UNIVERSIDADE D
COIMBRA

Pedro Guilherme da Cruz Ferreira

THE IMPACT OF SOFTWARE AGING ON THE POWER CONSUMPTION OF MOBILE DEVICES

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor João Ferreira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September of 2023

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Pedro Guilherme da Cruz Ferreira

The impact of software aging on the power consumption of mobile devices

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. João Ferreira and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

September 2023

Acknowledgements

I would like to express my gratitude to my friends for the unwavering support and memorable moments they shared with me throughout this journey.

I extend my heartfelt appreciation to my advisor, Prof. João Ferreira, and the entire Greenstamp project team for their invaluable assistance.

Last but certainly not least, I wish to thank my family, whose unwavering support has been a constant throughout my life.

Funding

This work was financed by FEDER (Fundo Europeu de Desenvolvimento Regional), from the European Union through CENTRO 2020 (Programa Operacional Regional do Centro), under project CENTRO-01-0247-FEDER-047256 - GreenStamp: Mobile Energy Services.

Abstract

Mobile devices are used globally in people's daily lives, and people spend, on average more than 3 hours a day on their smartphones, 90% of which are on mobile applications. In this context, the battery life of mobile devices is an increasingly important factor in consumer satisfaction, and the energy efficiency of mobile applications is something they take into account. Software aging is a cumulative process known as the increasing degradation of the internal state of the software during its operational cycle. This degradation is marked by mismanagement of the available resources of the system. Mobile applications are equally affected by this phenomenon, and it is expected that energy consumption can be affected, and thus smartphones' battery lives are affected as a consequence. Previous research has explored the realms of software aging and the energy consumption of mobile devices and applications. However, a distinctive aspect of our study is its pioneering attempt to establish a connection between these two areas. Drawing from methodologies and tools employed by prior researchers in the analysis of software aging and battery consumption, we developed our own approach aimed at uncovering a potential correlation between them. We conducted experiments using an accelerated workload, simulating random user events on a selection of 20 Android applications across four diverse categories. As part of our data collection process, we gathered system performance indicators to monitor the software aging process, in addition to collecting data on battery consumption. Ultimately, owing to the subjective nature of the aging phenomenon, we addressed the aging variable using distinct variants, each contingent on the number of aging conditions to which the system was exposed. Then, a statistical analysis was conducted to seek a correlation between this variable and battery consumption. Additionally, we explored whether there existed a noteworthy disparity in this correlation among the various application categories, each characterized by distinct energy and resource consumption profiles. Our analysis indicates that in cases of stronger aging states, characterized by the presence of multiple aging conditions in the system, only approximately half of the experiments exhibited an increase in battery consumption when transitioning from non-aged states to aged states. The primary category contributing to this increase was identified as the social media category. Ultimately, our analysis led to the conclusion that, based on our dataset, no significant correlation could be extrapolated, suggesting that software aging does not significantly impact the battery consumption of mobile devices. Instances where an increase in battery consumption was observed were attributed to factors other than the aging status of the system.

Keywords

Software Aging; Battery Consumption; Mobile Devices; Android

Resumo

Os dispositivos móveis são utilizados mundialmente no dia-a-dia das pessoas, que passam em média mais de 3 horas por dia nos smartphones, 90% dos quais são passados em aplicações móveis. Neste contexto, a vida útil da bateria dos dispositivos móveis é cada vez mais um fator importante na satisfação dos utilizadores e a eficiência energética de aplicações móveis é tida em conta. O envelhecimento de software é um processo cumulativo que se traduz na degradação progressiva do estado interno do software durante o seu ciclo de funcionamento. Essa degradação é caracterizada pela gestão inadequada dos recursos disponíveis no sistema. As aplicações móveis também são afetadas por esse fenómeno, e é expectável que o consumo de energia seja afetado, tendo como consequência um impacto na vida útil das baterias dos smartphones. Os tópico do envelhecimento de software e do consumo de energia de dispositivos e aplicações móveis já tinha sido anteriormente investigado. No entanto, o nosso estudo tenta ser pioneiro na medida em que pretende estabelecer uma ligação entre estas duas áreas. Partindo das metodologias e ferramentas utilizadas por outros investigadores na análise do envelhecimento de software e do consumo de bateria, desenvolvemos a nossa própria abordagem com o objetivo de identificar uma possível correlação entre ambos. Realizámos experiências utilizando uma carga de trabalho acelerada, que simula eventos aleatórios dos utilizadores em 20 aplicações Android, distribuídas por quatro categorias distintas. O processo de recolha de dados consistiu em monitorizar indicadores de desempenho do sistema para que fosse possível acompanhar o seu processo de envelhecimento, além de recolher dados sobre o consumo de bateria. Dado o caráter subjetivo do fenómeno de envelhecimento, abordámos a variável de envelhecimento através de diferentes variantes, consoante o número de condições de envelhecimento a que o sistema estava submetido em cada ocasião. Posteriormente, realizámos uma análise estatística com o intuito de procurar uma correlação entre esta variável e o consumo de bateria. Adicionalmente, investigámos se existia uma diferença significativa nesta correlação entre as diversas categorias de aplicações, cada uma delas caracterizada por perfis distintos de consumo energético de e recursos. A nossa análise indica que, em casos de estados de envelhecimento mais acentuados, caracterizados pela presença de múltiplas condições de envelhecimento no sistema, apenas cerca de metade das experiências registou um aumento no consumo de bateria ao transitar de estados não envelhecidos para estados envelhecidos. A categoria principal associada a esse aumento foi a categoria de redes sociais. Por fim, com base no nosso conjunto de dado e respetiva análise, concluímos que não foi possível identificar uma correlação significativa, sugerindo que o envelhecimento de software não tem um impacto significativo no consumo de bateria de dispositivos móveis. Os casos em que se observou um aumento no consumo de bateria foram atribuídos a fatores que não são o envelhecimento do sistema.

Palavras-Chave

Envelhecimento de software; Consumo de bateria; Dispositivos móveis; Android

Contents

Acronyms

ADaRTA Aging Detection and Rejuvenation Tool for Android.

ADB Android Debug Bridge.

APM Advance Power Management.

AR aging-related.

ARB aging-related bug.

ARF aging-related failure.

ART Android Runtime.

BMU Battery Monitoring Unit.

FSM Finite State Machine.

GC Garbage Collection.

HAL Hardware Abstraction Layer.

KPI Key Performance Indicators.

kswapd Kernel swap daemon.

LMK Low-memory killer.

mA milliamps.

mmapping Memory-mapping.

OS Operating System.

PSS Proportional Set Size.

RAM Random Access Memory.

RPC Remote Procedure Call.

RSS Resident Set Size.

SA Software aging.

SDK Software Development Kit.

SNMP Simple Network Management Protocol.

TTAF time to aging-related failure.

USS Unique Set Size.

List of Figures

List of Tables

Chapter 1

Introduction

Mobile devices are used globally in people's daily lives, and it is estimated that people spend, on average, more than 3 hours a day [29] on their smartphones, with an increasing trend for each year that goes by [42]. It is also expected that around 90% of the user's screen time is spent on mobile applications [33]. In this context, the battery life of mobile devices is an increasingly important factor in consumer satisfaction. There are mobile applications with millions of downloads with the purpose of optimizing the battery consumption of devices, which shows that users are interested in maximizing the battery duration of their mobile devices. Thus, mobile applications must use energy capacity as efficiently as possible.

Software aging (SA) is known as an increasing degradation of the internal state of the software during its operational cycle. This is a cumulative process, just like aging in humans, and the cumulative effects of successive occurrences of errors directly influence the manifestation of aging-related failures. The system mismanagement of available resources, such as improper locking of files, memory leaking, and unfinished threads, typically leads to this error accumulation. These errors can be challenging to detect in the system development phase and expensive to remove.

It is suspected that mobile applications that are affected by this phenomenon, due to this bad management of resources, can also see an increase in the consumption of energy and thus affecting the device battery life.

The main goal of this project is to investigate the effects of software aging on mobile devices (specifically smartphones) and the impacts on energy consumption. At the end of this investigation, we should be able to answer the following research question:

RQ *What is the impact of software aging on the energy consumption of smartphones?*

In our pursuit of understanding the relationship between software aging and its impact on battery consumption in the context of Android applications, we have established an experimental process. For this purpose, we submitted Android applications to heavy workloads representative of user actions. We collected metrics

related to the device's memory usage and management mechanisms to detect aging patterns. In parallel, battery consumption was also measured. After this, we searched for a correlation between aging and battery consumption using statistical tests such as Spearman's Correlation Coefficient and Wilcoxon Rank Sum.

This work was developed in the context of the GreenStamp project, which aims to investigate and develop innovative mechanisms for analyzing and cataloging the energy efficiency of mobile applications integrated into app store processes.

1.1 Document structure

Until now, an introduction and contextualization of the problem were presented, as well as the goals of this dissertation. This section intends to provide an overview of the structure of this document and give some insight into what will be present in each chapter.

Chapter 2 provides a review of relevant theoretical background on various topics closely related to this research, as well as a discussion of prior research in this field.

In **Chapter 3**, we present the structured methodology developed and implemented throughout this research, along with a detailed explanation of the experimental procedures employed.

Chapter 4 is dedicated to presenting the findings and conducting an in-depth analysis.

Chapter 5 looks at the planning done and challenges that arose throughout this investigation.

Finally, **Chapter 6** serves as the concluding chapter of this dissertation, where we offer remarks and reflections on the work conducted.

Chapter 2

Literature Review

This chapter presents an overview of all the topics necessary to understand this document and reviews previous work related to the research conducted.

This section discusses the theoretical background that underpins this research. This will include a review of the key concepts and frameworks that support this study. The key theories most relevant to the topic will be highlighted, and a discussion will take place about how they have been used to understand and explain the phenomena being investigated.

2.1 Software faults classification

A service can be described as a sequence of the system's external states [15]. With this notion, it is important to know that a fault, error, and failure are not the same and can be distinguished as the following:

- **Fault**: The cause of an error. Incorrect code or bugs are prevalent examples of faults. A fault is active when it produces an error; otherwise, it is dormant [16].
- **Error**: Part of the total state of the system that may lead to its subsequent service failure [15].
- **Failure**: A deviation of the service delivered by the system from its specification. This deviation can be in the form of a slowed or limited service (partial failure), incorrect service, or no service at all [28].

The "chain of threats" (Figure 2.1) can represent the relationship between faults, errors, and failures. The actions that lead to each be explained by the following sequence [15]:

1. A fault gets activated by applying input (the activation pattern) to a component that causes a dormant fault to become active, generating an error.

2. An error can propagate within a given component and successively be transformed into other errors. This is called internal propagation and is caused by the computation process. An external propagation happens when an error reaches the service interface of one component and consequently propagates into another component that receives service from the first one.

3. A service failure occurs when an error is propagated to the service interface and causes the service delivered by the system to deviate from its correct service. If this service serves others, this failure can cause a permanent or transient external fault in them, and so on.

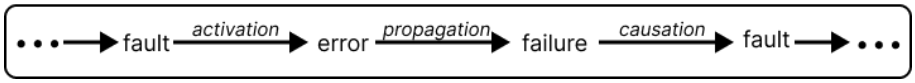

Figure 2.1: The fundamental chain of threats to dependability [16]

Avizienis et al. [15] presented a scheme for classifying faults according to eight criteria: Phase of occurrence, System boundaries, Phenomenological cause, Dimension, Objective, Intent, Capability, and Persistence.

According to this classification, bugs or faults in software code can be defined as *Internal human-made non-malicious permanent software development faults*.

Furthermore, the author extends this classification according to the activation reproducibility of faults:

- **Solid/Hard**: Faults whose activation and error propagation is reproducible.
- **Elusive/Soft**: faults whose activation or error propagation is not systematically reproducible.

Grottke and Trivedi [27] proposed a new fault classification for software faults, extending previous concepts given by Gray [26]:

- **Bohrbug**: "A fault that is easily isolated and that manifests consistently under a well-defined set of conditions because its activation and error propagation lack complexity as set out in the definition of Mandelbug. Complementary antonym of Mandelbug."
- **Mandelbug**: "A fault whose activation or error propagation is complex, where "complexity" can take two forms:
  1. The activation or error propagation depends on interactions between conditions occurring inside the application and conditions that accrue within the system's internal environment of the application.
  2. There is a time lag between fault activation and failure occurrence, e.g., because several different error states have to be traversed in the error propagation.

