



UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

PH.D. THESIS

IN

INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

DEPENDABILITY ASSESSMENT OF ANDROID OS

ANTONIO KEN IANNILLO

TUTOR: PROF. DOMENICO COTRONEO

XXX CICLO

SCUOLA POLITECNICA E DELLE SCIENZE DI BASE

DIPARTIMENTO DI INGEGNERIA ELETTRICA E TECNOLOGIE DELL'INFORMAZIONE

UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II

DOCTORAL THESIS

Dependability Assessment of Android OS

Author:

Antonio Ken IANNILLO

Supervisor:

Prof.Domenico COTRONEO

*A thesis submitted in fulfillment of the requirements
for the degree of Doctor of Philosophy in*

Information Technology and Electrical Engineering

Scuola Politecnica e delle Scienza di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

*benedicat tibi Dominus et custodiat te
ostendat Dominus faciem suam tibi et misereatur tui
convertat Dominus vultum suum ad te et det tibi pacem*

UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

Abstract

Scuola Politecnica e delle Scienza di Base
Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Doctor of Philosophy

Dependability Assessment of Android OS

by Antonio Ken IANNILLO

In this *brave new world* of smartphone-dependent society, dependability is a strong requirement and needs to be addressed properly. Assessing the dependability of these mobile system is still an open issue, and companies should have the tools to improve their devices and beat the competition against other vendors.

The main objective of this dissertation is to provide the methods to assess the dependability of mobile OS, fundamental for further improvements.

Mobile OS are threatened mainly by traditional residual faults (when errors spread across components as failures), aging-related faults (when errors accumulate over time), and misuses by users and applications. This thesis faces these three aspects. First, it presents a qualitative method to define the fault model of a mobile OS, and an exhaustive fault model for Android. I designed and developed *AndroFIT*, a novel fault injection tool for Android smartphone, and performed an extensive fault injection campaign on three Android devices from different vendors to analyze the impact of component failure on the mobile OS. Second, it presents an experimental methodology to analyze the software aging phenomenon in mobile OS. I performed a software aging analysis campaign on Android devices to identify the impacting factors on performance degradation and resource consumption. Third, it presents the design and implementation of a novel fuzzing tool, namely *Chizpurfle*, able to automatically test Android vendor customizations by leveraging code coverage information at run-time.

Acknowledgements

I'd like to thank my advisor prof. Domenico Cotroneo.

I'd like to thank Luigi De Simone, Francesco Fucci, Anna Lanzaro, Roberto Natella, prof. Cristina Nita-Rotaru, Roberto Pietrantuono, Stefano Rosiello, prof. Stefano Russo, and all the colleagues and friends that contributed to my doctoral course.

I'd like to thank my DESSERT labmates, the PhD students from itee XXX, the employees of CRITIWARE, and all the colleagues and friends of the Department of Electrical Engineering and Information Technology (DIETI) at University of Naples Federico II.

I'd like to thank prof. Cristina Nita-Rotaru and all the colleagues and friends of the College of Computer and Information Science (CCIS) at Northeastern University.

I'd like to thank Ole André Vadla Ravnås and all the FRIDA community.

I'd like to thank my Bostonian family, my Franciscan fraternity, and my family by blood.

I'd like to thank my true friends and my beloved one.

I'd like to thank you who are going to read my thesis.

Contents

Abstract	v
Acknowledgements	vii
1 Introduction	1
1.1 The Need for Dependable Smartphones	2
1.2 Dependability Threats and Assessment	3
1.3 Thesis Contributions	5
1.3.1 Fault Injection Testing	5
1.3.2 Software Aging Analysis	6
1.3.3 Fuzz Testing	7
2 State of the Art in Mobile System Dependability	11
2.1 Fault Injection Testing	12
2.2 Software Aging and Rejuvenation	17
2.3 Fuzz Testing	18
3 AndroFIT: A Software Fault Injection Approach for the Android Mobile OS	23
3.1 Overview	24
3.2 Fault Modeling	26
3.2.1 Methodology	26
3.2.2 Android Fault Model	32
3.3 Android Fault Injection Tool (AndroFIT)	45
3.3.1 Fault Injection Techniques	45
3.3.2 Design and Implementation of AndroFIT	52
3.4 Experimental Evaluation	60
3.4.1 Fault Injection in the Phone Subsystem	61
3.4.2 Fault Injection in the Camera Subsystem	63

3.4.3	Fault Injection in the Sensors Subsystem	65
3.4.4	Fault Injection in the Activity Subsystem	68
3.4.5	Fault Injection in the Package Subsystem	71
3.4.6	Fault Injection in the Storage Subsystem	72
3.4.7	Lessons Learned	74
4	Software Aging Analysis of the Android Mobile OS	79
4.1	Overview	80
4.2	Experimental Methodology	81
4.2.1	User-Perceived Response Variable	82
4.2.2	System-Related Response Variables	84
4.2.3	Factors and Levels	87
4.2.4	Experimental plan	90
4.3	Results	93
4.3.1	Software aging across Android vendors	93
4.3.2	Software aging across Android versions	98
4.3.3	Analysis of process internals	101
5	Chizpurple: A Gray-Box Android Fuzzer for Vendor Service Customizations	107
5.1	Overview	108
5.2	Chizpurple	109
5.2.1	Motivations	110
5.2.2	Design	111
5.3	Experimental Evaluation	124
5.3.1	Bugs in Samsung Customizations	124
5.3.2	Comparison with Black-Box Fuzzing	128
6	Conclusion And Future Directions	135
6.1	Fault Injection Testing	135
6.2	Software Aging Analysis	136
6.3	Fuzz Testing	138
6.4	Further Discussion	138
A	Android Insights	141
A.1	Android Architecture	141
A.2	Binder IPC	145

A.3 Service Manager 147

B Android Fault Model 149

References 190

List of Figures

2.1	Fault Injection Testing General Schema (Figure 1 of Hsueh <i>et al.</i> [1])	13
2.2	PAIN architecture (Figure 1 of Winter <i>et al.</i> [2])	14
2.3	Fault Injection Approach in modified QEMU architecture (Figure 2 of Ferraretto <i>et al.</i> [3])	16
2.4	Intent Fuzzer Architecture (Figure 2 of Sasnauskas <i>et al.</i> [4])	21
3.1	a Fault-Error-Failure Propagation Chain in Android	25
3.2	a Software Component Model View	28
3.3	Architecture of the Android Phone Subsystem	37
3.4	Architecture of the Android Camera Subsystem	39
3.5	Architecture of the Android Sensors Subsystem	41
3.6	Architecture of the Android Storage Subsystem	44
3.7	Binder IPC Hijacking Fault Injection Technique on Transaction Messages	47
3.8	Binder IPC Hijacking Fault Injection Technique on Reply Messages	48
3.9	Library Hooking Fault Injection Technique	49
3.10	System Call Hooking Fault Injection Technique	50
3.11	Unix Socket Hijacking Fault Injection Technique	51
3.12	Unix Signaling Fault Injection Technique	52
3.13	AndroFIT Architecture	53
3.14	Execution of the Fault Injection Experiments	56
3.15	Flow of a Fault Injection Experiment	57
3.16	Output Folder Structure and Files of the Experiment Launcher	58
3.17	Fault Injection Campaign Outcomes for the Phone Subsystem	62
3.18	Analysis of the Failure Scenario #1	63
3.19	Fault Injection Campaign Outcomes for the Camera Subsystem	64

3.20	Analysis of the Failure Scenario #2	64
3.21	Analysis of the Failure Scenario #3	66
3.22	Fault Injection Campaign Outcomes for the Phone Subsystem	67
3.23	Analysis of the Failure Scenario #4	68
3.24	Fault Injection Campaign Outcomes for the Activity Subsystem	69
3.25	Fault Injection Campaign Outcomes for the Package Subsystem	71
3.26	Fault Injection Campaign Outcomes for the Storage Subsystem	72
4.1	The Experimental Android Testbed	93
4.2	Groups Activities Launch Time for <i>EXP39</i>	94
4.3	Distribution of the Launch Time Trends, with all vendors and fixed to Android 6 (<i>EXP13~EXP60</i>)	95
4.4	PSS Trends Distributions: <i>EXP13~EXP60</i> (Android 6)	97
4.5	Launch Time Trends Distributions: <i>EXP49~EXP72</i> (Samsung S6 Edge)	99
4.6	Launch Time Trends Distributions: <i>EXP1~EXP24</i> (Huawei P8)	100
4.7	PSS Trends Distributions: <i>EXP1~EXP24</i> (Huawei P8)	102
4.8	PSS Trends Distributions: <i>EXP49~EXP72</i> (Samsung S6 Edge)	103
4.9	Occurrences of GC metric trend: <i>EXP1~EXP72</i>	104
4.10	Occurrences of task metric trend: <i>EXP1~EXP72</i>	106
5.1	AOSP and Vendor services.	112
5.2	Overview of the Architecture of Chizpurfle	112
5.3	Chizpurfle Instrumentation and Tracing Mechanism	117
5.4	Performance Overhead of Chizpurfle	130
5.5	Code Coverage Gain of Chizpurfle	131
5.6	Code Coverage Gain of Chizpurfle per Method	133
A.1	Android System Architecture	143
A.2	Binder IPC Iteration Between Two Android Processes	146
A.3	Android Services and Service Manager	147

List of Tables

3.1	A Comparison of Failure Classifications [5]	29
3.2	Summary of the Android Fault Model	34
3.3	Fault Injection Techniques and Target Components Map . .	77
3.4	Summary of the Fault Injection Campaign Outcomes	78
4.1	Factors and Levels for Android Software Aging Analysis . .	88
4.2	Experimental plan of the case study	90
4.3	Spearman Correlation Coefficients between All Activities LT Trends and PSS Trends of Android System Processes	98
5.1	Vendors' Smartphone Customizations on System Services .	111
5.2	Failures Detected by Chizpurple	125
B.1	RILD Fault Model	151
B.2	Baseband Driver and Processor Fault Model	153
B.3	Camera Service Fault Model	155
B.4	Camera HAL Fault Model	165
B.5	Camera Driver and Hardware Fault Model	170
B.6	Sensor Service and HAL Fault Model	172
B.7	Sensors Drivers and Devices Fault Model	174
B.8	Activity Manager Service Fault Model	175
B.9	Package Manager Service Fault Model	178
B.10	SQLite Library Fault Model	180
B.11	Bionic Library Fault Model	181
B.12	Mount Service Fault Model	182
B.13	Volume Daemon Fault Model	183
B.14	Storage Drivers and Hardware Fault Model	186

List of Abbreviations

ADB	A ndroid D ebug B ridge
AFL	A merican F uzzy L op
AIDL	A ndroid I nterface D escription L anguage
AndroFIT	A ndroid F ault I njection T ool
AOSP	A ndroid O pen S ource P roject
API	A pplication P rogramming I nterface
app	(mobile) a pplication
ARB	A ging R elated B ug
ARM	A dvanced R ISC M achine
ART	A ndroid R un- T ime
BYOD	B ring Y our O wn D evice
CPU	C entral P rocessing U nit
CUT	C omponent U nder T est
cfr.	c onfronta (compare)
DoE	D esign of E xperiment
<i>e.g.,</i>	<i>exempli grātiā</i> , (for example,)
FTMA	F ault T olerant M echanisms (and) A lgorithms
GC	G arbage C ollection
GPS	G lobal P ositioning S ystem
HAL	H ardware A bstraction L ayer
HIDL	H AL I nterface D escription L anguage
<i>i.e.,</i>	<i>id est</i> , (that is,)
I/O	I nterface O utput
IoT	I nternet of T hings
ICC	I nter- C omponent C ommunication
IPC	I nter- P rocess C ommunication
KSM	K ernel S amepage M erging
LLVM	L ow L evel V irtual M achine
MIME	M ultipurpose I nternet M ail E xtensions

MK	Mann-Kendall
MVDP	Mobile Vulnerability Discovery Pipeline
MuT	Module under Test
NFC	Near Field Communication
OEM	Original Equipment Manufacturers
OS	Operating System
PSS	Proportional Set Size
RDS	Radio Data System
RISC	Reduced Instruction Set Computing
SIR	Service Interfaces (and) Resources
SMS	Short Message Service
SNMP	Simple Network Management Protocol
TTE	Time-To-Exhaustion
UI	User Interface

to the Dreams

Chapter 1

Introduction

If you want to get someone's attention, show you can help.

