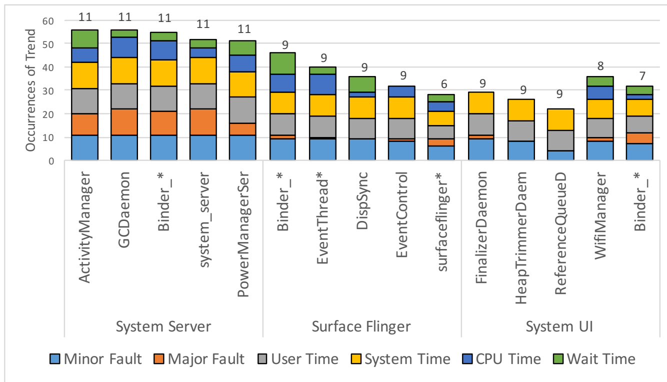

Fig. 6. First 5 tasks of *System Server*, of *Surface Flinger*, and of *System UI* processes with the highest number of significant trends.

demand (in terms of CPU ticks and minor/major page faults) of these threads increases over time as an effect of software aging. In turn, this suggests to focus software rejuvenation actions on the Android services that are most stressed and exhibit behavioral deviations, such as the Activity Manager and the Power Manager.

VI. Key Findings and Conclusion

In this paper, we proposed an experimental methodology to analyze software aging issues in the Android OS. We applied this methodology on recent Android devices, which provided the following findings about software aging.

The experiments actually triggered software aging effects in the Android devices, which manifested as an **increasing trend of the launch time of Android apps**. Therefore, it would be useful for Android vendors to adopt software rejuvenation to prevent this performance degradation, since it is perceived by the users as an indicator of poor responsiveness and low overall quality of the device.

To better understand this behavior, we considered several types of workload applications in our stress tests, both stock and third-party ones. We found that third-party applications accelerate the software aging trend, that is, the performance degradation is quicker compared to stock applications. Moreover, the type of application events (e.g., navigation, switch) also influences software aging trends. These results imply that **the occurrence of software aging, and its magnitude, depends on the type of workload running on the Android device**. Therefore, the software rejuvenation solution should be adaptive with respect to the workload. When the workload is more stressful, the software rejuvenation of the device should be anticipated.

To relate software aging with resource usage, we first analyzed memory and storage utilization metrics, both globally and for individual processes. We observed that among the global metrics, the *Free Cached PSS*, which denotes the memory footprint of processes recently executed on the Android device, showed consistent trends across the experiments. By looking at per-process memory consumption, we found that **specific processes of the Android OS are affected by a significant increase of memory consumption over time, correlated with the degradation of the launch time** (*i.e.*, both the launch time and the memory consumption degrade at the same time). The processes most affected by this behavior are: *System Server*, *Surface Flinger*, and *System UI*. Monitoring the memory consumption of these processes can support the detection of the onset of software aging in the device, and guide the triggering of software rejuvenation.

We further analyzed garbage collection and task-level metrics, to obtain fine-grained information about software aging effects. We found that **the garbage collection time is correlated with the degradation of the launch time** (*i.e.*, the time spent by the Android Runtime to reclaim unused heap memory from Android processes): the higher the garbage collection time, the higher the degradation of the launch time. Moreover, the task-level analysis pointed out that **specific components in the Android OS** (such the ActivityManager) often exhibit a trend in terms of CPU and virtual memory usage, which denotes **behavioral deviations** of these components in the presence of software aging. Thus, garbage collection and task-level metrics from selected components can provide further support for the detection of software aging.

According to the experimental results, we recommend that **Android software rejuvenation should adopt a measurement-based approach** to adapt to the workload conditions, which have a strong influence on software aging trends. The software rejuvenation solution should be triggered when measurements exhibit suspicious trends, including both resource usage and garbage collection activity. Moreover, the analysis identified processes and components that represent potential candidates for software rejuvenation actions.

As for the generality of the findings, it is important to note that software aging issues affected Android processes that belong to the Android AOSP codebase, which is mostly shared among different devices and vendors. As discussed in previous work on Android vulnerabilities [4], vendor customizations focus on device drivers and apps, while only moderate changes are introduced to the basic AOSP processes. Thus, it is likely that similar aging issues affect devices of several vendors. Of course, it is also possible that vendor customizations may introduce more aging issues. Our analysis of Garbage Collection times pointed out the *com.huawei.** processes, although their memory consumption is less inflated than the mentioned AOSP processes.