> Typically, a Mandelbug is difficult to isolate, or the failures caused by it are not systematically reproducible. Complementary antonym of Bohrbug."

An **Heisenbug** is a sub-type of Mandelbug and is defined as a fault that stops causing a failure or manifests differently when one attempts to probe or isolate it [27].

An **aging-related bug (ARB)** is also a sub-type of mandelbug and consists of a fault that leads to the accumulation of errors either inside the running application or in its system-internal environment, resulting in an increased failure rate or degraded performance [27]. This type of bug is further classified into sub-types by other authors, which can be seen in the following Section 2.4.

The following Figure 2.2 shows the representation of these four types of faults in the form of a Venn diagram.

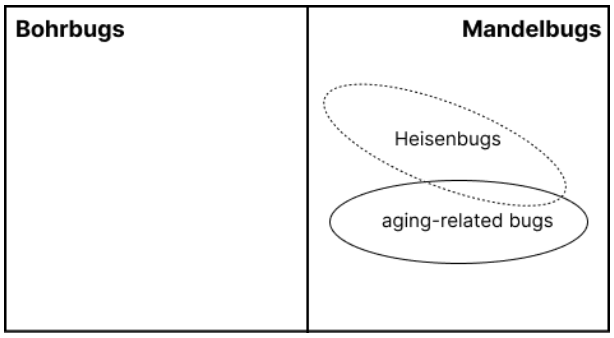

Figure 2.2: Venn diagram of software fault types [27]

2.2 Android Operating System (OS)

This research is focused on studying software aging in the Android platform due to its open nature and being the most used mobile OS these days, with a market share of around 72% [1]. With that said, it is essential to understand the basis of its architecture and some of the mechanisms built into it.

2.2.1 Android OS Architecture

Android is an open-source operating system developed by the huge multinational technology company Google, based on the Linux kernel. Because of its open nature, developers can easily modify and add enhanced features to meet the latest mobile technology requirements.

The major components that compose the architecture of the Android platform are the following [2]:

- **Linux Kernel**: As the base of the Android stack, the kernel provides essential services such as memory management, process management, power

management, networking, and device drivers. Using a Linux kernel allows Android to take advantage of key security features and enables device manufacturers to develop hardware drivers for a well-known kernel.

- **Hardware Abstraction Layer (HAL)**: Provides standard interfaces that expose device hardware capabilities to the higher-level Java API framework. This layer consists of multiple library modules that implement each interface for specific hardware components (camera, Bluetooth, etc.). When the Java API framework requests to access device hardware, the Android system loads the specific library module for that component.

- **Android Runtime (ART)**: ART is the runtime environment in which Android applications are executed. ART evolved into providing optimal performance to Android applications regarding compilation, garbage collecting, and debug support. Since version 5.0, each app runs in its own process and with its instance of the ART, which is written to run multiple virtual machines on low-memory devices.

- **Native Libraries**: Consists of a set of native libraries, written in C and C++, that provide the foundation for the Android operating system. These libraries support graphics, data storage, media, and more.

- **Java API Framework**: The application framework is a set of Java classes that provide the interfaces and tools needed to build Android applications. These classes support various features such as data storage, display, connectivity, and more.

- **System Apps**: The Android applications are at the top of the stack. Anyone can develop these using the Android application framework and the Android Software Development Kit (SDK).

2.2.2 Android Memory Management

Android has three types of memory:

- **RAM** is the fastest type of memory but is usually limited in size.

- **zRAM** is a partition of RAM used for swap space that grows or shrinks in size as pages are moved into or taken out of it. Everything is compressed when placed into zRAM and then decompressed when copied out of zRAM.

- **Storage** contains all the persistent data, such as the file system and the included object code for all apps and libraries. Its capacity is much larger than the other types, and unlike Linux, it is not used for swap space in Android.

.

RAM is broken into **pages** that are considered *free* (unused RAM) or *used* (RAM that the system is actively using). These pages fall into two categories:

- **Cached**, which is memory backed by a file in storage (e.g., code, memory-mapped files). Cached memory can be further classified as *private* if it is owned by one process and not shared or *shared* if used by multiple processes.
- **Anonymous**, which is memory not backed by a file in storage.

The ART keeps track of each memory allocation. Once it determines that the program is no longer using a piece of memory, it frees it up without any intervention from the programmer, a mechanism known as **Garbage Collection (GC)**. GC has two main goals: finding data objects in a program that cannot be accessed in the future and reclaiming the resources used by those objects.

Although, as said previously, Android does not use storage as swap space (in the sense of not paging out unused pages into storage to free up memory), the ART still uses Memory-mapping (mmapping) and paging to manage memory, meaning that storage can still be used to store and retrieve data from secondary storage, but in situations where applications forcefully terminated have a chance to save its state into storage [8].

Any memory an app modifies, whether by allocating new objects or touching mapped pages, remains resident in RAM and cannot be paged out. The only way to release memory from an app is to release object references that the app holds, making the memory available to the garbage collector. That is with one exception: any files mmapped in without modification, such as code, can be paged out of RAM if the system wants to use that memory elsewhere [11]. The only way to release memory from an app is to release object references that the app holds, making the memory available to the garbage collector.

Android has two main mechanisms to deal with **low memory situations**:

- **Kernel swap daemon (kswapd)** is part of the Linux kernel and converts used memory into free memory when the free memory of a device reaches a certain low threshold.
- **Low-memory killer (LMK)** is a mechanism performed by the kernel where processes are killed to free up memory in cases where kswapd is not enough. The processes are killed based on their priority, where background apps are the first to be killed, and system processes are the last.

The memory footprint of an application can be determined by the following metrics:

- Resident Set Size (RSS) is the number of shared and non-shared pages used by the app.
- Proportional Set Size (PSS) is the number of non-shared pages used by the app and an even distribution of the shared pages (e.g., if three processes share a 3MB page, each process has a 1MB PSS).
- Unique Set Size (USS) is the number of non-shared pages used by the app.

2.2.3 Mobile Power Management

A power supply is required to operate hardware components in a computer ecosystem. For that, a request is usually made by software: for instance, a process that requires the CPU or needs to read from memory or a notification that needs to make a sound. Power management policies are necessary to orchestrate how different hardware units use a single energy source.

In Android, at the kernel level, is the Power Management component which has been designed based on the standard Linux Advance Power Management (APM) technology and has the premise that the CPU should not consume energy if there is not at least one application or service demanding into applications or services require power [12].

Android applications, when in need of power resources, must request using software parameters called "wake locks", which the application sets to request or keep computing resources. When there are no active wake locks, Android shuts down the CPU and other battery-hungry peripherals to save energy when there is no reason to keep such components active.

Suppose an application needs to gain control of the power state. In that case, Android OS exposes it through the PowerManagement Java class that permits the creation of a manager object that manipulates the power state by setting the wake locks values. This class interfaces with lower levels of the OS stack to control the power state and can shut down the system when the unit is out of battery. Using this power management policy, Android allows the kernel to avoid races: wake locks are activated whenever wake-up events occur, which prevents the kernel from suspending the system and also provides a direct standard for the kernel to decide when opportunistic suspension should be started, this is, the kernel should attempt to suspend the system only whenever there is no active wake-locks [23].

To determine battery consumption, services with that purpose do not track battery current draw directly but instead collect timing information (e.g., resource requests) that can be used to approximate battery consumption by the different components. The power consumption is specified in milliamps (mA) of current drawn at a nominal voltage [9].

2.3 Measuring battery consumption on mobile devices

Compared to profiling the runtime of applications running on conventional computers, the profiling energy consumption of applications running on smartphones faces a unique challenge, asynchronous power behavior, where the effect on a component's power state due to a program entity lasts beyond the end of that program entity [36]. There are three main approaches to measuring the battery consumption of mobile devices: hardware-based, model-based, and software-based [17].

Hardware-based

This approach quantifies the device's energy consumption by physically measuring the electric current using a digital multimeter or other instruments with similar functions. The power consumption is calculated with the product of the current times the voltage. In the case of a smartphone, this requires connecting the measurement instrument in series between a terminal of the battery and the corresponding contact on the phone.

Monsoon power monitor [4] and ODroid [5] are two existing tools that fall in this category. Although this type of measurement provides the most accurate readings [17], its feasibility of application at a large scale can be challenged. A considerable monetary and setup-time cost is necessary for the reproducibility and parallelization of studies that consider several devices is limited. It may also imply disassembling the device to bypass the battery, incurring the risk of damaging the device or voiding the manufacturer's warranty [40]. A big disadvantage of this method is that it cannot profile the energy consumption at the application or method scope.

Model-based

Model-based approaches involve using mathematical models to estimate the energy consumption of an app or system. These models can be based on various factors, such as the hardware and software configurations of the device, the workload being run, and the power consumption characteristics of the hardware. Model-based approaches can help predict the energy consumption of a system before it is built or deployed but may not be as accurate as hardware-based measurements.

Software-based

Software-based approaches to measuring the energy consumption of mobile applications use data collected from the different software system interfaces to perform calculations to assess the energy consumption at the software application level. At this scope, it is possible to implement profiling techniques instrumenting the source code, which permits a detailed analysis of the routines and system calls that significantly contribute to energy dissipation [17, 36].

Android applications typically access hardware components in two different ways. When an application uses hardware components supported by the Linux kernel, the application requests related system calls. Otherwise, the application makes a Remote Procedure Call (RPC) via the Android binder. **Kernel activity monitoring** is one technique commonly used with this approach, where these requests are monitored and used to perform the calculations needed to assess the energy consumption of the software [44].

There are some disadvantages to using this approach:

- Not as accurate as the hardware-based approach

- Tools that require running in the background will consume some battery as well and cause an overhead in measurements.

Some tools have been developed to analyze and profile the energy consumption of mobile applications:

- **PowerTutor** [6] is an Android application that displays the power consumed by major system components such as CPU, network interface, display, GPS receiver, and different applications. It uses a power consumption model built by direct measurements during careful control of device power management states. This model generally provides power consumption estimates within 5% of actual values. It also provides a text-file-based output containing detailed results and can be used to monitor the power consumption of any application.

- **Android BatteryStats** [3] provides battery status and hardware usage information and is widely used for battery-related applications. BatteryStats inherits the fundamental limitations of procfs/sysfs, and per-process usage information is unavailable for particular hardware components. Furthermore, the granularity of information varies with hardware components. For example, BatteryStats produces component usage statistics on CPU and WNI traffic by reading procfs/sysfs, whereas display utilization is only available for the entire system [45].

- **Battery Historian** [3] converts the report from Batterystats into an HTML visualization that can be seen in a browser.

- **DevScope**, an Android application, is an automatic and online tool used to generate a power model for smartphones. It controls components according to the Battery Monitoring Unit (BMU) update rate and automatically derives the component power model by analyzing the power state changes [32].

- **Eprof** is a fine-grained energy profiler for smartphone applications. Based on the Finite State Machine (FSM) power model, Eprof can analyze the asynchronous energy state of an application, modeling the tail-state energy characteristics of hardware components with routine-level granularity. Energy metering is achieved via a post-processing mechanism using an explicit accounting policy [36].

- **AppScope** is an application energy metering framework for the Android system that uses hardware power models and usage statistics for each hardware component. It provides accurate and detailed information on the energy consumption of applications by monitoring kernel activities for hardware component requests [44].

- **PETrA** is a software-based tool compatible with Android 5.0 or higher smartphones, not requiring any device-specific energy profile. It relies on information provided by publicly available Android tools (specifically dmtracedump, BatteryStats, and Systrace) and is able to estimate the energy consumed by an app at the method level [24].

- **E-MANAFA** is a device-independent, plug-and-play, model-based energy profiler capable of obtaining fine-grained energy measurements on Android devices. Besides being able to calculate performance metrics such as the energy consumed and runtime during a time interval, it allows estimating the energy consumed by each device component (e.g., CPU, WI-FI, screen) [40].

- **Energy Dashboard** [10] is a tool that uses the Android BatteryStats functionalities to automatically collect the power consumption on Android applications, making it possible to execute experiments in a benchmark approach.

2.4 Software aging

The concept of software aging can be found in the literature since as early as the 1990s, although its definition has shifted over the years. [35] described software aging from the perspective of architecture degradation, where software becomes obsolete due to changing requirements and low maintenance over the years. [30] differentiates this type of architectural degradation aging from the one that is used nowadays by distinctly describing "software aging" and "process aging".