— John C. Maxwell

This thesis deals with the dependability assessment of Android-based mobile systems. The main objective is to provide novel methods and experimental procedures to assess the dependability of mobile OS, specifically Android OS, fundamental for further improvements. The contributions of this thesis are:

- a qualitative method to define the fault model of a mobile OS, and an exhaustive fault model for Android;
- the design and implementation of *AndroFIT*, a novel fault injection tool for Android smartphones;
- an extensive fault injection campaign on three Android devices from different vendors to analyze the impact of component failures on the mobile OS;
- an experimental methodology to analyze the software aging phenomenon in mobile OS;
- a software aging analysis campaign on Android devices to identify the impacting factors on performance degradation and resource consumption;
- the design and implementation of a novel fuzzing tool, namely *Chizpurfle*, able to automatically test Android vendor customizations by leveraging code coverage information at run-time.

1.1 The Need for Dependable Smartphones

The rapid and continuous evolution of information and communication technologies brought modern society to constantly interact with personal and portable computers. Gone are those days when mobile phones served as a device to make calls and occasionally send text. Now, mobile phones hold more of one's life than computers do. Smartphones will be, and partially already are, the most critical resource for the interaction among the physical and digital world. They provide access, through apps, to every kind of service: mail, data storage, telephony, information provisioning, data sharing, e-commerce, banking, and social-networking are only few examples. In the very next future, they will become digital wallets and holders of digital identity. Companies are already surrounded by a computing ecosystems with mobile devices that earn access to sensitive services and data, applying the so-called Bring Your Own Device (BYOD) paradigm. Furthermore, mobile devices can also communicate with other networked devices, playing a central role in the Internet of Things (IoT).

As mobile devices become more and more deeply embedded in business and personal contexts, the most important challenge is ensuring that a user can trust them. If users find that a device is unreliable or insecure, they will refuse to use it. Furthermore, they may also refuse to buy or use products from the same vendor, because they may believe that these products are also likely to be unreliable or insecure. "Poor quality of software can result in serious damage to the brand value of an organization and often incurs huge repair costs" [6]. The World Quality Report 2017–2018 confirms that the 1660 executives in 32 countries, involved in the analysis, are becoming aware of the importance of any failure experienced by end-users, who spread this information in a viral way on social media and can cause financial loss.

Users cannot afford any failure that could potentially affect and damage the way they relate to the world.

Companies cannot afford any failure that could certainly affect and damage the financial capital they own.

Smartphones must be dependable.

"The dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable." [7]

The mobile operating system (mobile OS) plays a crucial role since it allows smartphones, tablets and other mobile devices to run applications. Mobile OS is responsible to manage physical resources and abstract them for applications as every OS, but they also address the peculiarities of mobile devices: limited memory and battery, small display and touchscreen, and heterogeneous resources and protocols such as cellular, Bluetooth, Wi-Fi, Global Position System (GPS) navigation, built-in cameras, Near Field Communication (NFC), gyroscope, touch screens, and Radio Data System (RDS) receivers. Managing all these sensors and actuators with the constraints of an embedded device is not a trivial task and threatens the dependability of mobile devices.

In this scenario, the most influential mobile OS providers are Google, with Android, and Apple, with iOS. Android dominates the market with a 86.8% share, against the 12.5% of iOS [8]. While iOS is a closed-source mobile OS delivered only in Apple iPhones, Android is an open-source project and comes in different flavors, depending on which vendor is implementing it. Nowadays, more than 20 original equipment manufacturers (OEMs), including but not limited to Samsung, Huawei, OPPO, and LG, base their devices on the Android Open Source Project (AOSP) [9]. One of the main target of these companies is to provide a better device than their competitors, and *better* means also *more dependable* in order to secure the customer loyalty earned with value-added services. Unfortunately, the World Quality Report 2017-2018 [6] stated that most of the world-wide companies declared the challenges in testing mobile applications, devices, and wearable includes the lack of the right testing processes, methods and tools.

This thesis faces the dependability analysis of mobile OS, with a case study based on the complex and fragmented Android ecosystem.

1.2 Dependability Threats and Assessment

The basic concepts of dependability are well-defined by Avizenis *et al.* [7], formalizing the *fault-error-failure* chain and the fault tolerance.

There are mainly two categories of faults that need to be considered as potential causes of mobile OS failures. They are

- **Residual faults of the mobile OS:** they are hardware or software defects within the components of the mobile OS (also known as internal

A **failure** is an event that occurs when a system does not deliver the service as expected by its users (*e.g.*, the mobile OS crashes and the device can not be used);

An **error** is an erroneous internal state that propagates within the system and eventually turns into a failure (*e.g.*, a mobile OS internal service has a missing event handler);

A **fault** is an attribute of the system that leads to an error (*e.g.*, a missing event handler initialization instruction in the mobile OS code);

Fault Tolerance is a mean to obtain a dependable system by avoiding service failures in the presence of faults, carried out via Failure Tolerance Mechanisms and Algorithms (FTMA) (*e.g.*, an exception handler that shows an error message to the user and keeps the mobile OS running with reduced functionalities).

faults) that, under special conditions (*i.e.*, triggers), leads to an internal error state. According to their propagation, they can be further divided in

- traditional faults, when the errors, not correctly handled by FTMA, spread across other components in the mobile OS as component failures; or
- aging faults, when the errors accumulate over time causing performance degradation and poor quality of service.
- **Misuses of the mobile OS:** they are the misuses of the mobile device system by users and applications. They are external faults, *e.g.*, inconsistent inputs, that originate from the users of the system, including human users that interact with the device and applications that interact with the mobile OS framework.

These threats undermine the smartphone dependability as perceived by the users. The impact of these faults may consist of unresponsiveness to user's input, not-working conditions, or unauthorized actions, among others.

Dependability assessment of a mobile OS must face these threats. It should primarily test the FTMA, quantifying the impact of traditional faults on the system. Nevertheless, dependability assessment should also test the quality of service of a mobile OS, focusing on its performance during the

long-lasting activity and analyzing the effect of the aging faults. Finally, since the Android OS allows vendors to add custom interfaces, dependability assessment should pay particular attention to them and test them efficiently against misuses.

1.3 Thesis Contributions

This thesis revolves around three aspects: fault injection testing, aging analysis, and fuzz testing. Regarding the first aspect, mobile OS could be statically or dynamically analyzed to promote a comprehensive **fault injection** approach, which intentionally injects realistic faults into the mobile OS components to understand how the FTMA and the whole system react to them. With regards to the second aspect, since the performance degradation or **aging phenomenon** of mobile OS is not a direct consequence of faults that can be arbitrarily injected but the result of errors accumulation, an approach to analyze the impact of long running operational periods on the mobile OS performances is presented. Third and last, Android vendors introduce closed-source software customizations on their products exposed as interfaces, and a novel gray-box **fuzzing** approach can be used to analyze their robustness by exploiting run-time information. These three aspects are carefully developed in this thesis work to analyze the dependability of mobile OS.

1.3.1 Fault Injection Testing

Testing is a software development phase of paramount importance, and it is also the most costly one. Nevertheless, software comes with residual faults that need to be tolerated by the system [10]. Failure tolerance mechanisms and algorithms (FTMA) should satisfy the requirement to obtain a dependable system by avoiding service failure in presence of faults. Any failure that is not handled by the system may undermine the user experience (UX), and damage both the user and the vendor.

Fault injection is the process of introducing faults in a system, with the goal of assessing the impact of faults on performance and on availability, and the effectiveness of fault tolerance mechanisms. It is important to clearly divide the mobile OS architectures in two sets of components, such as:

- Fault injection targets: the components in which we expect that faults occur;
- Components under test: the components that should be able to handle or tolerate faults.

The main challenge with fault injection is to define a **fault model**, *i.e.*, a set of realistic component failures that could be injected in the fault injection targets and act as a fault for the mobile OS. In order to define a general and exhaustive fault model, this thesis proposes a simple but effective methodology (SIR methodology), that consists of:

1. analysis of the target architecture, identifying the services provided by the component and the resources managed by the component;
2. application of defined failure modes to every service and resource;
3. assignment of fault persistence.

I performed this procedure on the Android OS to extract a comprehensive fault model. Based on this model, I developed the Android Fault Injection Tool (**AndroFIT**) and performed a fault injection campaign on three popular Android smartphones to analyze how different vendor smartphones react to faults. The campaign injected more than 700 faults related to 6 different subsystems (*i.e.*, phone, camera, sensors, activity, package, and storage subsystems), executing 2196 experiments in total, where each experiment lasts about 5 minutes. The results show the effectiveness of the fault injection approach and how vendors still need to improve the dependability of their devices, even if they react differently to the same failures.

1.3.2 Software Aging Analysis

With regard to the requested responsiveness of mobile devices, this thesis faces the problem of the **software aging** phenomenon in mobile OS. Software aging can cause the device to slowly degrade its performance and to eventually fail, due to the accumulation of errors in the system state and to the incremental consumption of resources, such as physical memory. Software aging can be attributed to software faults that manifest themselves as memory leakage and fragmentation, unreleased locks, stale threads, data

corruption, and numerical error accumulation. Analyzing the public bug repository of Android, there are evidence that these bugs affect the Android OS, thus exposing commercial Android devices on the market to software aging issues.

This thesis presents an **experimental methodology** to analyze software aging issues in the Android OS, but it can be easily generalized to other mobile OS. The procedure consists of statistical methods and techniques to identify which factors (such as workloads and device configurations) exacerbate performance degradation and resource consumption. Moreover, it analyzes the correlation between software aging and resource utilization metrics, in order to pinpoint which subsystems are affected by aging and to support the design of software rejuvenation strategies.

I applied this procedure for an extensive empirical analysis of software aging in 4 recent Android devices (*i.e.*, Samsung Galaxy S6 Edge, Huawei P8, HTC One M9, and LG Nexus) running Android 5 (Lollipop), Android 6 (Marshmallow), and Android 7 (Nougat). In details, the experimental plans is based on 5 different factor, counting from 2 to 4 levels, resulting in 72 experiment. Each experiment lasts about 6 hours, for a total of more than 400 hours of testing time. The analysis of the experimental outcomes, presented in this thesis, pointed out that Android devices are indeed affected by software aging, among with other useful insights.

1.3.3 Fuzz Testing

Companies does not include only new hardware on mobile devices, but they realized that the difference they can make on the market is with new software. The Nokia failure case study clearly shows how a huge phone company, proficient at providing the best hardware, failed also because it defers realizing the dramatic change of focus from hardware to software [11]. However, Vendor software customizations introduce new software defects, which are vendor-specific. Because they are proprietary, vendor customizations are not integrated in the open-source Android and do not benefit from the feedback loop of the whole ecosystem. Thus, they are less scrutinized than the core AOSP codebase, and their vulnerabilities take significantly more time to be patched. It is worth noting that vendors' customizations are code running with special privileges, thus exacerbating

the security issues¹. Misuses of these peculiar interfaces may lead to severe failures and malicious attacks.