We believe that benchmarking the software aging of different Android products is an important research area. We have defined a general methodology that could be applicable to several Android devices, and in the future we plan to extend our analysis, and to include software rejuvenation solutions.

Acknowledgment

This work is part of a project sponsored by Huawei Technologies Co., Ltd. This work has been partially supported by the CECRIS EU FP7 project (grant no. 324334).

APPENDIX A
COLLECTED METRICS

Tables VI, VII, VIII, IX list all the metrics collected during the experiments as response variables.

TABLE VI
MEMORY METRICS

METRIC	UNIT	DESCRIPTION
Total Free	kB	Amount of physical RAM left unused
Free Cached	kB	Amount of physical RAM used as cache memory
Free Cached PSS	kB	Amount of physical RAM used by the PSS of cached processes (*i.e.* not in use)
Total Used	kB	Amount of physical RAM in use
Used PSS	kB	Amount of physical RAM used by the PSS of non cached processes
Used Buffers	kB	Amount of physical RAM used for file buffers
Used Shared Memory	kB	Amount of physical RAM used by shared memory
Used Slab	kB	Amount of physical RAM used by the kernel to cache data structures for its own use
Lost RAM	kB	Amount of physical RAM not accounted as free or as used
ZRAM Physically Used	kB	Amount of physical RAM used by compressed in memory swap
ZRAM In Swap	kB	Amount of uncompressed memory that has been transferred to ZRAM
KSM Saved	kB	Amount of physical RAM pages saved by memory de-duplication
KSM Shared	kB	Amount of physical RAM pages shared by memory de-duplication
KSM Unshared	kB	Amount of physical RAM pages not shared by memory de-duplication
KSM Volatile	kB	Amount of physical RAM pages changing too quickly to be merged

TABLE VII
STORAGE METRICS

METRIC	UNIT	DESCRIPTION
Reads Completed	#reads	Number of I/O read operations that have been successfully completed
Reads Merged	#reads	Number of I/O read operations on adjacent sectors, which have been merged
Sectors Read	#sectors	Number of sectors read successfully
Reading Time	ms	Total amount of time spent by I/O read operations
Writes Completed	#writes	Number of I/O write operations that have been successfully completed
Writes Merged	#writes	Number of I/O write operations on adjacent sectors, which have been merged
Sectors Written	#sectors	Number of sectors written successfully
Writing Time	ms	Total amount of time spent by I/O write operations
Read Completion Time	ms/read	Average amount of time spent by an I/O read operation
Write Completion Time	ms/write	Average amount of time spent by an I/O write operation
I/O Time	ms	Amount of time spent with at least one I/O operation in progress
Weighted I/O Time	ms	Number of I/O operations in progress, weighted by the amount of time spent doing I/O

TABLE VIII
GARBAGE COLLECTION METRICS

METRIC	UNIT	DESCRIPTION
GC Reason	#occur-rences per reason	Number of times each type of event (reason) triggers garbage collection [34]
GC Pause Time	ms	Amount of time for which the process has been in a pause state during the garbage collection
GC Duration	ms	Amount of time to complete the garbage collection

TABLE IX
TASK-LEVEL METRICS

METRIC	UNIT	DESCRIPTION
Minor Page Fault	#page faults	Number of page faults served by sharing pages already in memory
Major Page Fault	#page faults	Number of page faults that require an I/O operation
User Time	jiffies	Amount of time spent by the process in user mode
System Time	jiffies	Amount of time spent by the process in kernel mode
CPU Time	jiffies	Amount of time spent executing on the CPU
Wait Time	jiffies	Amount of time spent waiting for the CPU

REFERENCES

[1] D. Wheeler, “Sloccount,” May 2016. [Online]. Available: http://www.dwheeler.com/sloccount/

[2] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, “Characterizing Failures in Mobile OSes: A Case Study with Android and Symbian,” in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, 2010, pp. 249–258.

[3] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeyer, “An Empirical Study of the Robustness of Inter-Component Communication in Android,” in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, 2012, pp. 1–12.

[4] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, “The impact of vendor customizations on Android security,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 623–634.

[5] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, “Software Rejuvenation: Analysis, Module and Applications,” in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, 1995, pp. 381–390.

[6] M. Grottke, R. Matias, and K. S. Trivedi, “The fundamentals of software aging,” in *IEEE Proceedings of Workshop on Software Aging and Rejuvenation, in conjunction with ISSRE. Seattle, WA*, 2008.