The most recent literature describes software aging (described as "process aging" by Huang et al.) as a phenomenon affecting many software systems, characterized by their internal state degradation, progressive performance loss, and the accumulation of errors over mission time, eventually leading to failure. This phenomenon is a problem for many software systems because, besides the performance degradation, it affects the system's reliability, and security vulnerabilities can be exposed [22, 28].

In many cases, software aging is due to ARBs and the accumulation of the effects that these cause on the system. They are difficult to diagnose during testing and appear only after a long execution and under non-easily reproducible triggering and propagation conditions.

Figure 2.3 depicts the "chain of threats" in the context of software aging. In this image, we can see the process until an aging-related failure (ARF) occurs. It starts with the activation of aging-related (AR) faults by aging factors. These factors can be internal such as a function call triggering the execution of faulty code, or external, where elements from the system's environment, like users, are the source of this activation. After that, an accumulation of errors occurs, and eventually, they are propagated through the system-internal environment until a deviation from the system's specification can be noticed by its user, originating an ARF.

If, for example, an application crashes due to insufficient available physical memory caused by the accumulation of memory leaks, the ARB is a defect in the code that causes memory leaks; the aging factors are the input patterns that exercise the code region where the ARB is located and thus activating it; the memory leaks are the AR errors that accumulate; the ARF happens when the application crashes due to insufficient memory, meaning that the errors accumulated and propagated to the system interface.

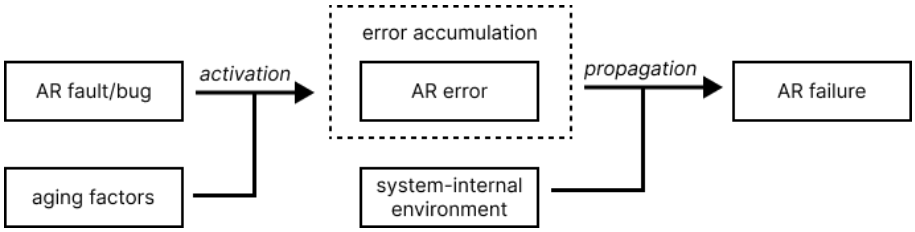

Figure 2.3: "Chain of threats" for an aging-related failure [28]

The main factors that cause the aging of software systems are the accumulation of **unterminated threads, data corruption, memory bloating, memory leaking, file fragmentation, unreleased file locks, and numerical error accumulation**, among others [13] [39].

Cotroneo et al. classified aging-related bugs based on the aging effects they have on the system and identified the following sub-types:

- **MEM**: ARBs causing the accumulation of errors related to memory management (e.g., memory leaks, buffers not being flushed);
- **STO**: ARBs causing the accumulation of errors that affect storage space (e.g., the bug consumes disk space);
- **LOG**: ARBs causing leaks of other logical resources, that is, system-dependent data structures (e.g., sockets or inodes that are not freed after usage);
- **NUM**: ARBs causing the accumulation of numerical errors (e.g., round-off errors, integer overflows);
- **TOT**: ARBs in which the increase of the fault activation/error propagation rate with the total system run time is not caused by the accumulation of internal error states. [28] presents a case where the system run time was incorrectly processed in a missile defense system due to a bug. Still, the error produced was only propagated into a failure if the system had been running for more than eight hours. The error states did not accumulate in this case, so it is considered a TOT ARB.

The **time to aging-related failure (TTAF)** is an important metric for reliability and availability studies of systems suffering from software aging. As the name implies, it corresponds to the time taken until a failure occurs as a consequence of aging-related faults present in the system. Some authors consider the counting

start as the system startup or process creation 3, while others start counting from the first ARF activation [34]. Others start the clock as an aging state (an aging indicator exposes an aging trend) is detected [20]. This document adopts the first definition.

The probability distribution of TTAF is mainly influenced by the intensity with which the system gets exposed to aging factors. So it is therefore influenced by the system workload (and thus by the operational profile and the usage intensity of the system) [34].

However, even in the absence of such faults in the code, aging effects can occur due to the natural dynamics of a system's behavior. This kind of aging is thus referred to as **natural aging**. Among the examples of natural aging are the fragmentation problems experienced by file systems, database index files, and main physical memory. Such aging effects are not related to a faulty code or design, but they are a consequence of the system/application usage over its lifetime [28].

2.4.1 Aging indicators

Aging indicators are explanatory variables that, individually or in combination, suggest if the system state is degrading and thus tell if software aging is affecting it. They can be divided into two categories based on their granularity [28]:

- **System-wide indicators**: Provide information related to subsystems like Operating Systems and Virtual Machines shared by multiple running applications. Indicators in this category can be free physical memory, used swap space, file table size, system load, and others.

- **Application-wide indicators**: Provide specific information about the individual application processes, and some examples are: the resident set size of the process, heap size, response time, launch time, etc.

Past studies show that software aging manifests itself mainly as resource depletion (namely memory depletion) and performance degradation. [19] did an extensive investigation to identify factors and resource utilization metrics correlated with software aging in the Android OS. Considering that this study was performed on a smartphone from a specific brand and that results may vary for other vendors, the following indicators are the ones that were found to be the most correlated to aging.

- **Launch time of activities**: Corresponds to the time taken to complete the initialization of a launched activity. An activity in Android is an application component that provides a screen with which users can interact in order to do something [7]. This is a fundamental user-perceived performance metric that can be used to assess the system performance degradation. Although, we cannot solely rely on this metric, as users might not perform multiple launches and thus not provide sufficient samples to reveal aging.

This is when the next OS performance indicators are useful to measure system degradation, as there is evidence that they are correlated to this metric [19], and their trends can be used to detect the aging phenomenon.

- **PSS** indicates the portion of main memory occupied by a process.
- **Free and Cached memory** respectively indicate the amount of unused memory and the amount of physical Random Access Memory (RAM) used as cache memory.
- **ZRAMinSWAP and ZRAMPhysicalUsed** are related to the usage of zRAM.
- **KSM-Saved, KSM-Shared, KSM-Unshared, KSM-Volatile** are related to the *memory-saving de-duplication* feature of the kernel that merges anonymous pages.
- **LostRAM** which corresponds to the TotalRAM - FreeRAM - UsedRAM, meaning the difference between the RAM usage that Android can compute and the actual available RAM.
- **Garbage collector pause time (GC-paused) and total time (GC-total)**, related to the duration of Garbage collector activities.
- Other memory-related indicators such as the **used slab** and **used buffers** may also be useful.

Table 2.1 was taken from [20] and represents the sets of aging indicators that were used in the aging detection module of the developed Aging Detection and Rejuvenation Tool for Android (ADaRTA), as well as its confidence level. This confidence level represents the likelihood of a trend in that set of indicators representing an ongoing aging phenomenon affecting the system.

2.4.2 Measuring Software aging in Android

The way researchers measure aging throughout the existing studies varies in how aging is identified and the indicators used.

Many studies on software aging in the Android platform consider basic metrics like heap memory of applications and free physical memory [17, 40, 41, 46]. This is a simplistic approach, but sometimes, aging trends can be detected through them.

Cotroneo et al.[20] were by far the investigators that developed a more profound methodology for this purpose. They developed ADaRTA, which has a module specialized for aging detection (Aging Detector module). This module does the following work:

1. From the first timestamp (t0), for each KPI, apply in each window of size W (50), the Mann-Kendall (MK) test with a confidence level C (95%), and the Sen procedure to the samples in the respective window. This returns a

binary value for the MK test, whether a trend is detected in that window (1) or not (0), and a slope of the respective trend value of the test. A 0 is attributed to every window's slope where no trend was detected.

2. Check the persistence of the detected trend. An aging alert is raised every time the slope of a Key Performance Indicators (KPI) is always negative or positive for the last k (5) times, where negative or positive means more severe aging.

3. Check actual aging states that occur when these aging alerts are raised simultaneously in sets of KPIs. These sets of aging indicators are represented in Table 2.1 with the respective confidence level associated. The higher the global confidence level, the more likely an ongoing aging phenomenon is heavily affecting the entire system when trends of that set are present.

Aging indicators	**Confidence level**
system_server PSS and Launch Time	VERY HIGH
system_server PSS and GC Paused or Total Time	VERY HIGH
Launch Time	HIGH
system_server PSS	HIGH
Free Memory, Cached Memory, Lost RAM	MEDIUM
Used PSS, ZRAMinSWAP, ZRAMPhysicalUsed	MEDIUM
systemui, huawei.systemManager GC Paused/Total Time	LOW
One of KSM-*, Used slab, Used buffers	LOW
One of systemui, surfaceflinger, mediaserver PSS	VERY LOW

Table 2.1: Sets of aging indicators and corresponding confidence level.

Besides the Launch Time, it is noticeable that all metrics are related to memory consumption. Cotroneo et al. [21] found this and that the main aging issues can be confined to key processes of the Android OS that exhibit a systematic inflated memory consumption. It was concluded that the *System Server* process plays a key role in determining the bad performance in responsiveness.

2.5 Related Work

Software aging is highly related to software dependability, and much research and thousands of papers exist about the subject. This section pretends to highlight the research done about the topics that are the main focus of this work: Software aging (and rejuvenation), Mobile (Android) platforms, and Energy consumption of mobile devices/applications.

Garg et al. [25] studied software aging issues from systems in operation by monitoring a network of UNIX workstations over 53 days. This research adopted Simple Network Management Protocol (SNMP) to collect data on resource consumption and OS activity, including memory, swap space, file, and process utilization

metrics. The analysis found that 33% of the reported outages were related to resource exhaustion, particularly memory utilization (which exhibited the lowest time-to-exhaustion among the monitored resources).

Araujo et al. [14] proposed an investigative approach to detect software aging by monitoring a specific Android application's memory utilization when running continuously for a sustained period. The Monkey tool was used to generate a stressful workload to accelerate software aging and quickly identify its effects. Linux utilities were used to collect information about the resource utilization of the device. Experimental results confirmed both the effectiveness of the procedure and the existence of the software aging phenomenon in the application.

Corral et al. [17] presented an approach to relate smartphones' energy consumption with the device's operational status, using parameters exposed by the OS. The solution is presented as an Android app ("CharM"). The authors explain the data collection strategy and the advantages and limitations of performing the analysis at the software application level.

Shahriar et al. [41] defined some memory leak patterns specific to Android applications and then used fuzz testing to emulate memory leaks and discover the vulnerabilities. Their goal is testing for robustness against memory leaks rather than aging detection.

Hussein et al. [31] applied a methodology that characterizes energy consumption and performance of an Android device to correlate the choice of the GC algorithm to the experienced energy consumption. The authors discuss alternative GC designs that extend Dalvik's default, mostly concurrent, mark-sweep collector, with generations and on-the-fly scanning of thread roots.

Qiao et al. [37] conducted experiments to verify aging manifestations under various aging conditions from the user's (response time metric) and system's (memory usage) viewpoints and discussed their differences with the aging indicators in Linux. In this study, aging was caused by injecting memory leaks in different areas (Dalvik or Native Heap) and processes with distinct priorities (cached, persistent) of default Android OS apps. The results showed various manifestations between the user's and the system's point of view, according to what heap leaks were injected into and the process priorities.

Cotroneo et al. [19] went further in the research as other metrics were used for the system's viewpoint, such as storage usage, garbage collecting times, and other task-level metrics. The correlation between these metrics and the response time was calculated, and statistical methods were used to identify the most influential factors in software aging. In this study, the authors were able to point out specific Android OS processes and components where an aging trend is present.

Qiao et al. [38] investigated software aging indicators prediction in Android, focusing on indicators such as the system's free physical memory and the application's heap memory. They used a Long Short-Term Memory Neural Network to make this prediction because of the unpredictability of user behavior when interacting with applications. They then compared the results and accuracy of these predictions with other prediction methods in the history of software aging

research.

Cotroneo et al. [20] also developed a tool named "ADaRTA" that i) performs selective monitoring of system processes and trends in system performance indicators; ii) detects the aging state and estimates the time-to-aging-failure through heuristic rules; iii) schedules and applies rejuvenation, based on the estimated time-to-aging-failure.

Most studies are limited to analyzing aging in devices of a specific Android vendor and version of the OS. Cotroneo et al. [21] extended its research to devices of different vendors under various usage conditions and configurations. Besides the expected loss of responsiveness and unjustified depletion of physical memory, the results revealed differences in the aging trends due to the workload factors and type of running applications, as well as differences due to vendors' customization. By tracking several system-level metrics, the authors showed that bloated Java containers significantly contribute to software aging and that it is feasible to mitigate aging through a micro-rejuvenation solution at the container level.