Fuzzing is a well-established and effective software testing technique to identify weaknesses in fragile software interfaces by injecting invalid and unexpected inputs. Fuzzing was initially conceived as a “black-box” testing technique, using random or grammar-driven inputs. More recently, “white-box” techniques have been leveraging information about the program internals (such as the test coverage) to steer the generation of fuzz inputs, either by instrumenting the source code or by running the target code in a virtual machine. The visibility of the test coverage has dramatically improved the effectiveness of fuzzing tools, as showed by the high number of subtle vulnerabilities found in many large software systems. Unfortunately, these tools are not applicable to proprietary Android services, since vendors are not willing to share their source code, and since virtual machine environments (*e.g.*, device emulators) do not support the execution of these proprietary extensions.

Thus, I developed **Chizpurfle**, a tool to address the gap in the spectrum of mobile fuzzers, and to improve the effectiveness of fuzzing on vendor customizations. Similarly to recent white-box fuzz approaches, Chizpurfle leverages test coverage information, while avoiding the need for recompiling the target code, or executing it in a special environment. The tool has been designed to be deployed and to run on unmodified Android devices, including any vendor customization to the Android OS. The tool leverages a combination of dynamic binary instrumentation techniques (such as software breakpoints and just-in-time code rewriting) to obtain information about the block coverage. Moreover, Chizpurfle is able to guide fuzz testing only on the vendor customizations, by automatically extracting the list of vendor service interfaces on the Android device. The tool also provides a platform for experimenting with fuzz testing techniques (such as evolutionary algorithms) based on coverage-based feedback.

I validated the applicability and performance of the Chizpurfle tool by conducting a fuzz testing campaign on the vendor customizations of the Samsung Galaxy S6 Edge, running Android version 7. Chizpurfle detected 2,272 service methods from Samsung customizations and performed 34,645

¹For example, some devices based on Qualcomm chipsets suffer from a vulnerability in the Qualcomm service API that allows privilege escalation and information disclosure [12].

tests on these methods, with an average of 7 seconds per test. Chizpurfle improves the depth of testing compared to the black-box approach, by increasing the test coverage by 2.3 times on average and 7.9 times in the best case, with a performance overhead that is comparable to existing dynamic binary instrumentation frameworks. Moreover, we discuss some vulnerabilities found in privileged services during these evaluation experiments.

Chapter 2

State of the Art in Mobile System Dependability

Those who cannot remember the past are condemned to repeat it.

— George Santayana

Since modern mobile systems showed up about ten years ago, current software dependability studies on them are very few and still represent a niche in the research community.

- Fault injection studies focuses on either the lower layers of the mobile systems [2, 3], or the Java applications [13, 14] that could be applied to the Android Java layer (see Section A.1). No work deeply analyzes the behavior of the whole mobile OS in presence of faults in one of its components.
- Software aging has been repeatedly reported both by scientific literature and by software practitioners [15–29], and it has been recognized as a chronic problem in many long-running software systems. Research on software aging in mobile devices is still at an early stage, focusing only on Android applications [30–34] and not on lower layers.
- Fuzz testing has been extensively adopted for testing several software systems as both black-box [35–38] and white-box [39, 40] approaches, in Android [4, 41–46] and other mobile systems [47–49]. Nevertheless, little work was done on the system service of the Android OS [50, 51].

Most of the current software dependability analysis approaches are not for mobile environments. Since modern mobile systems showed up about ten years ago, dependability studies on them are few and still represent a niche in the research community.

This chapter presents the state-of-the-art of the dependability of mobile systems, according to the three main contributions of this dissertation: fault injection testing, fuzz testing, and software aging.

2.1 Fault Injection Testing

Fault injection testing is a software testing technique that consists of deliberately introducing faults in a system, with the goal of assessing the impact of faults on performance and on availability, and the effectiveness of fault tolerance mechanisms. A fault model formalizes those faults that will eventually affect the system during operation. Then, these faults are injected into specific software components of the target system, while it is exercised with a workload. Internal faults can be either hardware or software faults, but they both can be emulated through software and referred as software fault injection.

Several approaches and tools exist to emulate internal faults¹, but all of them fit in the same conceptual schema [1], as shown in Figure 2.1. The system under analysis is usually named **target**. There are two entities that stimulate the system, respectively the **load generator** and the **injector**. The former exercises the target with inputs that will be processed during a fault injection experiment, whereas the latter introduces a fault in the system. The set of inputs and faults submitted to the system are respectively referred to as workload and faultload, which are typically specified by the tester through a library by enumerating inputs and faults or by specifying the rules for generating them. A fault is injected by altering the state of the system or the environment in which it executes. Fault injection usually involves the execution of several experiments or runs, which form a fault injection campaign, and only one or few faults from the faultload are injected during each experiment. The **monitor** collects from the target raw data (readouts or measurements) that are needed to evaluate the effects of injected faults. The choice of readouts depends on the kind of system considered and

¹Natella *et al.* [52] presented an exhaustive survey on software fault injection approaches.

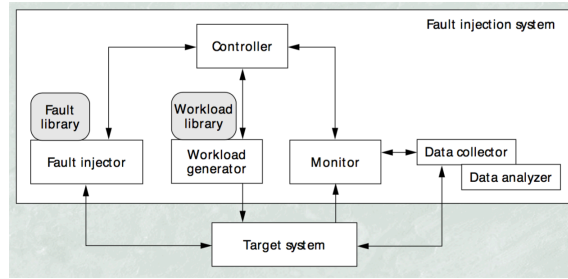


FIGURE 2.1: Fault Injection Testing General Schema (Figure 1 of Hsueh *et al.* [1])

on the properties that have to be evaluated. Measurement may include the outputs of the target (*e.g.*, messages sent to users or to other systems) and the internal state of the target (*e.g.*, the contents of a specific variable in memory). Readouts are used to assess the outcome of the experiment (*e.g.*, the tester can check whether the injected fault has been tolerated, or the severity of the system failure). In order to obtain information about the outcome of an experiment, readouts are usually compared to the readouts obtained from fault-free experiments (referred to as golden runs or fault-free runs). All the described entities are orchestrated by the controller, which is also responsible for iterating fault injection experiments forming the fault injection campaign as well as for storing the results of each experiment to be used for subsequent analysis.

Initially, in a fault injection test, the system is assumed to work in the correct state. As soon as a fault is injected and a workload is applied, two behaviors can be observed. First, the fault is not activated and it remains latent. In this case, after a timeout the experiment terminates and no failure is produced. Second, the fault is activated and it becomes an error. At this stage, an error may propagate, by corrupting other parts of the system state until the system exhibits a failure; can be latent in the system; or can be masked by FTMA. On the basis on the collected readouts, the monitor should be able to identify all these cases.

PAIN [2] is a framework for the parallel execution of fault injection experiments, in order to reduce the time required for fault injection testing. PAIN has been applied to perform fault injection in Android. The system is executed within the Android emulator [53]. Several instances of

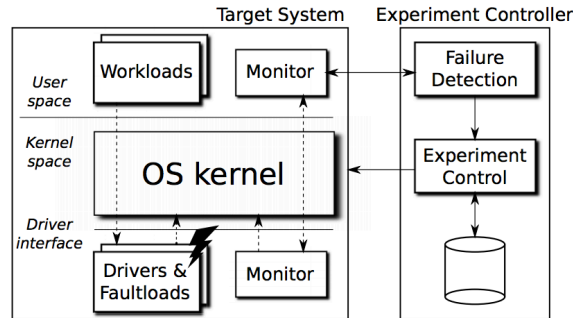


FIGURE 2.2: PAIN architecture (Figure 1 of Winter *et al.* [2])

the emulator are spawn, and a different fault injection test is executed on each instance. The study showed that parallel experiments can achieve a significant speed-up, and at the same time, it can guarantee accurate results. PAIN has adopted the **SAFE** fault injection tool [54]. The SAFE tool injects bugs into a software component, by mutating its source code. The SAFE tool supports the injection of the most typical software faults that have been defined using bug data from both commercial and open-source software. In particular, the tool has been used to inject bugs into device drivers of the Linux kernel (such as the driver of the SSD storage). The experimental setup of the PAIN framework (Figure 2.2) is based on the Android emulator, which executes the Android OS (including device drivers) and a workload. The workload runs the Roy Longbottom's Android benchmarks apps [55] to stimulate the Android OS. Moreover, there are failure monitoring agents, that run both inside and outside the Android emulator. These agents monitor the Android emulator and analyze the effect of the fault on the Android system, detecting failures such as system crashes, system errors, workload failures, system initialization hangs, system execution hangs, and workload hangs. For each experiment, a fault is injected into the device driver, by mutating its code using the SAFE tool, and by uploading the faulty driver on the Android emulator. The Android emulator is rebooted, and the workload and the failure monitors are executed. When the failure monitors detect a failure, this information is recorded into a database for later analysis.

Ferraretto *et al.* [3] presented a **QEMU-based fault injection** approach in order to assess the tolerance of embedded software against faults in CPU

registers. The injection emulates faults into CPU components (*e.g.*, the ALU and the bus). The stressed CPUs are ARM and x86 architectures, and most of the smart devices have an ARM processor in their system-on-chip boards. This approach can be slightly modified and applied to these architectures as well. The approach emulates faults by corrupting the contents of CPU registers. The CPU registers injected with faults are: the instruction register (IR), the program status register (PSR), and the general purpose registers (GPRs). The authors use three well-known fault models in order to corrupt the state of the registers, that are:

- stuck-at fault model: it consists of permanent faults where a bit of a register can stuck at the logic value 0 (stuck-at-0) or at the logic value 1 (stuck-at-1);
- transition fault model: a fault in this category is persistent and it may cause a delay in the switching activity of the affected bit such that the transition of the bit cannot be completed in time to guarantee the next instruction read its updated value. There is a slow-to-rise (slow-to-fall) fault when a bit have to pass from logic value 0 (1) to logic value 1 (0);
- bit flip fault model: this model can switch a bit in a register in a either intermittent or transient flavour.

The whole approach is based on a modified QEMU environment, as shown in Figure 2.3, to pursue the emulation of faults into the CPU registers. In order to inject faults in the IR, the authors modified the fetching mechanism of QEMU to map a different instruction on the instruction sequence for the host machine. For the PSR and GPRs, a particular data structure in QEMU (namely *CPUState*), used to describe the target CPU at execution time, needs to be modified.

On the other hand, Android relies on Java technologies to provide developers a complete environment for managing shared resources, communicating with lower layers, and providing the so-called Android framework. Moreover, Android application run in a specific Java virtual machine, known as Android Run Time (ART) (see Section A.1). Therefore, the rest of this section presents two fault injection approaches for Java applications.

Jaca [13] is a software fault injection tool for the robustness evaluation of Java programs. The tool's architecture is based on the Fault Injection

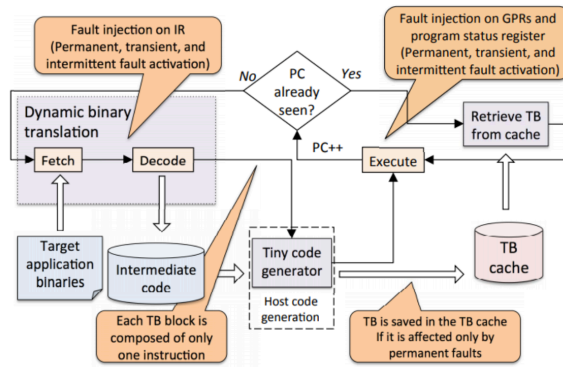


FIGURE 2.3: Fault Injection Approach in modified QEMU architecture (Figure 2 of Ferraretto *et al.* [3])

Pattern System, created by the same authors. Jaca and its documentation can be found on the official web page [56].