[7] Android Open-Source Project, “Issue tracker,” May 2016. [Online]. Available: https://code.google.com/p/android/

[8] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, “Aging-related bugs in cloud computing software,” in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012, pp. 287–292.

[9] D. Cotroneo, R. Natella, and R. Pietrantuono, “Predicting aging-related bugs using software complexity metrics,” *Performance Evaluation*, vol. 70, no. 3, pp. 163–178, 2013.

[10] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, “Fault triggers in open-source software: An experience report,” in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International*

Symposium on, 2013, pp. 178–187.

[11] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A Survey of Software Aging and Rejuvenation Studies," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, p. 8, 2014.

[12] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi, "A Methodology for Detection and Estimation of Software Aging," in *Proc. of the 9th Intl. Symp. on Software Reliability Engineering (ISSRE)*, 1998.

[13] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of Software Aging in a Web Server," *IEEE Trans. Reliability*, vol. 55, no. 3, pp. 480–491, 2006.

[14] L. Silva, H. Madeira, and J. Silva, "Software Aging and Rejuvenation in a SOAP-based Server," in *Proc. of the 5th IEEE Intl. Symp. on Network Computing and Applications (NCA)*, 2006, pp. 56–65.

[15] R. Matias Jr and P. Freitas, "An Experimental Study on Software Aging and Rejuvenation in Web Servers," in *Proc. of the 30th Intl. Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2006, pp. 189–196.

[16] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory leak analysis of mission-critical middleware," *Journal of Systems and Software*, vol. 83, no. 9, pp. 1556–1567, 2010.

[17] D. Cotroneo, S. Orlando, R. Pietrantuono, and S. Russo, "A measurement-based ageing analysis of the JVM," *Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 199–239, 2013.

[18] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software Aging Analysis of the Linux Operating System," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, 2010, pp. 71–80.

[19] J. Araujo, R. Matos, V. Alves, P. Maciel, F. Souza, K. S. Trivedi *et al.*, "Software Aging in the Eucalyptus Cloud Computing Infrastructure: Characterization and Rejuvenation," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, p. 11, 2014.

[20] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, and K. S. Trivedi, "Recovery From Software Failures Caused by Mandelbugs," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 70–87, 2016.

[21] M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault types in space mission system software," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, pp. 447–456.

[22] J. Alonso, J. Torres, J. L. Berral, and R. Gavalda, "Adaptive On-Line Software Aging Prediction based on Machine Learning," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, 2010, pp. 507–516.

[23] P. Zheng, Y. Qi, Y. Zhou, P. Chen, J. Zhan, and M. R. Lyu, "An Automatic Framework for Detecting and Characterizing Performance Degradation of Software Systems," *Reliability, IEEE Transactions on*, vol. 63, no. 4, pp. 927–943, 2014.

[24] R. Matias, A. Andrzejak, F. Machida, D. Elias, and K. Trivedi, "A systematic differential analysis for fast and robust detection of software aging," in *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*, 2014, pp. 311–320.

[25] J. Araujo, V. Alves, D. Oliveira, P. Dias, B. Silva, and P. Maciel, "An Investigative Approach to Software Aging in Android Applications," in *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, 2013, pp. 1229–1234.

[26] H. Wu and K. Wolter, "Software aging in mobile devices: Partial computation offloading as a solution," in *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*, 2015, pp. 125–131.

[27] Q. Wang and K. Wolter, "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*, 2015, pp. 3–13.

[28] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "PersisDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions," *arXiv preprint arXiv:1512.07950*, 2015.

[29] S. Marcek and M. Drozda, "Predicting system failures on mobile devices," in *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*. Springer, 2016, pp. 499–508.

[30] D. C. Montgomery, *Design and analysis of experiments*. John Wiley & Sons, 2008.

[31] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Prentice Hall Press, 2014, ch. 10.8.

[32] Android, "Developers - keeping your app responsive," May 2016. [Online]. Available: https://developer.android.com/training/articles/perf-anr.html#Reinforcing

[33] R. Matos, J. Araujo, V. Alves, and P. Maciel, "Characterization of software aging effects in elastic storage mechanisms for private clouds," in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*, 2012, pp. 293–298.

[34] Android, "Developers - investigating your ram usage," May 2016. [Online]. Available: https://developer.android.com/studio/profile/investigate-ram.html

[35] A. Bovenzi, D. Cotroneo, R. Pietrantuono, and S. Russo, "Workload Characterization for Software Aging Analysis," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, 2011, pp. 240–249.