Cotroneo et al. [22] proposed the first complete resource-specific aging detection and rejuvenation solution for Android, acting selectively on bloating system data structures. This micro-rejuvenation technique avoids unavailability by rejuvenating only selected data structures instead of system processes or the whole OS. It is transparent to the end-user, who perceives no aging-related failure yet no downtime.

2.6 Gaps in the literature

We can see that software aging is a widely explored subject, and much research has been done, not only on mobile devices but also on other platforms, by analyzing different metrics of the system to find out if it is affected by software aging and what are the main signs of this phenomenon.

On the other hand, some research was also done to explore the battery consumption of mobile applications and devices relating to the type of workload and status of the system. Multiple tools were also developed with this goal.

However, no research can be found about this phenomenon's impact on the battery consumption of devices.

The following Table 2.2 shows what the main themes of this work (Software aging, Mobile platform, and Energy consumption) the studies referenced in the previous Section 2.5 address. This investigation tries to fill the gap by connecting the research on software aging on the mobile platform with energy consumption.

	Software aging	**Mobile platform**	**Energy consumption**
Garg et al. [25]	x		
Araujo et al. [14]	x	x	
Corral et al. [17]		x	x
Shahriar et al. [41]		x	
Hussein et al. [31]		x	x
Qiao et al. [37]	x	x	
Cotroneo et al. [19]	x	x	
Araujo et al. [38]	x	x	
Cotroneo et al. [20]	x	x	
Cotroneo et al. [21]	x	x	
Cotroneo et al. [22]	x	x	
This work	**x**	**x**	**x**

Table 2.2: Table of related work.

Chapter 3

Methodology and Experimental Procedures

This chapter outlines the methodology employed to conduct an empirical analysis of the impacts of software aging on the battery performance of mobile devices. Figure 3.1 outlines the experimental methodology, and it is structured to provide a clear framework for the research objectives in a way that has been carefully delineated to provide a clear understanding of the stages taken to achieve the objectives of this research. Thus, its structuring ensures the reliability and validity of the conclusions reached and the possibility of consistent reproduction of our findings.

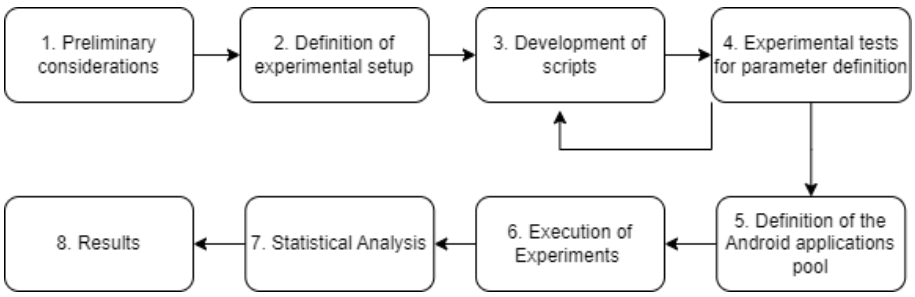

Figure 3.1: Experimental methodology Flow Chart

3.1 Preliminary Considerations

The underlying principles of this methodology are based on previous studies about software aging in the Android platform [14, 19–22, 37, 38, 43]. Three main decisions were initially taken based on those same studies: What data was to be collected, How aging would be achieved, and How long should the experiments last.

What data was to be collected?

In order to monitor the performance of the device regarding software aging, KPI regarding system memory and its management mechanisms had to be collected. These indicators were chosen based on other studies referenced in section 2.4. Specifically, the KPIs collected were:

- PSS of the process System Server
- Cached RAM
- Lost RAM
- Used zRAM
- Total PSS
- Garbage Collection Total Time of the Process System Server
- Garbage Collection Pause Time of the Process System Server

Besides that, battery consumption was another crucial metric to collect.

How was aging going to be achieved?

Previous studies [14, 19–22, 38, 43] into software aging in the Android environment have used the Exerciser Monkey tool to generate randomized actions that simulate user inputs. These actions create a representative workload for the system under analysis, thus inducing its aging over time. Based on the approach in such studies, this work follows an analogous strategy to subject Android applications to the aging process.

How long should the experiments last?

The experiment duration used by previous studies [14, 19, 21, 22, 37, 38, 43] varied tremendously, going from 2 hours [37] to 72 hours [14, 43] or until crash or battery drain [38]. Cotroneo et al. [21], based on the results of one of his previous studies [19], concluded that in the experiments where aging was present in the Android platform, all its trends were noticed after 6 hours of data. With this in mind, we decided to run our experiments until the device's battery drained or the application crashed when it gave us a minimum of 6 hours of data. It is also important to notice that the experiments run by others during 72 hours could not be replicated in this research since we are studying software aging and how it impacts battery consumption. To run one application for 72 hours, we would have to plug the device into an energy source during the experiment, eliminating the possibility of gathering consumption data.

3.2 Experimental Setup

The testbed used to run the experiments is represented in Figure 3.2 and consists of two main devices: a PC with 16 GB of RAM Windows 10 Home (Build 19045), powered by a 2.80 GHz Intel Core i7-7700HQ CPU with 4 Cores 8 Threads; and a smartphone (Oppo A16s) running on Android 13, with 5000 mAh of battery capacity, 4 GB of RAM and powered by an Octa-Core CPU Helio G35.

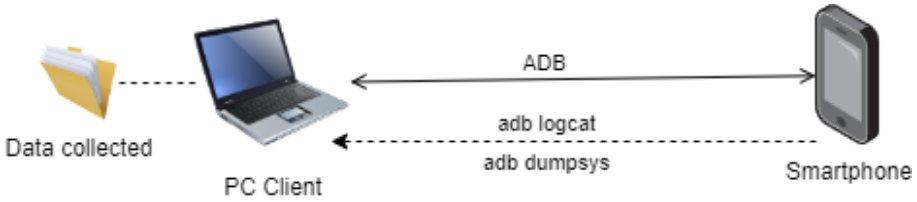

Figure 3.2: Testbed

Choosing the tools to assist the experiments was also crucial in the research. After considering all the options, we chose the following tools:

- To create a connection between the PC and the Android smartphone, **Android Debug Bridge (ADB)** was our choice for the experiment since it provides a stable wireless connection that makes it possible to collect data throughout the experiment constantly. Besides that, it is an official Android tool, and its command-line nature makes it easy to use in automation scripts.

- As explained in the previous section, **Exerciser Monkey** was chosen for workload generation since it is used to age Android applications in most related studies.

- **Dumpsys** and **Logcat** were chosen to retrieve information from the smartphone from the PC command line since they are strongly coupled with the ADB tool. Dumpsys collected most of the memory parameters: PSS of the process System Server, Cached RAM, Lost RAM, Used zRAM, and Total PSS. Logcat was used to collect Garbage Collection logs, making it possible to calculate the following times: Garbage Collection Total Time of the Process System Server and Garbage Collection Pause Time of the Process System Server

- **Batterystats** was chosen to retrieve battery consumption information from the smartphone since it is a lightweight tool that doesn't consume significant device resources, runs in the background, and doesn't impact the device's performance while collecting data. It is an official Android tool, does not rely on third-party apps or services, and just like dumpsys and logcat, it is scriptable and automatable due to being easily called from the 'adb dumpsys' command.

3.3 Development of Scripts

Three shell scripts were created to assist the experiments:

- The **Exerciser** ('exerciser.sh') runs the exerciser monkey tool to provide a workload for the tested application. This aims to provide a substantial workload for the application under testing. It is important to note that the significant amount of events generated by the tool in question is included in this script. This means this quantity is not the limiting factor in determining the experiment's conclusion.
- The **Collector** ('collector. sh') is a centerpiece in obtaining information from the Android device, playing a crucial role in obtaining data for analysis and continuous monitoring. It uses tools such as 'dumpsys' and 'logcat', as well as the Android file system located in '/proc/meminfo', to collect a wide range of vital information. Data collection occurs every 30 seconds, ensuring regular and timely capture of relevant information.
- The **Runner** ('run.sh') was created with the purpose of preparing the Windows file system to store the files collected from the smartphone and coordinate the simultaneous execution of the exerciser and the collector.

3.4 Experimental Tests

Some experimental tests were run to tune the scripts mentioned above by defining the following parameters.

Sample rate

The sample rate of data collection needed to be tuned. From previous studies [14, 19–21, 38, 43], we noticed that the sampling interval varied significantly, ranging from 5 seconds to 2 minutes. We understand that the more data we have to monitor system performance, the more our analysis will improve. After some preliminary tests with 10 and 20 seconds of sample rate, 30 seconds was chosen for this parameter since it was a period in which the system could constantly retrieve data without desynchronizing.

Workload throttle

One parameter that needed to be tuned was the throttle of the exerciser, which denotes the time passed between each event generation. When analyzing previous studies, we realized that the common values for this event generation rate generally varied between 500ms and 1000ms. In order to decide which value to use, we conducted a series of preliminary tests, the results of which are summarised in

Table 3.1. During these tests, we experimented with different acceleration values to determine which would be most suitable for our experiment.

Throttle	**Behavior**
500 ms	The device crashed due to too many inputs.
1000 ms	The device can handle all the inputs but slow event generation.
800 ms	Good balance between workload velocity and handling of inputs by the device.

Table 3.1: Workload throttle tests done and observed behavior.

In the end, **800 ms** was the time used for the throttle of the event generator.

This choice was fundamental since this methodology stage played an essential role in guaranteeing future experiments' integrity and uninterrupted operation, ensuring that all the necessary data was collected without unexpected occurrences.

3.5 Selection of Android Applications

At this stage, we selected a set of Android applications that would serve as research targets. These apps were chosen based on strictly established criteria, which included:

- Available at the Google Play store or F-droid
- At least 1M downloads
- A rating of at least 3.5 (of 5)
- Be in a Western European Language so that its UI can be understood
- Does not use high energy demanding sensors, specifically GPS.

We chose five applications of multiple four different categories to guarantee that a large spectrum of applications was covered, contributing to the generalizability of the research. Table 3.2 provides the group of Android applications used in the research, categorized by functionality.

3.6 Execution of Experiments

With the experimental setup ready and the applications pool defined, the planned experiments were executed. This phase involved systematically running the chosen Android applications using the developed scripts, collecting data, and recording observations. Each experiment followed the following structure:

Category	Name	Category	Name
Utility	Unit Converter Ultimate Open Camera Turbo Cleaner Notion Simple Notes	Game	Uno Monopoly Go Bubble Shoot Candy Crush Saga Solitaire
Social Media	TikTok Weibo Reddit Twitter Facebook	Browser	Baidu Chrome Brave Edge Firefox

Table 3.2: Android application pool.

1. Charge the smartphone battery up to 100%.
2. Restart the device.
3. Connect the smartphone to the laptop through a wireless ADB connection.
4. Guarantee all the predefined test conditions are applied (Table 3.3).
5. Clean the app's cache.
6. Open the app and fix it so it always stays in the foreground during the experiment.
7. Unplug the smartphone USB cable so it stops charging.
8. Execute run.sh, which executes Exerciser and Collector simultaneously.
9. Wait until the battery drains or the app crashes.

Table 3.3 summarizes the conditions that our device was configured to every time to ensure the reliability and consistency of our experimental setup.

Condition	Status
Battery	100%
Screen Brightness	50%
Volume	0%
Bluetooth	Off
NFC	Off
GPS	Off
Others	Off / 0

Table 3.3: Device conditions for each experiment.

Our Exerciser was configured to a random seed every time it ran. This means that each experiment executed had a completely different sequence of events injected into the application. So, we ran each application twice to embrace a larger portion of functionalities in each experiment. Due to time constraints, this was possible to do with every app except one of each category.

3.7 Data processing and Statistical Analysis

For the analysis of the data collected from the device, a temporal analysis was made based on the ADaRTA tool [20]. The authors of this tool and article describe the tools' algorithm and its parameters, such as sample period, window size, and others. All this was described in Section 2.4.2. We used this information to develop two Python scripts responsible for the data processing and statistical analysis of the data retrieved from the device. Firstly, they go through the system data collected and gather only the important values, creating a complete matrix with every indicator (columns) value on the respective timestamps (rows). We created an extra 'aging' column initialized to 0 that stores the aging state of the system. In second place, timestamps with unsuccessful indicators collected are discarded. Then, a sliding window is applied to calculate the trends, and every five consecutive times an aging trend is detected, the aging column is incremented by one. This increment is based on Table 3.4, which shows the KPI trends or a combination of trends considered an aging software behavior. This table is an adaptation of the one presented in Section 2.4.2. Although not having all the conditions, these are the four most reliable according to [20], and the only ones that had data we could collect.