The fault load is defined by the user through the fault specification file. Every line of this file describes an injection, indicating the fault location (*e.g.*, attributes, method return values or parameters), the fault type (*i.e.*, how to corrupt the value), and the fault trigger (*i.e.*, every time, after or for a fixed number of invocations).

In my Master's thesis [14], I presented a **fault injector for Java programs**. It can inject various types of faults into a java software and assist software engineers to analyze the impact of such faults on the runtime behavior of the application.

The tool gets as input the code of a Java software component, and it can emulate two kinds of fault: internal faults (code defects) and external faults (Java exceptions). The injection of code changes for emulating the effects of real software faults is based on the empirical observation that code changes produce errors and failures that are similar to the ones produced by real software faults [57]. The faults are injected at the bytecode level, consistently with the Java language such as it worked with the source code.

2.2 Software Aging and Rejuvenation

This section reviews the most relevant results and techniques for the empirical analysis of software aging [58]. Software aging has been repeatedly reported both by scientific literature and by software practitioners [15], and it has been recognized as a chronic problem in many long-running software systems.

Garg *et al.* [16] presented an early study on software aging issues from systems in operation, by monitoring a network of UNIX workstations over a period of 53 days. This study adopted SNMP to collect data on resource consumption and OS activity, including memory, swap space, file, and process utilization metrics. The analysis found that the 33% of reported outages were related to resource exhaustion, and in particular to memory utilization (which exhibited the lowest *time-to-exhaustion* among the monitored resources).

Garg *et al.* [16], and later Grottke *et al.* [17], adopted statistical hypothesis testing and regression to identify *degradation trends* in resource consumption measurements (*i.e.*, if random fluctuations are excluded, the time series exhibits a gradual increase or decrease over time). The *Mann-Kendall test* and the *seasonal Kendall test* were adopted to confirm the presence of trends, respectively without and with periodic cycles, and the *Sen's procedure* and *autoregressive models* to forecast the time-to-exhaustion.

Silva *et al.* [18] and Matias *et al.* [19] studied software aging in SOA and web server environments by performing stress tests. They showed that aging can lead to gradual performance degradation in terms of throughput, latency, and success rate of web-service requests. A similar effect was observed by Carrozza *et al.* [59] on a CORBA-based middleware, in which the performance degradation of remote object invocations was attributed to memory leak issues, reducing the performance of memory allocators and bloating internal data structures.

Subsequent studies found that software aging issues can also affect the lower layers of the software stack, such as the Sun's Java Virtual Machine [20], the Linux kernel [21], and cloud management software [22]. In particular, the study on the JVM revealed that performance degradation trends were exacerbated by the inefficiency of the garbage collector.

Some empirical studies focused on the analysis of bugs behind software aging issues (*i.e.*, **aging-related bugs**), both in several open-source software

projects for the LAMP stack [23, 24] and cloud computing [25], and in embedded software used for space missions [26]. These studies provided insights on the nature of aging-related bugs: they represent a minor share of all software defects but are quite subtle to identify and to fix; most of them affect memory consumption and, in many cases, application-specific logical resources (such as thread pools and I/O connections).

Recent research has been focused on monitoring techniques to detect software aging in deployed systems, which is especially challenging due to varying workload conditions and configuration. They include machine learning techniques [27], such as decision trees and robust time series analysis techniques [28, 29], *e.g.*, the Cox-Stuart test and the Hodrick-Prescott filter.

Research on software aging in mobile devices is still at an early stage. Araujo *et al.* [30] designed a testbed for stress testing of Android applications, and found software aging issues in the Foursquare Android app. However, their approach was not meant to study aging issues inside the Android OS, and their tests did not point out any software aging symptom at the lower layers of the Android OS. Other studies were focused on preventing performance degradation of mobile applications through off-loading of tasks to the cloud and local application restarts [31, 32], debugging apps for performance bugs [33], and on forecasting Android device failures with time series analysis techniques [34].

A preliminary study on the aging phenomenon in Android OS has already been published [60]. This study was the base for the extensive analysis presented in this thesis.

2.3 Fuzz Testing

This section gives an overview of previous work in the general area of fuzzing.

Since its initial years, fuzz testing has been extensively adopted for testing systems software, such as network servers, shell applications, libraries, and OS kernels. The early study by Miller *et al.* [35] on fuzzing UNIX system utilities, by injecting random inputs through their command line interface and standard input stream, found a surprisingly high number of targets that experienced crashes, leaks and deadlocks, even when exposed to apparently

trivial (but invalid) inputs. Other approaches for OS robustness testing, such as BALLISTA [36], MAFALDA [61], and the DBench project [37] injected invalid inputs by bit-flipping them or replacing them with “difficult” inputs, or forced the failure of kernel APIs and device drivers [62,63].

As an example, **Ballista** [36] is a famous testing system built to evaluate the handling of exceptional input parameter values of POSIX functions and system calls. This approach emulates misuses of the kernel from the user space. The authors define the faultload based on the parameters data types of the POSIX calls, by defining a set of test values for every data type in the standard (*e.g.*, file handle or memory buffer). The test values are (valid and invalid) values both suggested from testing literature and chosen by the authors’ experience. For instance, these values are selected by considering: zero, negative one, maximum/minimum values, pointers to nonexistent memory, lengths near virtual memory page size, pointers to heap-allocated memory, files open for combinations of read/write with and without exceptional permission settings, and files/data structures that had been released before the test itself was executed. The Ballista approach is based on combinatorial testing using both valid and invalid parameter values. Every test case consists of a Module under Test (MuT) and the test values. After each test is executed, the approach classifies the results into ²:

- Catastrophic: the OS is corrupted and/or the machine crashes and reboots;
- Restart: a call to a MuT never returns and the task requires to be terminated and restarted;
- Abort: the task results in abnormal termination.

The same approach of Ballista can be partially used to evaluate the robustness of the Android Linux Kernel, that complies in large part the POSIX specification.

Among the most modern and mature fuzzing tools, **American Fuzzy Lop** (AFL) is well-known for having found notable vulnerabilities in dozens of popular libraries and applications [38]. AFL is an *instrumentation-guided genetic fuzzer*, which modifies the target program at compile-time in order to efficiently profile the branch coverage during the execution of the tests, and

²These categorization is a subset of the “C.R.A.S.H.” severity scale [64]

to communicate with the main AFL process. Based on coverage measurements, AFL iteratively improves the quality of fuzz inputs, by mutating the previous inputs that discovered new paths. AFL has also been extended to avoid compile-time instrumentation, by using the QEMU virtual machine to trace the instructions executed by the target (at the cost of higher run-time overhead and of the additional dependency on a virtual machine emulator). Another example of coverage-guided fuzzer is *syzkaller* [65], which also uses QEMU and compile-time instrumentation to fuzz the whole Linux kernel through its system call interface.

Another significant advance has been represented by white-box fuzzing techniques that leverage symbolic execution. The most well-known is **KLEE** [39], a virtual machine environment, based on the LLVM compiler infrastructure, with a symbolic state for every memory location (*i.e.*, boolean conditions that must hold at a given point of the execution) that is updated as code is executed by an interpreter. When KLEE encounters a branch condition, it forks in two execution flows, each with a different constraint on the variables involved in the branch condition. When a failure path is found, a constraint solver is used to find an input that fulfills all the conditions on that path. **SAGE** [40] is another well-known fuzzing tool by Microsoft: starting from some tentative concrete input, the tool traces the program execution using a record&replay framework [66] to identify the path constraints for the input; then, it negates one of these constraints, and uses a constraint solver to generate inputs to cover the new conditions. It is important to note that white-box fuzzing is extremely powerful, but very resource-consuming due to the overhead of constraint solving and to the exponential explosion of program paths. Thus, these techniques are best applied in combination with black-box fuzzing: Bounimova *et al.* [67] report a split of 66%-33% of bugs found respectively by black- and white-box fuzzing during the development of Microsoft's Windows 7. Moreover, white-box fuzzing can only be applied when the target is executed in an environment (such as a virtual machine) able to trace and to fork symbolic states.

In Android-related research, fuzzing has been extensively used to attack network and inter-process interfaces. For example, Mulliner and Miller [41] found severe vulnerabilities in the SMS protocol. **Droidfuzzer** [42] is a fuzzing tool that targets Android activities that accept MIME data

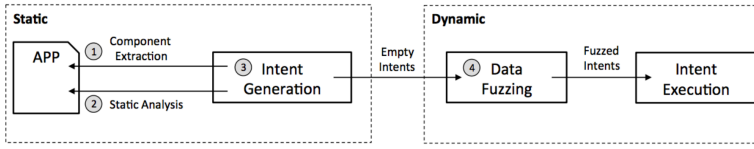


FIGURE 2.4: Intent Fuzzer Architecture (Figure 2 of Sasnauskas *et al.* [4])

through Intents (a higher-level IPC mechanism based on Binder IPC [68]) Sasnauskas *et al.* [4] developed a more generic **Intent fuzzer** that can mutate arbitrary fields of Intent objects. The aim is to balance the tension between generating intents that applications expect, permitting deep penetration into application logic, and generating intents that trigger interesting bugs that have not been previously uncovered. Fault load is based on intents and their structure. Faulty intents are created populating an empty intent with totally random values, using QuickCheck [69] as generator. The overview of the intent fuzzer is depicted in Figure 2.4. For each target app, the fuzzing work flow consists of:

- component extraction to identify the exported components and their actions;
- static analysis to obtain the structure of the expected intents;
- intent generation to create well-formed intents that trigger the actions;
- data fuzzing to randomly fuzz the intent data.

Component extraction is performed thanks to the information in the manifest file of the app, particularly intent filters information that allow to create intents for the fuzzing phase. Static analysis retrieves the structure of the intents that is processed during the execution of the advertised actions. Each new instance of an intent with fuzzed data is generated and explicitly sent to the target component for execution. Upon delivery, the component is first restarted and does not depend on previous executions. During intent execution, the tool monitors both code coverage (open-source apps only) and crashes.

Furthermore, Mahmood *et al.* [43] adopted the white-box fuzzing approach by decompiling Android apps to identify interesting inputs and

running them on Android emulator instances on the cloud. However, these and similar tools [44–46] focus on the robustness of Android apps, and can not be directly applied to fuzz Android system services.

Other work was done on different mobile OS. Miller *et al.* [47] presented and adopted a fuzzing tool, namely *zzuf*, for fuzzing iOS applications. It intercepts input files and applies random mutation. The authors found *zzuf* particularly efficient on targets such as media players, image viewers, and web browser, because of the quantity and complexity of files they take as input. Lee *et al.* [48] designed the Mobile Vulnerability Discovery Pipeline (**MVDP**), an approach that generates random, invalid input files to crash mobile apps, either Android or iOS, by exploiting the smartphone farms. Liang *et al.* [49] introduced **Caiipa**, a cloud service for testing Windows mobile apps. The apps are stressed with random GUI events under several contexts or conditions (*e.g.*, network connectivity and availability of sensors), distributing the tests among both emulators and actual devices.

To the best of our knowledge, the few notable studies on fuzzing Android system services are the ones by Cao *et al.* [50] and Feng *et al.* [51]. Cao *et al.* [50] focus on the input validation of Android system services. Their tool, **Buzzer**, sends crafted parcels (*i.e.*, the basic messages on the Binder) to invoke AOSP system services with fuzzed arguments. Since *Buzzer* was an early tool of its kind, it relied on manual efforts for several tasks, such as to identify the arguments of service methods, to avoid fuzzing on methods that could not be invoked by third-party apps anyways (due to limited permissions). Feng *et al.* [51] developed **BinderCracker**, a more sophisticated parameter-aware fuzzer that can automatically understand the format of Binder messages and that supports more complex communication patterns over the Binder (such as callback objects returned by system services). However, both these tools are purely black-box approaches and do not gather any information about the internal coverage of the tested services, thus missing the opportunity to improve the efficiency of fuzzing. This problem has only been partially addressed by Luo *et al.* [70], which recently developed a **successor of Buzzer** that exploits symbolic execution. However, this tool is not applicable to vendor customizations, since it is designed to run outside the Android system and requires the availability of the target source code.

Chapter 3

AndroFIT: A Software Fault Injection Approach for the Android Mobile OS

As engineers, we were going to be in a position to change the world - not just study it.





— Henry Petroski

Fault injection testing deliberately inserts a software threat into the system to assess whether the emulated fault in one of the software components affects all the other components or not. Android fault injection wants to analyze the Android OS behavior, when any of its component is faulty. The contribution of this work is three-fold:

- a novel methodology and methods to extract a fault model from a mobile OS architecture, and its application on the Android 5 (Lollipop), 6 (Marshmallow), and 7 (Nougat), extracting 871 faults from 14 components in 6 subsystems;
- a fault injection tool, namely AndroFIT, to support a fault injection campaign of an Android device, including all the fault injection techniques necessary to emulate the fault in the Android fault model;
- an experimental evaluation campaign for AndroFIT on 3 Android smartphones (*i.e.*, Samsung S6 Edge, HTC One M9, and Huawei P8), performing 2334 fault injection experiments, analyzing the propagation chains and suggesting reliability improvements.

3.1 Overview

This chapter presents the first aspect of this thesis: fault injection testing. Android and the other mobile systems consist of several components at different layers, that communicates each others to provide services to the final users¹. Fault injection testing is the approach to assess whether a fault in one of these components (the fault injection target or target) affects all the other components (the component under test or CUT) or not. Indeed, a fault in a component may lead an error in that component and be propagated to other components in the system, through the Inter Component Communication (ICC) channels. This is the so-called *fault-error-failure* propagation chain.

Figure 3.1 shows an example of propagation chain in Android OS.  A hardware sensor, such as the front camera of an Android smartphone, may break down because of a faulty connection.  The fault results into an erroneous state of the device driver, which is perceived as failure by upper components in the Android stack.  This failure represents a fault for the camera service of the Android OS, which is in charge of mediating accesses from applications to the camera. When an app tries to use the camera, it asks to the CameraManager, but the device driver's fault will cause the Camera Manager to throw an exception, that is, a failure of the Camera Manager.  Again, the exception represents a fault for the application: if the application does not properly handle this exception, it will experience a failure (*i.e.*, a crash of the application). This propagation chain can be applied to every hardware or software component in the Android OS. Summarizing: A fault can cause an internal error state of a component, and when the error surfaces to the delivered service of the component, a failure occurs. A failure from a component is a fault for other components in the system. Recursively, a fault produces an error, which is likely to propagate and create new errors; when the failure reaches the user, he/she experiences the failure of the Android system.

This chapter introduces a novel methodology and methods to extract a fault model from a mobile OS architecture. We analyzed the whole Android architecture and applied it to 14 components in 6 subsystems, extracting

¹Android architecture and mechanisms are presented in Appendix [A](#)

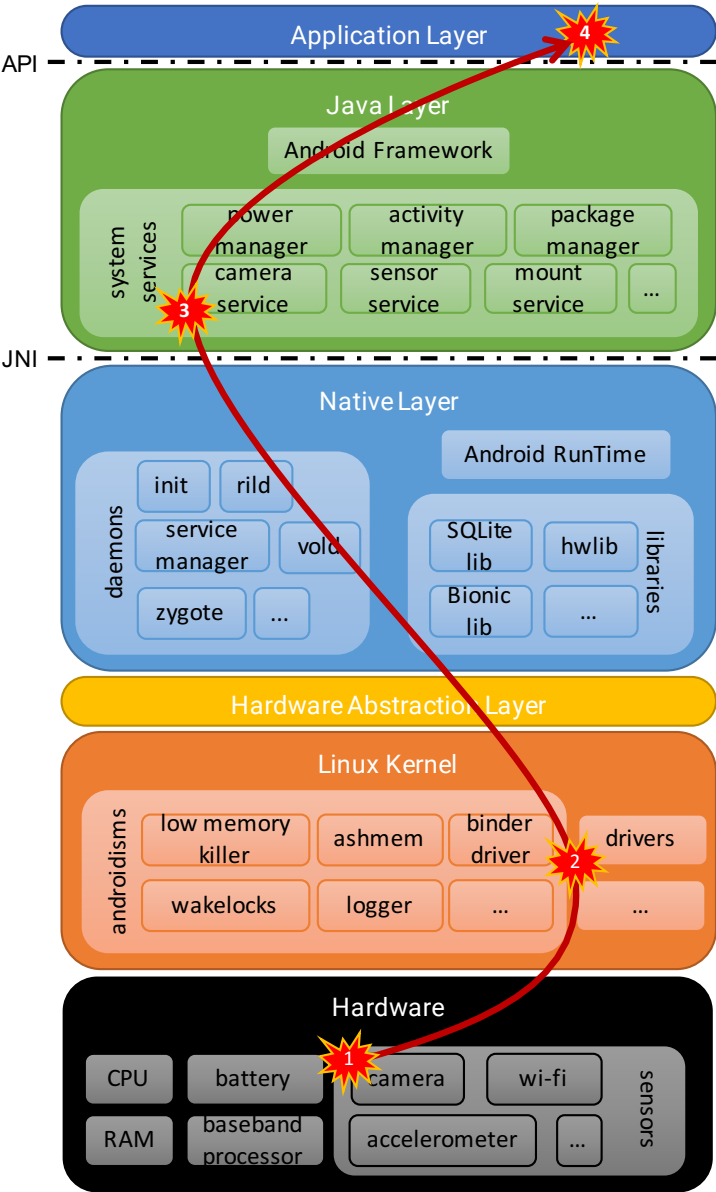


FIGURE 3.1: a Fault-Error-Failure Propagation Chain in Android

871 faults for Android 5 (Lollipop), 6 (Marshmallow), and 7 (Nougat). Furthermore, we designed and developed a fault injection tool suite, namely AndroFIT, to enable the fault injection testing in an Android system. We performed an experimental evaluation campaign on 3 Android smartphones: Samsung Galaxy S6 Edge, HTC One M9, and Huawei P8. They run Android 6 (Marshmallow). AndroFIT injected 780 faults within controlled experiments, gathering several system information (e.g., the Logcat logs). We analyzed the test outcomes, revealing strengths and weaknesses of the three devices. Moreover, we performed and presented an in-depth analysis of some failures to further understand the error propagation in the Android OS, also suggesting some potential reliability improvements.

3.2 Fault Modeling

This section includes the **Service Interfaces and Resources** (SIR) methodology for the definition of a mobile OS fault model, that is applied to define the fault model of the Android OS, presented in [3.2.2](#).

3.2.1 Methodology

A fault model (*i.e.*, a formal definition of how to change the code or state of the software to emulate faults [\[71\]](#)) is the basic element for any fault injection experiment. However, defining a fault model for *software* is also a problematic aspect, since software faults (*bugs*) involve the human factor (*e.g.*, mistakes by developers during the development lifecycle) that is difficult to understand and to characterize.

Previous studies on software fault injection addressed this aspect by following two approaches. The first approach has been to define *corruption patterns* based on past software faults, by analyzing either the buggy code (*e.g.*, by inspecting the bug-fixes of the faults) or the erroneous software states or outputs caused by the fault (*e.g.*, by inspecting problem descriptions reported by users or developers); and to emulate these corruptions by modifying either the code (similarly to mutating operators) [\[72\]](#), or the software state and outputs (*e.g.*, replacing them with random noise) [\[73,74\]](#). The second approach has been to define *exceptions* and *error codes* to be returned on API calls. These exceptions and error codes are identified by

analyzing definitions of the API interface, and are injected by throwing the exception or error code [75,76].

However, there is still a lack of a widely-agreed consensus on which approach is the most appropriate for fault modeling. The first one (corruption patterns) is quite onerous to apply, since it entails to manually look at a significant number of previous faults to get statistical confidence on the corruption patterns; it may even be inapplicable if there is little data about past faults. The second approach (exceptions/error codes) is more straightforward and is applicable to black-box software, but it is limited to a narrow class of software faults: previous work [77] highlighted that this approach does not account for a significant percentage of software faults, which are not signaled by any exception or error code.

We defined a fault modeling methodology oriented towards ease of use and applicability to the Android OS. We aimed to keep low the human effort to define the fault model, and to achieve a fault model that is enough comprehensive and credible to be accepted by engineers. To this goal, we introduced the **Service Interfaces and Resources (SIR)** methodology. SIR is a lightweight approach that only relies on the architectural analysis of the target system, driven by a set of checklists. It follows the second approach mentioned above (exceptions/error codes) to avoid the extensive analysis of internals and of past faults, which would not be affordable for complex software systems, such as the Android OS. However, in order to get a more realistic and comprehensive fault model, we extend the fault model beyond exceptions/error codes.

To define the fault model for the Android mobile OS, we started from the observation that it is a *service-oriented* system [78], as shown Figure 3.2, where its software components have two fundamental roles: they are providers of **services** that are consumed through well-defined interfaces exposed by remote procedure calls, libraries, sockets, system calls, and other communication mechanisms; and they are managers and users of **resources** (both logical and physical), such as memory, threads/processes, communication channels, and hardware devices. The interactions between a component and the rest of the system (other OS component, the user, the apps, the physical phone) must necessarily pass through service interfaces and resources.

The outcome of the SIR methodology is a set of **failure modes** for each

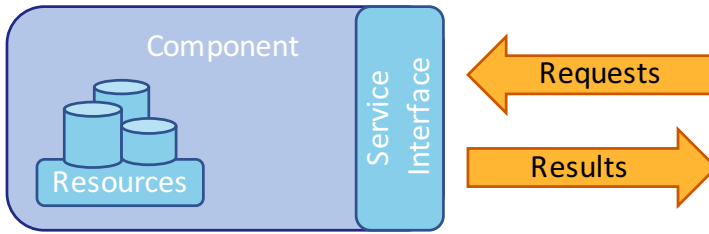


FIGURE 3.2: a Software Component Model View

component of the OS. With reference to the taxonomy of Avizienis et al. [7], a *failure mode* is an incorrect behavior of a component that is perceived by other components (e.g., through service interfaces), and that results from an incorrect state of the component (*error*), which is in turn caused by a fault inside the component. This *fault-error-failure* chain repeats again as the failure propagates to another component of the OS (the *failure* represents a *fault* for this other component), until it surfaces to the end-user as a *system failure*. In our approach, we position fault injection at the interfaces of a component: we inject *failure modes* of a component to emulate faults for the other components, and look for cascading failures (a *what-if* analysis). This approach aids us at defining the fault model by starting from the analysis of component's interfaces; moreover, injecting at component's interfaces is technically easier to implement in a reliable way, as it avoids to modify the component internals (i.e., its source- or binary-code) [79, 80].