Conditions
Increasing trend of System Server PSS
Increasing trend of System Server PSS and increasing trend of GC Paused Time
Increasing trend of System Server PSS and increasing trend of GC Total Time
Decreasing trend of Free RAM and increasing trend of Cached RAM and Lost RAM

Table 3.4: Aging conditions used to identify aged system states.

Finally, the correlation between the 'aging' and 'energy consumption' columns was calculated. We considered two approaches to dealing with the aging variable:

- **Approach 1**: Aging is represented by the natural number resulting from the previous steps. We investigated whether the aging factor, represented by a level with five possible values (0 to 4), correlates with the battery consumption observed in different experiments. A significant correlation in this approach would indicate that the more aging conditions are met concerning the different KPIs, the greater the impact on battery consumption. To facilitate easier identification, we shall refer to this variant as 'V0'.
- **Approach 2**: Aging is represented by a binary value corresponding to the two system states: aged (1) or not aged (0). From this perspective, three different variants of the aging factor were used:
  - **V1**: If the aging of the previous stage was equal to or greater than 1, a value of 1 was assigned; otherwise, 0 was assigned.

  - **V2**: If the aging of the previous stage was equal to or greater than 2, a value of 1 was assigned; otherwise, 0 was assigned.
  - **V3**: If the aging of the previous stage was equal to or greater than 3, a value of 1 was assigned; otherwise, 0 was assigned.

  All these variants result in a binary vector representing the aging factor, with the second and third requiring more aging conditions to identify the system as aged. We decided not to go further than V3 because three conditions met already mean a very high likelihood of the system being in an aged state, based on the article that supported this methodology [8]. Also, further restrictions in identifying the aged state would mark a lot of states as not aged when the system would be.

Both approaches use the 'battery consumption' variable as a continuous measure however, due to the different nature of the 'aging' variable, it was necessary to choose suitable statistical tests to analyze the relationships. In the first approach, 'aging' is considered an ordinal numerical variable from 0 to 5. With this in mind, we chose **Spearman's rank correlation coefficient**. This non-parametric statistical test assesses how much a monotonic function can describe the relationship between two variables. Assuming a 95% confidence level, if the p-value resulting from the test is less than 0.05, we consider there to be a significant correlation. In these cases, the correlation coefficient returned by the test ranges between -1 and +1, reflecting both the magnitude (absolute value) and the direction of the correlation (positive or negative)

For the second approach, we divided the battery consumption data into two groups:

- **B1**: Battery consumption values gathered from the system in an aged state (1).
- **B0**: Remaining values gathered from a non-aged system (0).

We then applied the non-parametric **Wilcoxon Rank Sum Test** to assess whether there was a statistically significant difference between these two groups. This analysis allows us to determine whether software aging impacts battery consumption in a statistically relevant way.

The hypotheses formulated for this test were:

- **H0**: B1 <= B0, meaning that battery consumption is inferior or not significantly different in aged and non-aged states.
- **H1**: B1 > B0, meaning that battery consumption is affected positively by aging in this experiment.

For the same confidence level (95%), if the resultant p-value of the test is less than 0.05, this leads to the rejection of the null hypothesis, indicating that aging significantly impacts battery consumption in the context of the experiment in question.

Other values were considered for the sliding window size (W, where 50 was used) and the number of consecutive slopes to conclude the presence of an aging trend (where 5 was used). Some variations of these parameters were taken into account and tested. However, the ones we used provided the most consistent results, detecting more trends regarding individual KPIs and the aging column. Furthermore, these were the numeric values supported by the literature [20].

Chapter 4

Results and analysis

In this chapter, we present and discuss the outcomes of our investigation. We will unravel the potential patterns and trends that emerged from the data, shedding light on whether or how software aging might influence battery consumption. This chapter is divided into two sections. First, we analyze the whole pool of results based on the different approaches to the aging vector (by level and binary). In the second section, we investigate if different app categories behave differently regarding the relationship between aging and battery consumption.

Appendix A shows the data of the 36 successful experiments and the respective correlation metrics between aging and battery consumption. Overall, these experiments consist of 324 hours of experiments.

4.1 Overall aging impact on battery usage

In Figure 4.1, the graph on the left shows, for the first approach to the aging variable, the number of experiments where aging and battery consumption are significantly correlated (in orange) and the portion of these that are positively (green) and negatively (red) correlated. The graph on the right consists of a box plot for this same approach, where it is possible to see the distribution of correlations through the experiments that showed any.

From both graphs, we see that of the 36 experiments, 21 showed a statistically significant correlation, 9 positive and 12 negative. The average correlation coefficient is 0.005, and the median is -0.075. The strongest negative correlation detected was -0.289, and the strongest positive correlation was 0.451, which could be seen as an outlier since the next strongest one was 0.237. Therefore, most experiments showed a weak correlation (inferior to 0.3), with similar disparity for both directions and a median and average close to 0.

With this data, we conclude that we cannot infer that aging affects battery consumption if we identify aging by levels based on the number of conditions it represents.

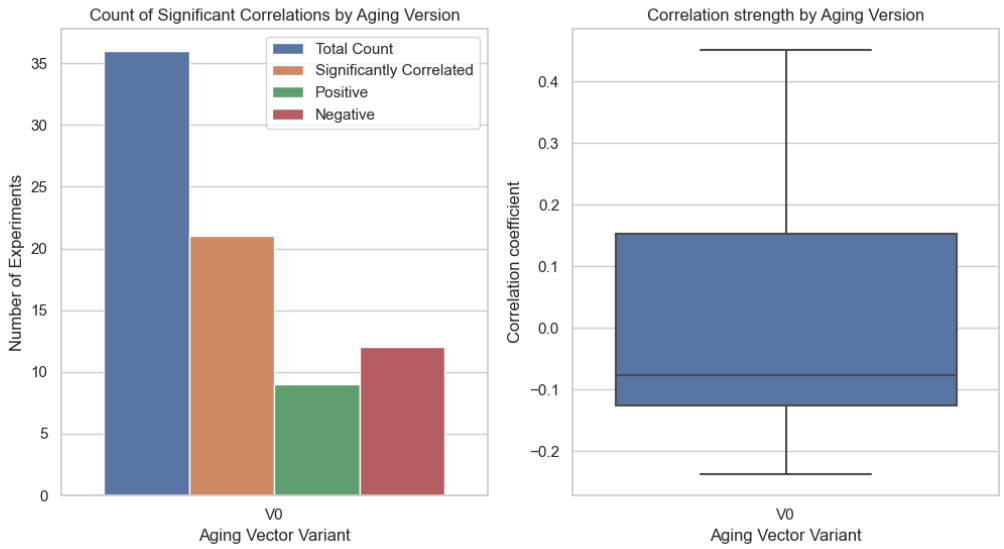

Figure 4.1: Barplot of experiments with significant correlations (on left) and Box plot of Correlation strength (on right)

The graph in Figure 4.2 shows, for each of the other versions, how many present a greater battery consumption when the system was aged versus the battery consumption measured when the system was not identified as aged. This graph represents the Wilcoxon rank sum test results applied to every experiment. It shows for each version of the binary aging vector (V1, V2, and V3 corresponding to a minimum of 1, 2, and 3 favorable conditions to identify each window as aged) the total number of experiments (in blue) in relation to the number of experiments where aging was detected (in orange) and the portion of those that was detected a greater battery consumption amongst aging states vs non-aging states (in green).

Firstly, we notice that the orange bars decrease amongst the different versions of the binary aging vector. This was expected since the more simultaneous aging conditions are needed to identify the system as aged, the fewer times system states will be marked as so.

As we can see, in V1 (Only 1 condition to identify as aged), only 9 out of 36 experiments showed a greater battery consumption in the aged instances, corresponding to 25% of the cases. In V2, 9 out of 16 experiments (56%) had a greater battery consumption when the system was aged. In V3, 4 out of 8 (50%) experiments had the same behavior.

The way we interpret the results is the following:

- In V1, every experiment was 'diagnosed' with aging, meaning that there is a greater possibility for misidentification of aging in these experiments since only one of the aging conditions had to be met. This would lead to only a small proportion of experiments manifesting a greater battery consumption while the system was aged.

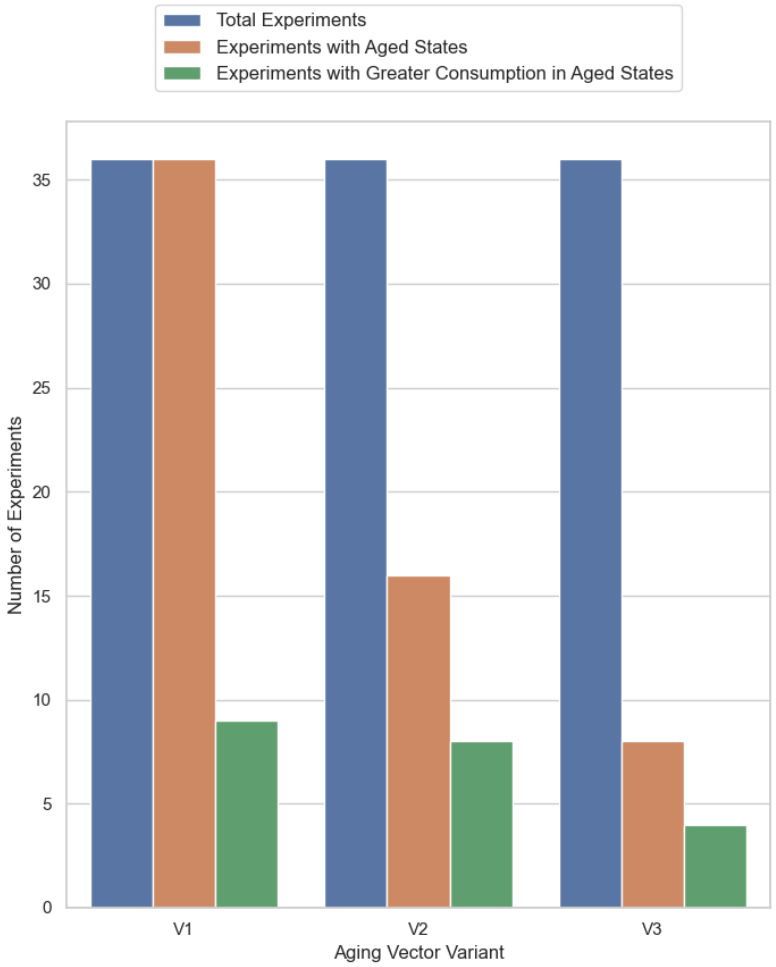

Figure 4.2: Count of Significant Correlations by Aging Vector Version

- In V2 and V3, aging was more restrictively diagnosed in the experiments since more conditions had to be in favor, possibly leading to a more precise identification of the aging states. This led to a smaller proportion of experiments diagnosed with aging, and around 50% of those exhibited a greater battery consumption while the system was aged. We should also notice that this means that the other half of the experiments showed no increase or even a decrease in battery consumption when the system aged.

4.2 Categorized aging impact on battery usage

After analyzing the pool applications as a whole, we checked if different app categories had different behaviors in this context. The first approach of the aging vector by levels was not considered since no solid correlations were noticed in the previous section. So, in this section, we only analyze the second approach, where aging is characterized as a binary factor (0 or 1) based on the number of aging conditions met (minimum of one, two, or three).

Table 4.1 displays two metrics related to the experiments that can be useful in interpreting some results. These metrics are the average duration of the experiments per category and the average battery consumption per 30 seconds of execution.

Category	**Average duration (h)**	**Average battery consumption / 30 sec (mAh)**
Utility	10,5	2,39
Game	7	3,75
Social Media	9,5	3,16
Browser	9	2,78

Table 4.1: Experiment metrics per category.

Game category

Figure 4.3 presents a bar plot where, for each variant of the aging vector, a blue bar indicates the total number of experiments made on applications of the game category, and an orange bar corresponds to the portion of experiments that showed aging states.