The SIR methodology consists of three phases:

- analysis of the target architecture: for each component, the *services* provided by the component (e.g., an API function exposed by the component), and the *resources* managed by the component (e.g., memory or sockets) are identified.
- application of the failure modes: the failure modes are applied to every identified resource and service of the mobile OS to construct system faults.
- assignment of fault persistence: one or more fault persistence attribute is assigned to each fault, providing the final fault model of the mobile OS.

TABLE 3.1: A Comparison of Failure Classifications [5]

Barton [82]	Cristian [83]	Suh [84]
Response too late	Timing (early/late)	Timeout (late response)
Invalid Output	Response (value/state)	Failure (incorrect answer)
Crash Task stop (process crash)	Crash (partial/total amnesia, pause, halt)	Crash Abort (crash with error message)

Definition of Failure Modes

Powell *et al.* [81] proposes a general approach to describe failures in computer systems, and this section extends it for fault modeling of mobile OS. The authors define a service as a tuple $\langle vs, ts \rangle$. The vs is the value produced by the service, which can be a numerical result, an object, a data block, a message, or other types of output. The ts is the time at which the service response is observed.

The service is correct when vs is a correct value, and ts is short enough, according to the specification of the service (*e.g.*, user requirements). The service is faulty and produces a failure:

- in the value domain, when a fault affects the value produced by the component, *i.e.*, vs is incorrect (*e.g.*, the component may produce an out-of-range value, garbled data blocks, and out-of-sequence event or message);
- in the time domain, when a fault affects the timing of services delivered by the component, *i.e.*, ts is incorrect (*e.g.*, a component may response after a very long delay, or may not respond at all).

The SIR methodology considers four general classes of failure modes. These classes are broad and are derived from failure classifications from previous studies, as shown in Table 3.1 from Mukherjee and Siewiorek [5], which shows how the same failure modes were labeled with different terminologies [82–84].

Our failure modes considers that

- the component fails in the value domain and

- saturates, exhausts, or disables the resources that it uses or manages (**resource management failure**);
- produces a wrong service result, by returning incorrect data to its user (**output value failure**);

or

- the component fails in the time domain and
 - refuses to provide a service to its user, by returning an exception or error (**availability failure**);
 - provides a service response only after a long delay, or no response at all (**timeliness failure**).

Analysis of the Target Architecture

In this phase, for each component in the mobile OS, we extract a list of all the service interfaces implemented by the component, and all the resources used by the component. This information should be obtained from architectural documentation, for the inspection of the source code (if available), and from run-time and reverse engineering analysis of the mobile OS.

Application of the Failure Modes

In this phase, we need to apply the failure modes on service interfaces and resources extracted by the previous phase to obtain potential and realistic faults for the mobile OS. We developed the **SIR2F** (Service Interfaces and Resources to Faults) method to fulfill, where the faults are built by construction based on the four failure modes. The SIR2FM method is a lightweight method that consists of going through predefined checklists.

The first checklist focuses on components' services. The checklist has a series of questions to identify which of the four generic failure modes can happen for each service interface. A fault is added to the final fault model if the scenario is plausible according to the checklist:

1. Does the service interface declare exceptions, or erroneous return codes? If yes, add an *availability* failure for the service.

2. Can the service lose a request or response (e.g., due to service queue overflow, or omit to respond), without performing any operation? This possibility should be considered when the component is multi-threaded or event-driven. If yes, add a *timeliness* failure for the service.
3. Can the service experience a long delay? This possibility should be considered if the component performs complex processing on data (which may lead to performance bottlenecks) or performs high-volume I/O activity. If yes, add a *timeliness* failure for the service.
4. Can the service return a result (e.g., a numerical computation or a data structure) that may be incorrect due to a bug? This possibility should be considered if the service implements complex processing algorithms, or if it is responsible to generate complex data structures. If yes, add an *output value* failure for the service.

In a similar way, the second checklist focuses on components' resources:

1. Can the hosting process&threads crash (i.e., killed by the OS), or terminate prematurely, or be stalled (e.g., because of a deadlock), before replying? This possibility should be considered when the component is relatively large (several thousands of lines of code) and include native code. If yes, add an *resource management* failure for the use of processes or threads.
2. Is the resource protected by permissions, and can it become inaccessible due to lack of permission? For example, this is the case of inter-process shared resources in UNIX systems. If yes, add a *resource management* failure for the resource.
3. Can the component leak the resource (e.g., memory and file descriptors that are frequently allocated/deallocated), thus preventing further allocations of the resource? If yes, add a *resource management* failure for the resource.
4. Does the component allocate new processes or threads? These may terminate prematurely, or the component may hit hard system limits when allocating them (e.g., `ulimit` in UNIX systems). If yes, add a *resource management* failure for the use of processes or threads.

5. Does the component manages persistent files (e.g., a database file or a configuration file) that may be corrupted when reading or writing it?
If yes, add a *resource management* failure for the corruption of the file.

Assignment of Fault Persistence

In this final phase, the SIR methodology adds information on the **persistence** of the faults [7]. The fault persistence indicates the behavior of the injected fault over time, *i.e.*, whether it is *permanent* (the fault persists for a long period of time), *transient* (the fault occurs only in a specific moment of the execution), or *intermittent* (the fault appears periodically during the execution). The fault is flagged as permanent if the fault's effects are persistent unless explicitly recovered or cleaned (for example, a resource leak or a crash); as transient, if the hypothesized fault is triggered by a rare environmental condition (such as an exception); or as intermittent if the hypothesized fault is triggered by specific inputs to the service (for example, a data corruption caused by a corner case of an algorithm). A single item could be assigned to multiple persistences: in this case, we duplicate the item and generate a properly flagged fault for each assigned persistence.

When the SIR methodology is complete, we have a fault model in a tabular form: a row for each fault that can be injected in the component, where the columns are the name of the fault, the failure mode from which is derived, the name of the service or resource, a brief description of the fault, and the fault persistence.

The SIR methodology provides generic guidance for engineers, but it still leaves room for the human judgment, as it is their call to decide whether a service is *complex* or a condition is *rare* to apply the checklists. During our work on the fault model for the Android OS, we involved the test engineers in the company, by asking them if a fault could be plausible according to their personal experience with the Android OS. Framing the discussion in these terms helped us to iteratively improve the fault model, and to make it accepted by them as realistic.

3.2.2 Android Fault Model

To define the Android fault model, we refer to the Android architecture in Section A.1 and focus on 6 subsystems: phone, camera, sensors, activity,

package, storage. These 6 subsystem are arbitrary chosen as representative because with the highest impact on the final user and the highest interest from the vendor. Every subsystem consists of more than one components, however we consider the components at the lowest layers of the Android stack as fault injection targets (marked with *[target]*), and the components at the application and framework layers left as CUT (marked with *[CUT]*). We studied these components reading the source code of Android, from version 5 to 7, and reverse engineering them on actual smartphones.

We considered 14 fault injection target components with their interfaces, and formalized more than 870 potential faults for the Android OS. Table 3.2 provides a summary of the faults inside the fault model. The complete fault model is in Appendix B.

TABLE 3.2: Summary of the Android Fault Model

subsystem	fault injection target	resource management	output value	availability	timeliness	total
phone	RILD	11	12	12	24	59
	Baseband Driver and Processor	4	3	3	3	13
camera	Camera Service	12	30	30	102	174
	Camera HAL	0	12	9	36	57
	Camera Driver and Hardware	4	3	3	3	13
sensors	Sensors Service and HAL	14	6	6	6	32
	Sensors Drivers and Devices	48	36	36	36	156
activity	Activity Manager Service	7	3	15	30	55
package	Package Manager Service	7	9	6	24	46
storage	SQLite library	9	3	9	3	24
	Bionic library	0	36	18	36	90
	Volume Daemon	9	15	9	9	42
	Mount Service	7	6	6	6	25
	Storage Drivers and Devices	4	57	3	24	88
		136	231	165	342	874

Android Service Interfaces and Resource Failures

The analysis of the Android subsystems, as reported further in this section, identified the set of components services and resources types, that will be considered for formalizing the fault model.

The service interfaces types are the following:

- binder service interface: a service based on the Binder protocol, which provides a proxy object to communicate with a remote process. It can returns error or exception, it can corrupts the output parameters, and it can reply later or not at all.
- service over unix socket: a service based on the socket message exchange. it can return error on read/write, it can corrupt on read/write, and it can reply later or not at all on read/write.
- library service interfaces: a service exposed by specific libraries, usually vendor-specific. It can returns error or exception, it can corrupts the output parameters, and it can reply later or not at all.
- driver service over system call: a service provided by a driver that can be queried through system calls on specific device files. It can return error, it can corrupts the output parameters, and it can reply later or hang.

The resource types are the following:

- Processes&Threads: processes and threads are abstractions provided by the OS to execute programs (*e.g.*, the Media server uses several threads, one for each media-related service);
- Memory: memory is a volatile support to temporarily store information used by the CPU (*e.g.*, the RAM of the smartphone);
- Device Files: a device file is an interface for a device driver; it is not an ordinary file on storage, but it is a virtual file emulated by the device driver (*e.g.*, the camera driver exposes the virtual file `/dev/video0`);
- Sockets: a socket is an endpoint of bidirectional communication, used by two processes to communicate with bytestreams (*e.g.*, the RILD socket used by the RILD process to exchange phone commands and events with the application layer);

- **Pipes:** pipes are unidirectional bytestreams that connect the standard output of one process to the standard input of another process (*e.g.*, the AudioFlinger uses pipes to exchange audio streams between its threads);
- **Binder Objects:** a Binder object is an instance of a class that implements a Binder interface, a well-defined set of RPC methods, properties and events that are exchanged through the Binder driver (*e.g.*, the Connectivity Manager communicates with other network managers, such as BluetoothManager or WifiManager, using Binder objects as proxy);
- **(Ordinary) Files:** a file is an abstraction of the OS used to store information on a storage device (*e.g.*, executable code, configuration data, and multimedia data).

Phone Subsystem

The phone subsystem (Figure 3.3) is in charge of providing communication capabilities to the device such as telephone call and messages. It consists of the following components:

- *Phone Framework Services* [CUT]: an API library is exposed to applications; in turn, commands and events are exchanged with the RILD process through a UNIX socket interface;
- **RILD** [target]: a system process that embeds a proprietary, vendor-specific RIL library and the Event Scheduler, which dispatches the events from the baseband processor, and the commands from the upper layer;
- **Baseband Driver and Processor** [target]: the Baseband Driver exposes a device file (*e.g.*, `/dev/ttyS1` or `/dev/ttyUSB1`) to send/receive commands and events to/from the Baseband Processor, which performs the actual signal transfers.

To apply the SIR methodology, we analyzed documentation on the Android architecture and the open-source version of the Android OS [9, 78, 85, 86], to obtain the list of all the service interfaces and resources for the

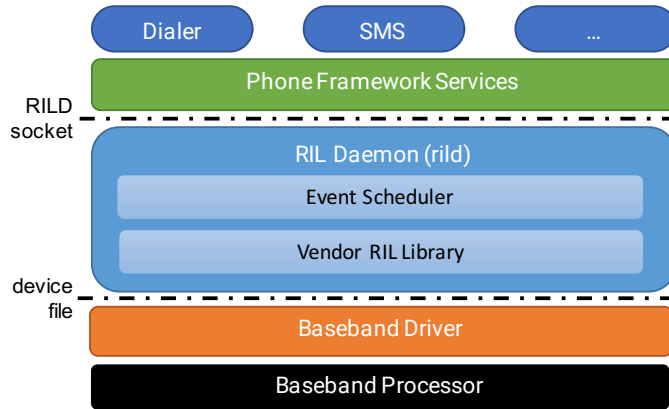


FIGURE 3.3: Architecture of the Android Phone Subsystem

RILD and Baseband Driver and Processor components. The RILD provides services over a UNIX socket, and consumes services of the Baseband Driver through system calls on a device file. The Baseband Processor is in charge of physically communicate with the actual network it is attached to. Focusing on a single component, the RILD includes the following service interfaces:

- *Receive phone commands on RILD socket:* the RILD receives phone commands from the stock apps (start a call, send a message, etc.);
- *Send phone events on RILD socket:* the RILD sends phone events to the upper layers (e.g., a call is dropped);
- *Write AT command to modem:* the RILD sends commands to the Baseband Driver and Processor, using AT the protocol [87];
- *Read AT response from modem:* the RILD reads and handles AT commands from the Baseband Driver and Processor;

The RILD resources include:

- *Process and threads:* the RILD process and its threads;
- *Memory:* the memory used by the RILD process and its threads;
- *Sockets:* the RILD uses a socket to communicate with the phone library;

- *Pipes*: the RILD uses pipes to enable communication between different threads;

The RILD service interfaces are based on socket and file primitives, such as `receive`, `send`, `read`, and `write`. They all declare erroneous return codes that can be encountered during service. Thus, we introduce availability failures for all the RILD services. The RILD service is a multi-threaded service that could be flooded by several messages, from/to both the higher and lower levels, in a short amount of time. There is a not negligible possibility that the service lose requests or responses. Thus, we introduce timeliness failures. Similarly, other timeliness failures are added considering the potential delay that can be accumulated when handling such a great amount of messages. The RILD service also handles the data transmitted with these messages, that can be altered in an involuntary way by the dispatching algorithms. Thus, we also introduce output value failures for all the RILD service. Moreover, the RILD is hosted by a specific native process (*i.e.*, the `rild` process) that could crash or hangs. Similarly, memory and sockets are protected by strong permissions or can be easily leaked. For all this possibilities, we introduce the resource management failures for the RILD component.

Using the checklists and defining the fault persistences, we introduced a total of 59 faults for the RILD fault model. The fault model of RILD is presented in Table B.1

The faults in the baseband driver and processor, Table B.2, affect the state of the phone, such as: the phone is inactive, or the kernel cannot access it; and the phone traffic, such as: AT events or commands are ignored or corrupted; data transfers through the kernel are corrupted because of incorrect memory management (*e.g.*, failed allocations of an I/O region on the PCI bus management) or protocol I/O errors with the device controller (*e.g.*, an incorrect write to a control register). Resources can be corrupted, such as memory and device files.

Camera Subsystem

The camera subsystem, presented in Figure 3.4, consists of the following components:

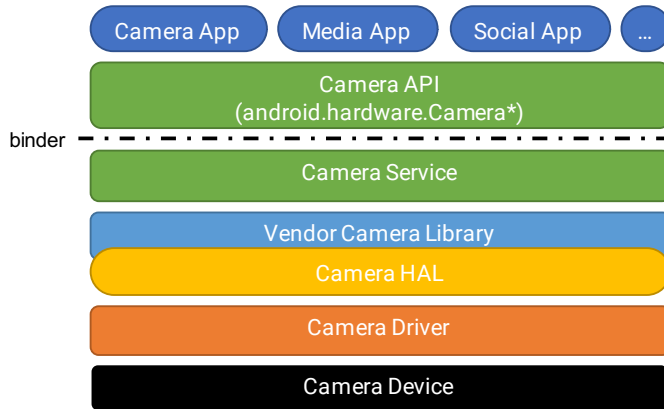


FIGURE 3.4: Architecture of the Android Camera Subsystem

- Camera API [CUT]: it provides a Java interface (*i.e.*, `android.hardware.Camera*`) for Android applications that use the camera;
- Camera Service [target]: it provides the media server process an interface through the Binder IPC for handling camera image streams and metadata;
- Camera HAL [target]: it interacts with the camera service, it uses a vendor-specific library to handle the camera device, it receives data from the camera hardware, and it performs basic image filtering (*e.g.*, scaling, cropping, and noise reduction);
- Camera Driver and Hardware [target]: the driver handles the camera at the kernel-level.

The Camera Service is hosted by the media server process, which exposes the camera services to other processes in the Android OS. It provides the Camera Service several resources, and they will be included in the fault model of Camera Service, even if the media server process provides the same resources also to other hosted services.

The camera subsystem can be affected by the faults in the Camera Service, the Camera HAL, the Camera Driver and Hardware.

The faults in the Camera Service, Table B.3, affect the IPC interactions between the Camera subsystem and applications. The Camera Service

API may return errors or be unresponsive; or, the camera applications may overload the Camera subsystem or generate incorrect parameters. Resources can be corrupted, such as process&threads, memory, files, sockets, pipes, and binder objects.

The faults in the Camera HAL , Table B.4, affects the use of the vendor-specific libcamera library, which handles image streams from the Camera device.

The faults in the Camera Driver and Hardware, Table B.5, can affect the state of the camera, such as: the camera is inactive, or cannot be accessed it; the camera commands are ignored or corrupted; data transfers through the kernel are corrupted because of incorrect memory management (*e.g.*, failed allocations of an I/O region) or protocol I/O errors with the device controller (*e.g.*, an incorrect write to a control register). Resources can be corrupted, such as memory and device files.

Sensors Subsystem

The sensors subsystem, presented in Figure 3.5, consists of the following components:

- Sensor Manager [CUT]: it is part of the Android Framework, and it provides classes and APIs to consume sensor measurements;
- Sensor Service and HAL [target]: the Sensor Service executes within the system server process, it provides a Binder interface to the Android Framework, and the HAL uses a vendor-specific sensors library to poll sensor events through files in the /dev and /sys virtual file system.
- Sensors Drivers and Devices [target]: the drivers handle the sensors at kernel-level.

The Sensor Service is hosted by the system server process, which provides several resources. They will be included in the fault model of Sensors Service, even if the system server process provides the same resources also to other hosted services.

The Android platform supports three categories of sensors:

- Motion sensors, which measure acceleration, forces and rotational forces along axes. This category includes accelerometers, gravity sensors, gyro- scopes, and rotational vector sensors.

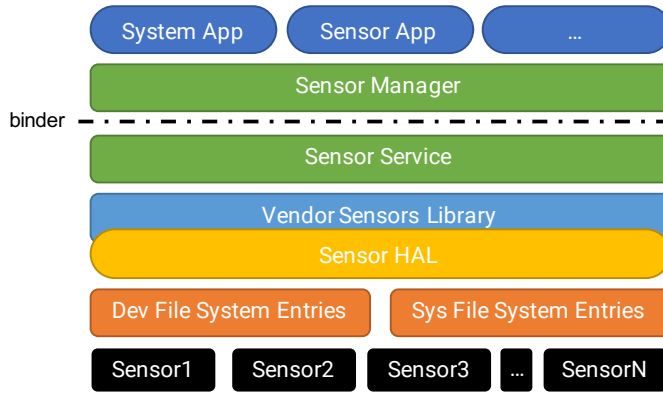


FIGURE 3.5: Architecture of the Android Sensors Subsystem

- Environmental sensors, which measure various environmental parameters, such as ambient air temperature and pressure, illumination, and humidity. This category includes barometers, photometers, and thermometers.
- Position sensors, which measure the physical position of a device. This category includes orientation sensors and magnetometers.

The sensors subsystem can be affected by the faults in the Sensor Service, the Sensors HAL, the Sensors Drivers and Devices.

The faults in the Sensor Service and HAL, Table B.6, may affect the sensor data and information reported to the application layer, the responsiveness of the Sensor Service threads, and the configuration of the sensors subsystem (*e.g.*, sampling period or accuracy level). Resources can be corrupted, such as process&threads, memory, files, sockets, and binder objects.

The faults in the Sensor Drivers and Devices, Table B.7, affect the state of the sensor, such as: the sensors are inactive, or the kernel cannot access them, and the sensor data are ignored or corrupted; data transfers through the kernel are corrupted because of incorrect memory management (*e.g.*, messages that contain acceleration or orientation values) or protocol I/O errors with the device controller (*e.g.*, an incorrect write to a control register). Resources can be corrupted, such as memory and device files. The fault model in Table B.7 is for a generic Android sensor, and it is actually specialized for each Android supported sensor, such as temperature sensor,

orientation sensor, accelerometer, gravity sensor, gyroscope, uncalibrated gyroscope, linear acceleration sensor, step counter, magnetic field sensor, light sensor, pressure sensor, and relative humidity sensor.

Activity Subsystem

The activity subsystem consists of the following components:

- Activity Manager [CUT]: it presents the activity services as a Java interface in the framework;
- Activity Manager Service [target]: it provides services to start and handle Android activities, and manage the activity stack.

The Activity Manager Service is hosted by the system server process, which provides several resources. They will not be included in the fault model of Activity Manager Service, because they are already included in the Sensor Service and HAL fault model.

The faults in the Activity Manager Service can affect the activity management operations, Table B.8. Resources can be corrupted, such as process&threads, sockets, pipes, and binder objects.

Package Subsystem

The package subsystem consists of the following components:

- Package Manager [CUT]: it presents the package services as a Java interface in the framework;
- Package Manager Service [target]: it provides services to install or remove packages, and manage the package permissions and intent resolution.

The Package Manager Service is hosted by the system server process, which provides several resources. They will not be included in the fault model of Package Manager Service, because they are already included in the Sensor Service and HAL fault model.

The faults in the Package Manager Service can affect the package and permission management operations, Table B.9. Resources can be corrupted, such as process&threads, sockets, pipes, and binder objects.

Storage Subsystem

The storage subsystem, presented in Figure 3.6, consists of the following components:

- Application Framework [CUT]: it provides several high-level I/O interfaces for Java applications for storing data in SQL database (*i.e.*, android.database APIs), for managing data stores (*i.e.*, android.os.storage), and for accessing files;
- SQLite Library [target]: it is adopted to embed a lightweight SQL DBMS into Android applications;
- Bionic Library [target]: it is a lightweight C library for Android, that includes many library functions for accessing the storage;
- Mount Service [target]: it is implemented into the System Server, it provides an API to manage volumes, and it interacts with the Volume Daemon.
- Volume Daemon [target]: it is an Android process that manages the internal and external storage partitions in the Android system, it automatically mounts partitions on the Android filesystem, both at boot-time and on demand for external storage, it manages their configuration (*e.g.*, labels, mount points, and permissions), and it receives events from the Linux kernel through a Netlink interface;
- Storage Drivers and Devices [target]: the drivers handle the storage devices at the kernel-level.

The Mount Service is hosted by the system server process, which provides several resources. They will not be included in the fault model of Mount Service, because they are already included in the Sensor Service and HAL fault model.

The storage subsystem can be affected by the faults in the SQLite Library, the Bionic Library, the Mount Service, the Volume Daemon, the Storage Drivers and Devices.