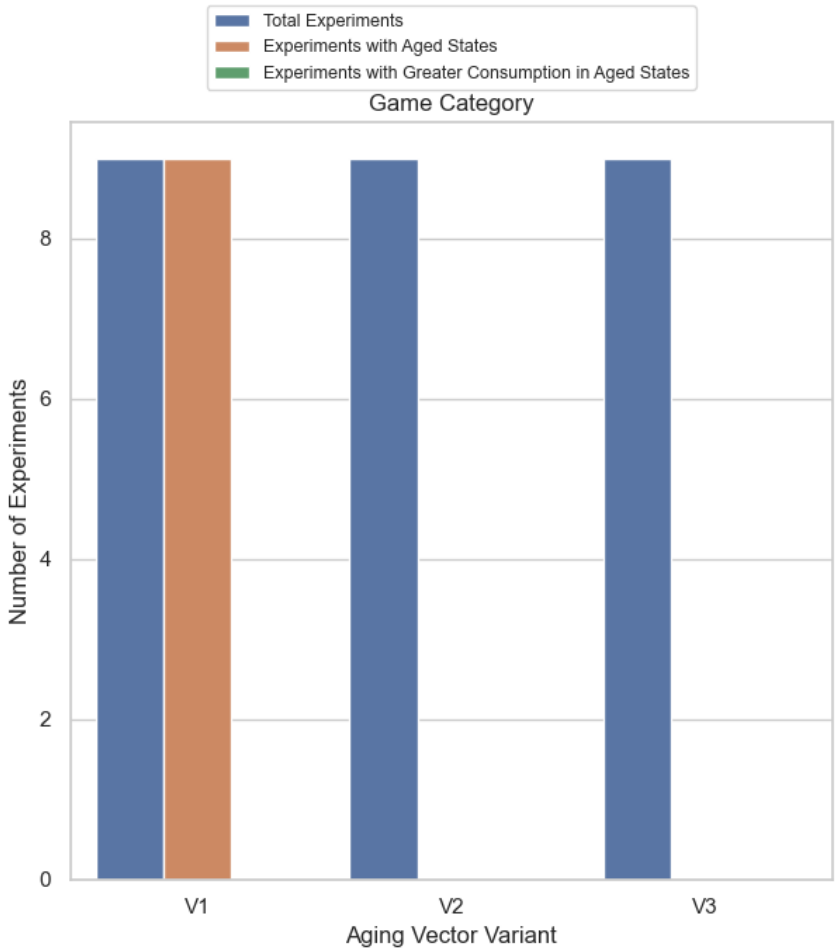

Figure 4.3: Count graph of the game category.

Although no cases where consumption was associated with aging were recorded in this category, we should note that only in V1 were there opportunities for this relationship to manifest since, at most, one aging condition was recorded simultaneously across every application. Mobile games are known to be one of the largest energy-consuming categories in the mobile space due to their heavy computational resources (GPU and CPU). The data in Table 4.1 supports this statement, as the experiments with mobile games averaged the least duration and the most battery consumption per 30 seconds. We speculate that the main reason no relationship was detected between aging and battery consumption for this category is that, in case this relationship exists, it is not significant in comparison to the natural behavior of the application in terms of energy consumption.

Browser category

Figure 4.4 graphically presents the 'browser' category results. On the left, a count plot is displayed with the total number of experiments in this category (in blue), the number of experiments where aging was detected (in orange), and the portion of those where battery consumption was greater when the system was considered aged (in green). These values are presented across the three variants of the binary aging vector - V1, V2, and V3. The graph on the right consists of a box plot that exhibits the difference in average consumption per 30 seconds across the apps of this category. This difference is computed between the consumption measured in non-aged states (B0) and aged states (B1). Only the apps where an increase was noticed from B0 to B1 were considered for this plot.

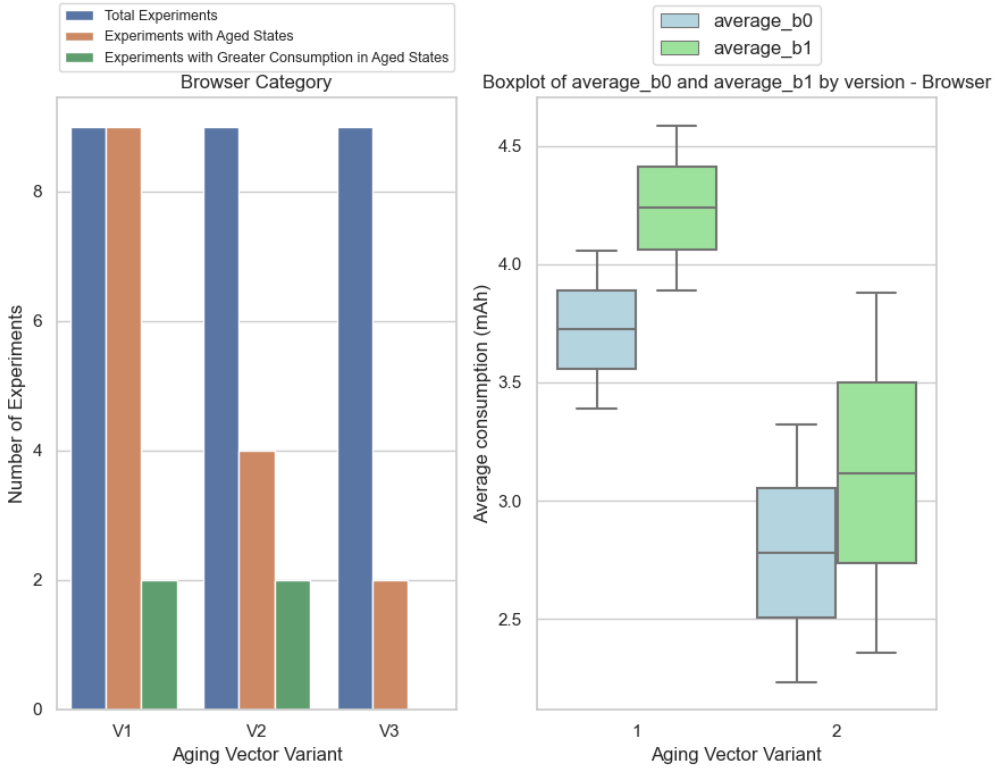

Figure 4.4: Count plot (on left) and Box plot (on right) of the browser category.

This category does not seem to have any solid pattern since in V1, only in a small portion of the experiments with aging, the battery consumption had any relation with that factor; in V2, this relation was present in half of the aged applications, and none V3. This variety of results could be explained due to the nature of these applications. Depending on the workload applied to the browser, it can navigate to many different pages with distinct resource demands and energy profiles. This could be the main reason for such heterogeneity in results, which makes us believe that it is not the most adequate category to analyze software aging behavior. When analyzing the battery consumption difference from B0 to B1, a 0.5 mAh increase was recorded in V1 and V2.

Utility category

Figure 4.5 presents the same information as the previous category but in relation to the applications categorized as utilities.

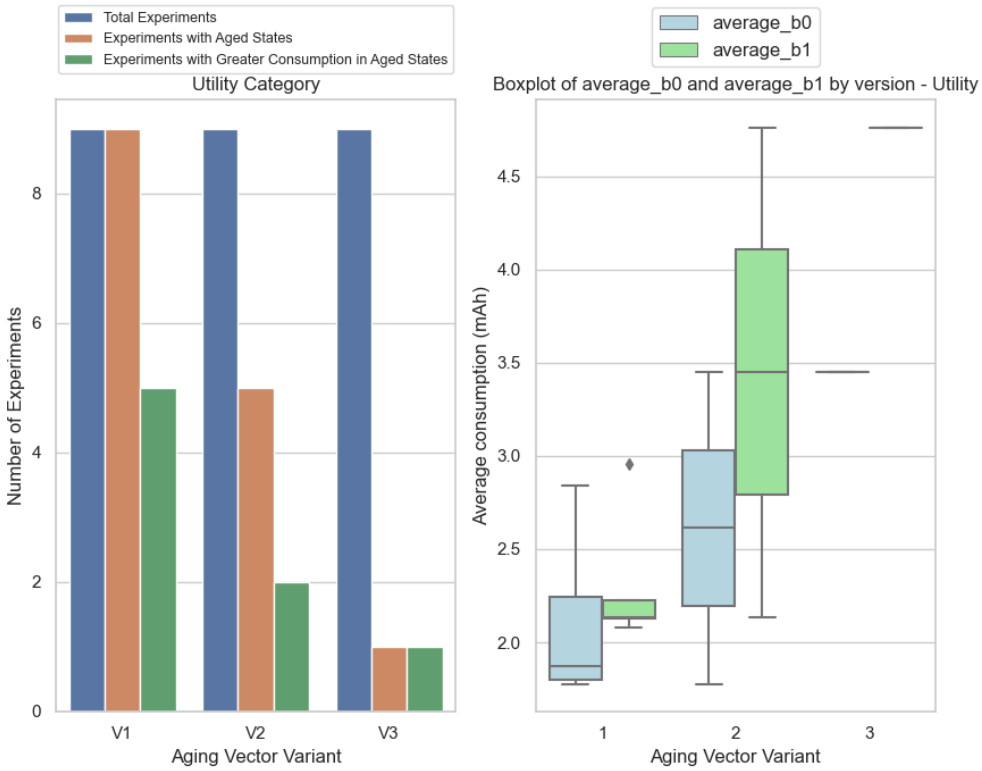

Figure 4.5: Count plot (on left) and Box plot (on right) of the utility category.

The utility category, characterized by being a category of applications with low resource usage and battery consumption, had a decrease in the number of experiments where battery consumption was considered to be affected by aging along the different versions. If we consider only V1, where only one condition is necessary to identify aging, this is the category where battery consumption was considered to have a greater relationship with aging. Based on these applications' low resource usage and energy-consuming nature (supported by our data in table 4.1), we would expect this category to manifest a higher relation between aging and battery consumption, assuming that there is one. If that were the case, any possible effect aging had would have a greater impact since an increment in energy consumption would be more visible. However, stronger signs of aging, specifically V2, did not support this hypothesis, and in V3, we do not have enough data since only one experiment reached aging states. When it comes to the difference in energy consumption from the non-aged states to the aged states across the experiments that showed an actual increase, it was more noticed the stronger the aging signs were. In V1 the applications saw an increase of around 0.5 mAh, and an increase of up to 1.3 mAh in V2 and V3. We should notice that these increases in consumption are significant (10% to 40%), but we cannot discard the cases where no relationship between aging and battery consumption could be inferred.

Social Media category

Figure 4.6 presents the same plots as the others but for the social media applications.

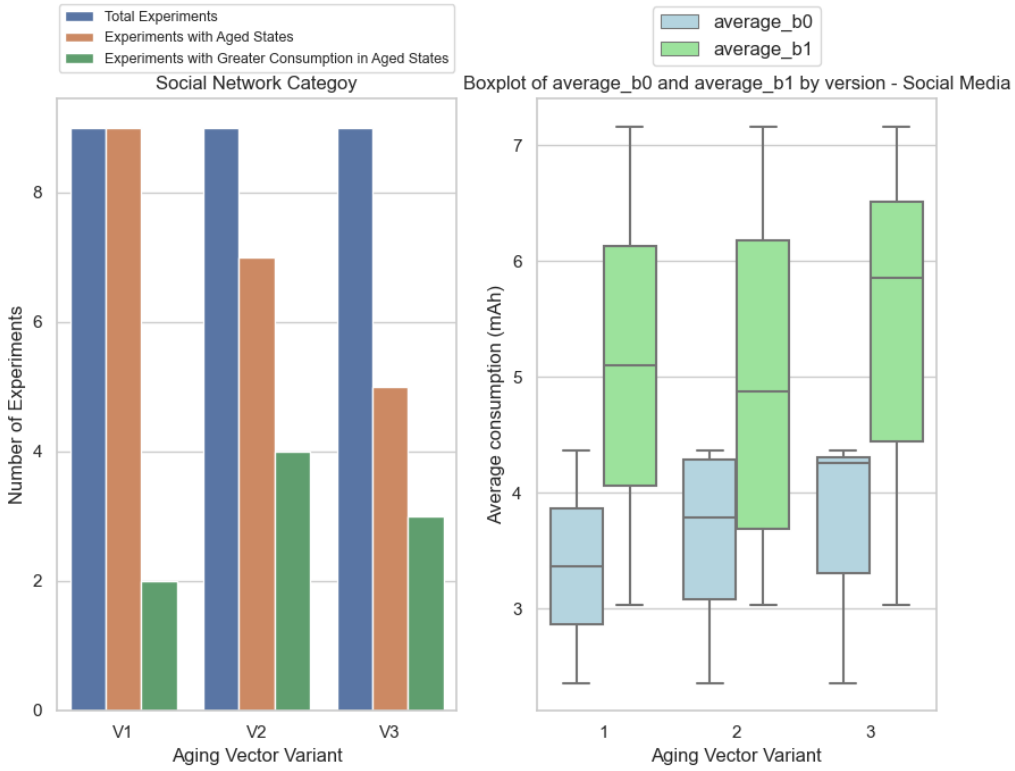

Figure 4.6: Count plot (on left) and Box plot (on right) of the social media category.

The Social Media category had the most cases where aging was detected, and battery consumption increased in aged states amongst V2 and V3 variants. In both, around 60% of the cases where aging was detected, an increase in battery consumption was noticed. Based on our results, social media applications were the apps that, under strong aging signs (minimum of two and three simultaneous aging conditions), most consistently showed an influence on battery consumption. When looking at the difference in energy consumption between non-aged and aged states, this was also the category where this increase was more visible, averaging 0.7 to 2.8 mAh, which corresponds to 30% to 60% more energy consumed in aged states.

4.3 Discussion

Based on the presented results, the key findings we derive from this research are:

- If we consider aging a multi-level variable characterized by the number of aging conditions simultaneously present in each state, no strong correlation was observed between aging and energy consumption.
- However, When we treat aging as a binary variable, the following observations are made:
  - Weakly identified aging states, where only one aging condition is observed, see no relevant impact on battery consumption.
  - When strong aging signs are present in the system, i.e. at least two and three aging conditions are recorded simultaneously, half of the applications analyzed show an increase in battery consumption. This suggests that these aging states can significantly influence the energy consumption of applications.
- When considering the different app categories, based on our data, we can conclude that:
  - Mobile games do not reveal strong aging signs, and in the weak states of aging manifested, it was impossible to infer any correlation with battery consumption.
  - When browsing, we should not expect a relevant increase in battery consumption due to aging.
  - Utility applications are low energy consuming, and we do not have enough evidence to say that when aged, the battery consumption is influenced.
  - Social Media applications, when submitted to strong aging conditions, have a high probability of seeing their battery consumption impacted by 30% to 60%

When we consider the results as a whole, intending to address the main question of this study, we can summarize by stating that aging does not directly affect battery consumption. In the spontaneous cases where we saw a correlation, the increase in battery consumption could be explained by other factors, such as the execution of higher CPU and GPU-demanding tasks. Although we do not have data to confirm this assumption, we know that the battery consumed was higher in aged states versus non-aged states in only about half of the total experiments where aged states were achieved. This is enough to conclude that our data does not support the hypothesis that aging affects the battery consumption of an Android device.

Chapter 5

Planning

This chapter offers an overview of the global planning for this work. The first section presents the work plan in the form of two Gantt charts, illustrating both the anticipated and the actual achievements. The second section outlines some challenges that posed potential threats to the success of this work.

5.1 Work plan

This section provides a timeline of the tasks and the major milestones that were expected to be achieved versus what actually was achieved.

Initially, planning was done considering the delivery at the end of June. The following list and Fig. 5.1 show the expected tasks in the Gantt chart that was designed with that goal in mind.

- **Aim 1**: Clarification of the software aging techniques to implement, what tools to use, and familiarization with them.
- **Aim 2**: Execution of the experiments.
- **Aim 3**: Validation of the experiments.
- **Aim 4**: Analysis of the results and re-execution of experiments (if needed)
- **Aim 5**: Writing the dissertation.

However, some obstacles appeared, especially in the definition of the aging technique to implement, retarding the progress of the dissertation and changing our initial plan. Fig. 5.2 presents the final Gantt with the fulfilled plan.

- **Aim 1**: Choice and experimentation of the software aging technique to implement.
- **Aim 2**: Choice of tools for data collection and experimentation with them.

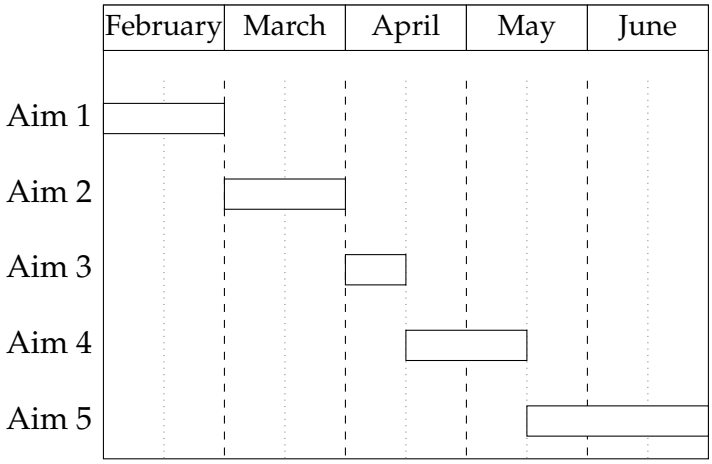

Figure 5.1: Expected Gantt Chart

- **Aim 3**: Definition of parameters and development of scripts for experiment automation.
- **Aim 4**: Execution of the experiments.
- **Aim 5**: Development of scripts for data processing.
- **Aim 6**: Statistical analysis of the results.
- **Aim 7**: Writing the dissertation.

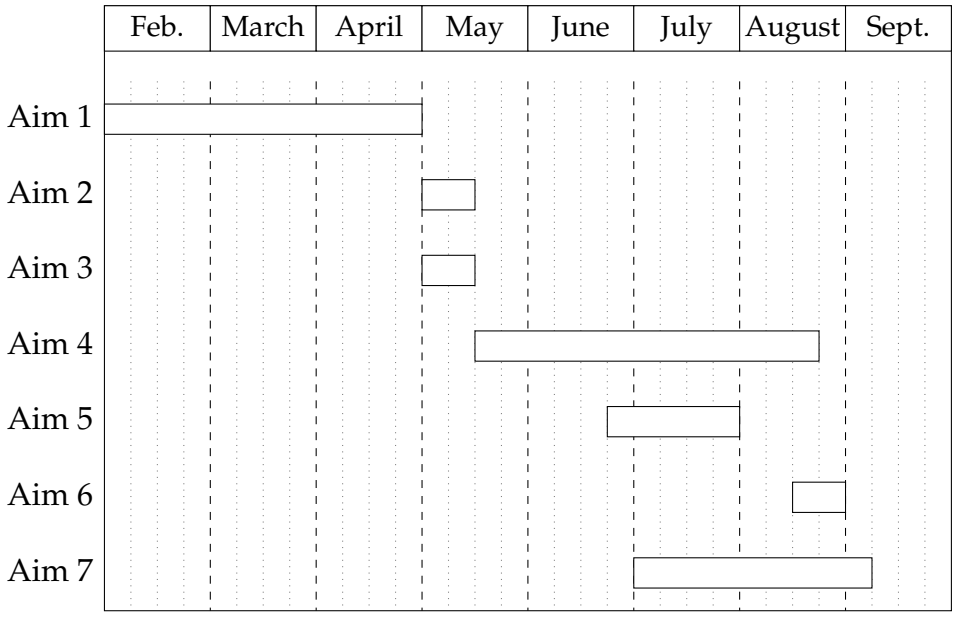

Figure 5.2: Final Gantt Chart

5.2 Challenges

This section presents some challenges that deviated from our initial planning.

Challenge #1

The first challenge to our investigation was the choice of the aging technique to implement. The initial approach aimed to use fault injection to incite aging symptoms in the applications. However, this approach was not successful, retarding the progression of the work done.

Challenge #2

Another challenge to this work was the long duration of the experiments. These were very time-consuming since most of them ran until the smartphone battery drained. Besides that, every experiment that unexpectedly ended early cause of a crash had to be discarded. This led the number of experiments available for the statistical analysis in the end to be lower than expected.

Chapter 6

Conclusion

This work presents an approach to examine the impact of software aging on the energy consumption of mobile devices. Previous research has already delved into the realm of software aging, particularly within the Android platform, where several approaches aimed to analyze the presence and effects of this phenomenon within the OS. Additionally, the topic of energy consumption in mobile devices and applications has garnered significant attention in the scientific community, leading to the development of numerous tools designed to measure and optimize battery usage. However, the correlation between software aging and battery consumption has remained unexplored by researchers.

The state of the art was examined to acquire the essential knowledge regarding the software aging phenomenon, including its causes, primary aging indicators, methods employed for its detection, and the industry's methodologies for measuring battery consumption in mobile devices.

Armed with this knowledge, we devised an experimental process aimed at drawing essential conclusions about software aging impact on battery consumption. This process involved executing Android applications under an accelerated workload, collecting the requisite metrics for identifying signs of aging and assessing battery consumption, and subsequently applying appropriate statistical analyses to the collected data.

Due to the subjective nature of the aging factor, we adopted two distinct approaches to identify this factor. The first approach treated it as an ordinal variable, with each state characterized by the number of aging conditions present. In the second approach, aging was considered a binary variable, with states categorized as either 'aged' (1) or 'not aged' (0) based on the number of detected aging conditions.

None of the results from either approach strongly indicated a clear impact of software aging on battery consumption. When considering aging as a binary factor, approximately half of the experiments that reached an aging state supported by multiple aging conditions showed increased battery consumption from non-aged states to aged states. However, we concluded that this is an insufficient number of experiments to assert that aging was the primary cause of this increase in battery

consumption. Other contributing factors may have played a role in explaining this behavior of the applications.

6.1 Future Work

This dissertation provides the possibility of several future work directions. Firstly, a promising approach would involve addressing the main limitation of this study by expanding the research to encompass a larger data set of applications and/or conducting a higher frequency of experiments for each application.

Moreover, exploring the extension of this research to other platforms, such as iOS, could provide valuable insights into whether the relationship between aging and battery consumption varies.

Lastly, there is an opportunity to extend this research to investigate the impact of software rejuvenation on the battery consumption of mobile devices. This could be achieved by implementing rejuvenation techniques on aged applications and thoroughly analyzing battery consumption throughout the experiment before and after the rejuvenation actions were performed.

References

[1] Mobile operating system market share worldwide | statcounter global stats, . URL https://gs.statcounter.com/os-market-share/mobile/worldwide. (Accessed on 09-03-2023).

[2] Platform architecture | android developers, . URL https://developer.android.com/guide/platform. (Accessed on 09-03-2023).

[3] Profile battery usage with batterystats and battery historian | android developers, . URL https://developer.android.com/topic/performance/power/setup-battery-historian. (Accessed on 09-03-2023).

[4] High voltage power monitor | monsoon solutions | bellevue, . URL https://www.msoon.com/high-voltage-power-monitor. (Accessed on 09-03-2023).

[5] Odroid, . URL https://www.hardkernel.com/. (Accessed on 09-03-2023).

[6] Powertutor, . URL http://ziyang.eecs.umich.edu/projects/powertutor/index.html. (Accessed on 09-03-2023).

[7] Activity | android developers, . URL https://developer.android.com/reference/android/app/Activity/. (Accessed on 09-03-2023).

[8] Memory management in android | medium, . URL https://arsenasatryanit.medium.com/memory-management-in-android-79f899347d9. (Accessed on 09-03-2023).

[9] Power profiles for android | android open source project, . URL https://source.android.com/docs/core/power. (Accessed on 01-16-2023).

[10] Energydashboard, . URL https://github.com/multilanguageservice/energyDashboard. Accessed: 11-01-2023.

[11] Overview of memory management | android developers, . URL https://developer.android.com/topic/performance/memory-overview. (Accessed on 09-03-2023).

[12] Power management | android open source, . URL https://wladimir-tm4pda.github.io/porting/power_management.html. (Accessed on 09-03-2023).

[13] Zuriani Abdullah, Jamaiah Yahaya, Siti Rohana, Sazrol Bojeng, and Aziz Deraman. The implementation of software anti-ageing model towards green and sustainable products. *International Journal of Advanced Computer Science and Applications*, 10, 01 2019. doi: 10.14569/IJACSA.2019.0100507.

[14] Jean Araujo, Vandi Alves, Danilo Oliveira, Pedro Dias, Bruno Silva, and Paulo Maciel. An investigative approach to software aging in android applications. pages 1229–1234, 2013. doi: 10.1109/SMC.2013.213.

[15] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. doi: 10.1109/TDSC.2004.2.

[16] Algirdas Avizienis, Vytautas U, Jean-claude Laprie, and Brian Randell. Fundamental concepts of dependability. 04 2001.

[17] Luis Corral, Anton B. Georgiev, Alberto Sillitti, and Giancarlo Succi. A method for characterizing energy consumption in android smartphones. pages 38–45, 2013. doi: 10.1109/GREENS.2013.6606420.

[18] Domenico Cotroneo, Michael Grottke, Roberto Natella, Roberto Pietrantuono, and Kishor S. Trivedi. Fault triggers in open-source software: An experience report. pages 178–187, 2013. doi: 10.1109/ISSRE.2013.6698917.

[19] Domenico Cotroneo, Francesco Fucci, Antonio Ken Iannillo, Roberto Natella, and Roberto Pietrantuono. Software aging analysis of the android mobile os. pages 478–489, 2016. doi: 10.1109/ISSRE.2016.25.

[20] Domenico Cotroneo, Luigi De Simone, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. A configurable software aging detection and rejuvenation agent for android. pages 239–245, 2019. doi: 10.1109/ISSREW.2019.00078.

[21] Domenico Cotroneo, Antonio Ken Iannillo, Roberto Natella, and Roberto Pietrantuono. A comprehensive study on software aging across android versions and vendors. *Empirical Software Engineering*, 25, 09 2020. doi: 10.1007/s10664-020-09838-3.

[22] Domenico Cotroneo, Luigi De Simone, Roberto Natella, Roberto Pietrantuono, and Stefano Russo. Software micro-rejuvenation for android mobile systems. *Journal of Systems and Software*, 186:111181, 2022. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.111181. URL `https://www.sciencedirect.com/science/article/pii/S0164121221002636`.

[23] Soumya Kanti Datta, Christian Bonnet, and Navid Nikaein. Android power management: Current and future trends. pages 48–53, 2012. doi: 10.1109/ETSIoT.2012.6311253.

[24] Dario Di Nucci, Fabio Palomba, Antonio Prota, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Software-based energy profiling of android apps: Simple, efficient and reliable? pages 103–114, 2017. doi: 10.1109/SANER.2017.7884613.

[25] S. Garg, A. van Moorsel, K. Vaidyanathan, and K.S. Trivedi. A methodology for detection and estimation of software aging. pages 283–292, 1998. doi: 10.1109/ISSRE.1998.730892.

[26] Jim Gray. Why do computers stop and what can be done about it? 1985.

[27] Michael Grottke and Kishor Trivedi. A classification of software faults. 27, 01 2005.

[28] Michael Grottke, Rivalino Matias, and Kishor S. Trivedi. The fundamentals of software aging. pages 1–6, 2008. doi: 10.1109/ISSREW.2008.5355512.

[29] Josh Howarth. Time spent using smartphones (2023 statistics), 2023. URL `https://explodingtopics.com/blog/smartphone-usage-stats`. (Accessed on 09-03-2023).

[30] Y. Huang, C. Kintala, N. Kolettis, and N.D. Fulton. Software rejuvenation: analysis, module and applications. pages 381–390, 1995. doi: 10.1109/FTCS.1995.466961.

[31] Ahmed Hussein, Mathias Payer, Antony Hosking, and Christopher A. Vick. Impact of gc design on power and performance for android. 2015. doi: 10.1145/2757667.2757674. URL `https://doi.org/10.1145/2757667.2757674`.

[32] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. Devscope: A nonintrusive and online power analysis tool for smartphone hardware components. page 353–362, 2012. doi: 10.1145/2380445.2380502. URL `https://doi.org/10.1145/2380445.2380502`.

[33] Lauren. Mobile app download statistics usage statistics (2023), 2022.

[34] Rivalino Matias Jr., Kishor S. Trivedi, and Paulo R.M. Maciel. Using accelerated life tests to estimate time to software aging failure. pages 211–219, 2010. doi: 10.1109/ISSRE.2010.42.

[35] D.L. Parnas. Software aging. pages 279–287, 1994. doi: 10.1109/ICSE.1994.296790.

[36] Abhinav Pathak, Y. Hu, and Ming Zhang. Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. *EuroSys'12 - Proceedings of the EuroSys 2012 Conference*, 04 2012. doi: 10.1145/2168836.2168841.

[37] Yu Qiao, Zheng Zheng, and Fangyun Qin. An empirical study of software aging manifestations in android. pages 84–90, 2016. doi: 10.1109/ISSREW.2016.19.

[38] Yu Qiao, Zheng Zheng, and YunYu Fang. An empirical study on software aging indicators prediction in android mobile. pages 271–277, 2018. doi: 10.1109/ISSREW.2018.00018.

[39] Tajmilur Rahman, Joshua Nwokeji, and Tejas Manjunath. Analysis of current trends in software aging: A literature survey. *Computer and Information Science*, 15:19, 08 2022. doi: 10.5539/cis.v15n4p19.

[40] Rui Rua and João Saraiva. E-manafa: Energy monitoring and analysis tool for android. 2023. doi: 10.1145/3551349.3561342. URL `https://doi.org/10.1145/3551349.3561342`.

[41] Hossain Shahriar, Sarah North, and Edward Mawangi. Testing of memory leak in android applications. pages 176–183, 2014. doi: 10.1109/HASE.2014.32.

[42] Srishti. Average screen time: Statistics (for laptops & smartphones), 2023. URL `https://elitecontentmarketer.com/screen-time-statistics/`. (Accessed on 09-03-2023).

[43] Caisheng Weng, Jianwen Xiang, Shengwu Xiong, Dongdong Zhao, and Chunhui Yang. Analysis of software aging in android. pages 78–83, 2016. doi: 10.1109/ISSREW.2016.20.

[44] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. Appscope: Application energy metering framework for android smartphones using kernel activity monitoring. *USENIX ATC*, 06 2012.

[45] Lide Zhang, Birjodh Tiwana, Robert P. Dick, Zhiyun Qian, Z. Morley Mao, Zhaoguang Wang, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. pages 105–114, 2010.

Appendices

Appendix A

Table with every experiment results

test/folder	process	category	samples	duration (h)
teste_26	com.baidu.searchbox	4	922	7.7
teste_30	com.physphil.android.unitconverterultimate	1	1423	11.9
teste_31	com.simplemobiletools.notes	1	651	5.4
teste_32	com.simplemobiletools.notes	1	1433	11.9
teste_33	net.sourceforge.opencamera	1	1416	11.8
teste_34	com.sina.weibo	3	756	6.3
teste_35	com.reddit.frontpage	3	1170	9.8
teste_36	com.baidu.searchbox	4	1142	9.5
teste_41	notion.id	1	996	8.3
teste_44	com.physphil.android.unitconverterultimate	1	1482	12.4
teste_45	com.free.turbocleaner	1	1238	10.3
teste_46	com.matteljv.uno	2	751	6.3
teste_47	com.scopely.monopolygo	2	812	6.8
teste_48	com.zhiliaoapp.musically	3	1055	8.8
teste_49	com.zhiliaoapp.musically	3	1093	9.1
teste_50	com.android.chrome	4	1072	8.9
teste_51	org.mozilla.firefox	4	951	7.9
teste_52	com.microsoft.emmx	4	1485	12.4
teste_54	beetles.puzzle.solitaire	2	990	8.3
teste_55	com.twitter.android	3	1495	12.5
teste_56	com.facebook.lite	3	1499	12.5
teste_57	notion.id	1	1207	10.1
teste_58	com.reddit.frontpage	3	885	7.4
teste_59	com.twitter.android	3	1057	8.8
teste_60	com.facebook.lite	3	1499	12.5
teste_61	com.android.chrome	4	1234	10.3
teste_62	com.microsoft.emmx	4	1196	10
teste_63	org.mozilla.firefox	4	793	6.6
teste_64	com.matteljv.uno	2	846	7.1
teste_66	com.king.candycrushsaga	2	953	7.9
teste_67	net.sourceforge.opencamera	1	1499	12.5
teste_68	com.scopely.monopolygo	2	954	8
teste_69	com.king.candycrushsaga	2	737	6.1
teste_70	com.brave.browser	4	794	6.6
teste_71	sonic.bubbleshoot.classic	2	679	5.7
teste_72	sonic.bubbleshoot.classic	2	728	6.1

	Spearman correlation (V0)		Wilcoxon Rank Sum (V1)	
test/folder	statistic	p-value	statistic	p-value
teste_26	-0.054	0.150	-1.371	0.915
teste_30	0.160	0.000	5.060	0.000
teste_31	0.231	0.000	4.996	0.000
teste_32	0.150	0.000	5.035	0.000
teste_33	0.238	0.000	7.203	0.000
teste_34	-0.050	0.257	-1.806	0.965
teste_35	0.234	0.000	6.998	0.000
teste_36	-0.212	0.000	-6.616	1.000
teste_41	-0.012	0.723	-0.489	0.688
teste_44	-0.026	0.370	-0.875	0.809
teste_45	-0.104	0.000	-3.568	1.000
teste_46	0.011	0.806	0.246	0.403
teste_47	0.059	0.124	1.540	0.062
teste_48	-0.015	0.676	-0.529	0.702
teste_49	-0.021	0.557	-0.587	0.722
teste_50	-0.076	0.020	-2.322	0.990
teste_51	0.153	0.000	4.121	0.000
teste_52	-0.024	0.408	-0.950	0.829
teste_54	-0.017	0.606	-0.516	0.697
teste_55	-0.084	0.002	-3.048	0.999
teste_56	0.094	0.001	3.239	0.001
teste_57	0.093	0.003	2.978	0.001
teste_58	-0.195	0.000	-5.034	1.000
teste_59	-0.041	0.258	-1.140	0.873
teste_60	-0.185	0.000	-6.414	1.000
teste_61	-0.074	0.013	-2.472	0.993
teste_62	-0.092	0.004	-2.874	0.998
teste_63	0.451	0.000	11.026	0.000
teste_64	-0.085	0.022	-2.296	0.989
teste_66	-0.237	0.000	-6.391	1.000
teste_67	0.035	0.240	1.174	0.120
teste_68	-0.021	0.555	-0.591	0.723
teste_69	-0.125	0.003	-2.936	0.998
teste_70	-0.224	0.000	-5.786	1.000
teste_71	0.027	0.533	0.624	0.266
teste_72	0.027	0.525	0.637	0.262

	Wilcoxon Rank Sum (V2)		Wilcoxon Rank Sum (V3)	
test/folder	statistic	p-value	statistic	p-value
teste_26	-1.317	0.906	0.336	0.368
teste_30				
teste_31	3.469	0		
teste_32	0.758	0.224		
teste_33				
teste_34	4.038	0	3.859	0
teste_35	5.137	0	3.01	0.001
teste_36	2.722	0.003	1.588	0.056
teste_41	4.435	0	3.028	0.001
teste_44	-1.716	0.957		
teste_45				
teste_46				
teste_47				
teste_48	1.924	0.027		
teste_49				
teste_50				
teste_51				
teste_52	2.185	0.014		
teste_54				
teste_55	-0.892	0.814	-0.892	0.814
teste_56	4.972	0	1.861	0.031
teste_57	-1.685	0.954		
teste_58				
teste_59	-0.053	0.521	-0.038	0.515
teste_60	-1.349	0.911		
teste_61				
teste_62	0.449	0.327		
teste_63				
teste_64				
teste_66				
teste_67				
teste_68				
teste_69				
teste_70				
teste_71				
teste_72				

test/folder	average consumption b0 (mAh)	average consumption b1 (mAh)
teste_26	1.78	1.94
teste_30	2.25	2.96
teste_31	1.78	2.13
teste_32	1.87	2.08
teste_33	1.8	2.23
teste_34	4.26	5.85
teste_35	4.36	7.17
teste_36	3.32	3.88
teste_41	3.45	4.76
teste_44	2.38	1.7
teste_45	3.03	2.52
teste_46	4.31	4.57
teste_47	3.96	4.25
teste_48	3.32	3.91
teste_49	3.31	3.25
teste_50	2.97	2.75
teste_51	3.39	3.89
teste_52	2.23	2.36
teste_54	2.98	2.94
teste_55	2.53	2.2
teste_56	2.36	3.03
teste_57	2.84	2.13
teste_58	2.29	2.01
teste_59	3.96	4.03
teste_60	2.27	1.04
teste_61	2.08	2.02
teste_62	2.72	3.32
teste_63	4.06	4.59
teste_64	3.17	2.96
teste_66	3.18	3
teste_67	2.31	2.38
teste_68	2.88	2.83
teste_69	3.29	3.13
teste_70	2.87	2.59
teste_71	5.25	5.28
teste_72	5.31	5.33