The faults in the SQLite library, Table B.10, affect the execution of SQL queries on the database, both insertions and selections (*e.g.*, the queries can be aborted or be slowed down); the correctness of data processed by

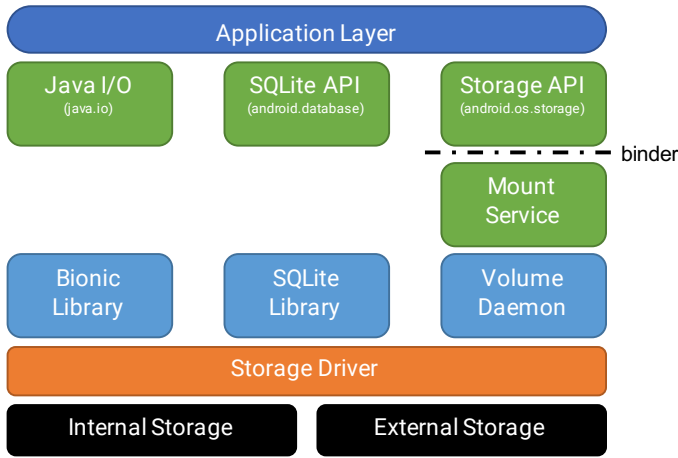


FIGURE 3.6: Architecture of the Android Storage Subsystem

queries (e.g., a query only inserts partial data, omitting some tuples); and the correctness of the physical database (e.g., the database file may be truncated or corrupted with random errors). Resources can be corrupted, such as files.

The faults in the Bionic library, Table B.11, impacts on applications that use I/O library call provided by the library. In particular, the most relevant I/O library call used in Android applications include: open, read, write, seek, close, and link. Table B.11 presents only the faults related to open and read library functions, but the actual fault model consists of the faults of all the storage-related library functions.

The faults in the Mount Service, Table B.12, and in the Volume Daemon, Table B.13, may cause that the storage partitions may not be available. For example, the Android system may be unable to mount an external storage inserted by the user. Resources can be corrupted, such as process&threads, memory, files, sockets, pipes, and binder objects.

The faults in the Storage Drivers and Devices, Table B.14, are related to the I/O operations. The drivers can become performance bottlenecks when accessing the storage, or they can corrupt data from/to the storage. The faults in storage hardware can corrupt the physical blocks managed by the filesystem. The most critical types of blocks are: Superblocks, Inodes, Data Blocks, Dentries. Moreover, the physical storage can generate I/O errors (e.g., due to a problem in the storage controller or firmware) when accessing

to the blocks. Resources can be corrupted, such as memory and device files.

3.3 Android Fault Injection Tool (AndroFIT)

This section first presents the fault injection techniques necessary to emulate the faults in the Android fault model. Then, it reports the design and implementation of the android fault injection tool, namely AndroFIT.

3.3.1 Fault Injection Techniques

The fault injection techniques for the Android platform, presented hereafter, are the ones necessary to emulate the faults in the Android fault model (Subsection 3.2.2). They are derived from the analysis of the Android architecture and from the survey of previous work on fault injection techniques (Section 2.1).

Table 3.3 maps these techniques with the target components in the Android fault model.

Binder IPC Hijacking

Binder is the most important IPC mechanism of the Android OS. A client can invoke a method on a proxy that implements a public interface; the proxy sends the request over the binder driver with the `ioctl` system call; the server receives the request and, potentially, respond back to the client through the binder driver (*cf.* Section A.2).

The binder IPC hijacking technique intercepts IPC messages that the target component (*e.g.*, Camera Service) sends to and receive from the binder driver. More specifically, the injector intercepts the `ioctl` system call on the binder driver and modify the contents of the messages.

The injector consists of two main components:

- the target controller, which remotely controls the target process (*i.e.*, the process to be injected), by forcing it to perform function call to the injection library; and
- the injection library, which modifies the ELF relocation tables of the target process, provides fault injection functions, and is loaded as a dynamic library in the context of the target process.

The steps of this fault injection technique are:

- the injector main method invokes the `ptrace` system call to probe the target process, which generates binder messages that will be injected;
- the injector uses the `ptrace` system call to mislead the target process to call the `dlopen` function, *i.e.*,
 - the injector saves the current processor registers of the target process;
 - it replaces the instruction pointer (IP) register with the address of the `dlopen` function and the registers with its parameters²;
 - the target process loads the injection library.
- the library loader executes the `init` method in the injection library, which modifies the procedure linkage table (PLT) section of the target process, by replacing the address of the `ioctl` function with the address of an hook function in the injection library.
- from this moment, every time the target process invokes the `ioctl` system call, the hook function will be first invoked. This function will perform fault injection, and then will invoke the original `ioctl` function.

For fault injection purpose, the most important messages are the transaction and reply messages. The transaction messages contain an identification code which identifies the invoked ROP, and the set of input parameters of the RPC. To request the execution of an RPC, a client process sends messages through the binder to the server process. Then, the client process receives a response from the server with the return value of the requested RPC (*cfr.*Section A.2).

Figure 3.7 shows how the injector operates on transaction messages sent over the binder. The injector intercepts the messages, which are blue in the figure, the client process sends to the server process. After the injector catches a transaction messages, it modifies the message content. Then, the

²Due to Address Space Layout Randomization (ASLR), the address of `dlopen` function has to be discovered by inspecting the `/proc/<PID>/maps` file, which contains the addresses of shared libraries linked to the process.

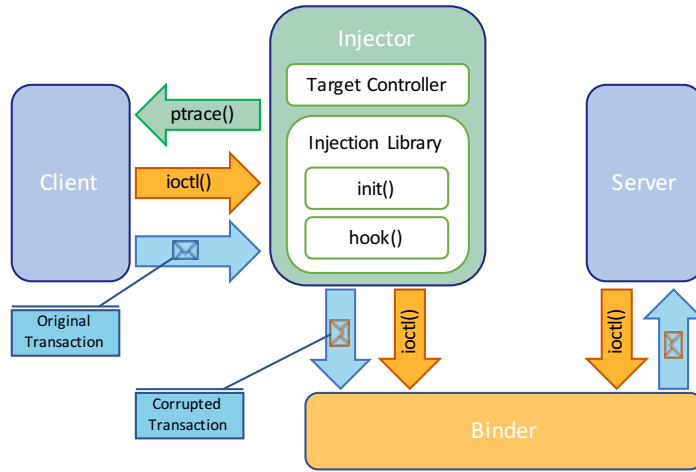


FIGURE 3.7: Binder IPC Hijacking Fault Injection Technique on Transaction Messages

injector calls the actual `ioctl` system call which requests the binder driver to deliver the corrupted message, red-colored in figure, to the server process.

Similarly, Figure 3.8 shows how the injector operates on reply messages sent over the binder. The injector intercepts the messages, which are blue in the figure, the server process sends to the client process. Then, the injector corrupts the message contents when the client process receives the reply message from the server process.

For example, a camera application communicates with the Camera Service in the media server process through binder, to send commands and to set the parameters of the camera devices (*e.g.*, to take a photo). This injection technique can inject incorrect parameters to the camera, by corrupting the input parameters to the RPC towards the Camera Service. The camera application receives also a reply message that contains the zero value if the phone has been correctly taken, not zero otherwise. This injection technique can corrupt the return value to emulate an API failure caused by faulty camera hardware.

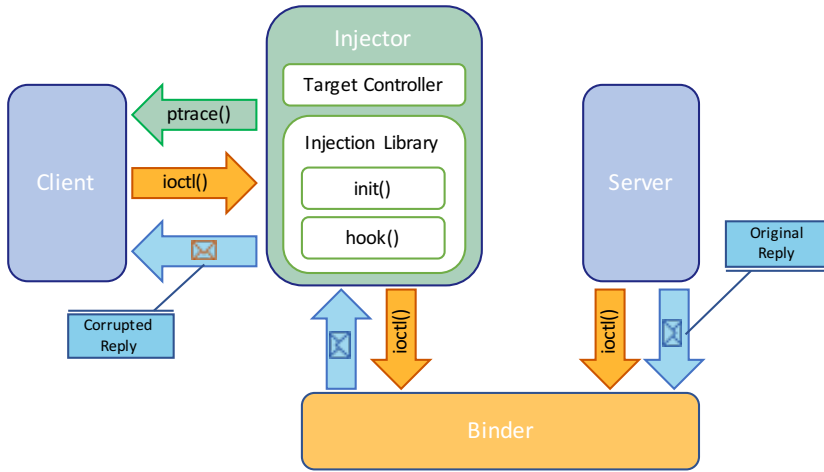


FIGURE 3.8: Binder IPC Hijacking Fault Injection Technique on Reply Messages

Library Hooking

The library hooking technique intercepts invocations of library functions, and allows to emulate faults in shared libraries. This technique changes the execution flow of the target process to invoke an hook function, instead of the original library function.

The injector consists of the following components:

- the target controller, which remotely controls the target process, by forcing the target process to load the injection library in the same way of the binder IPC hijacking technique; and
- the injection library, which diverts the control flow of the target process libraries to a set of libraries owned by the injector.

The injection library uses a control flow modification procedure to modify the control flow of the target process, which selects the addresses of the library entries stored in the symbol table of the target process. The control flow modification looks at the relocation table of the target process and substitutes the original library function with the addresses of the hooks. Each hook can inject the corruption of the input parameters, the delay of the actual function invocation, or the corruption of the return value.

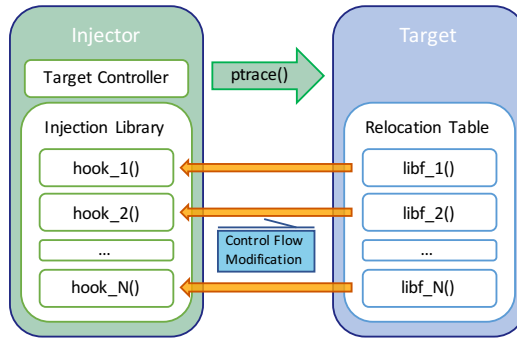


FIGURE 3.9: Library Hooking Fault Injection Technique

Figure 3.9 shows how the injector operates.

For example, the write function of the bionic library is hooked. The control flow of the target process does not go directly into bionic, but it is diverted to a custom write function, the hook, that always returns the EIO error code.

System Call Hooking

The system call hooking technique diverts system call executions and allow to emulate faults in the kernel and native components. This technique changes the execution flow of call on the system call interface by the target process.

The injector consists of the following components:

- the target controller, which remotely controls the target process, by forcing the target process to load the injection library in the same way of the binder IPC hijacking technique; and
- the injection library, which diverts the control flow of the target process libraries to a set of libraries owned by the injector.

The injection library uses a system call entry modification procedure which forces the target process to call the system call hooks. This operation can be done using the `ptrace` system call. Each hook can inject the corruption of the parameters, the delay of the actual system call invocation, or the corruption of the return value.

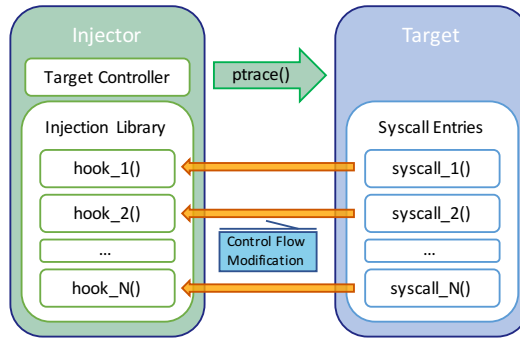


FIGURE 3.10: System Call Hooking Fault Injection Technique

Figure 3.10 shows how the injector operates.

UNIX Socket Hijacking

The UNIX socket hijacking technique intercepts messages which are sent or received by a target component (*e.g.*, RILD, or Sensor Service) to or from UNIX socket.

The injector consists of the following components:

- the target controller, which remotely controls the target process, by forcing the target process to load the injection library in the same way of the binder IPC hijacking technique; and
- the injection library, which probes the `send` and `receive` functions on UNIX sockets of the target process, in order to intercept the messages and to modify their contents

The injection library finds the points in the code area where the target process sends or receives the messages through the sockets, and it instruments the found locations with custom functions that redirect the message flow.

Figure 3.11 shows how the injector operates.

For example, the Sensors Service thread notifies clients with sensor events, which are messages sent though sockets. The injector intercepts

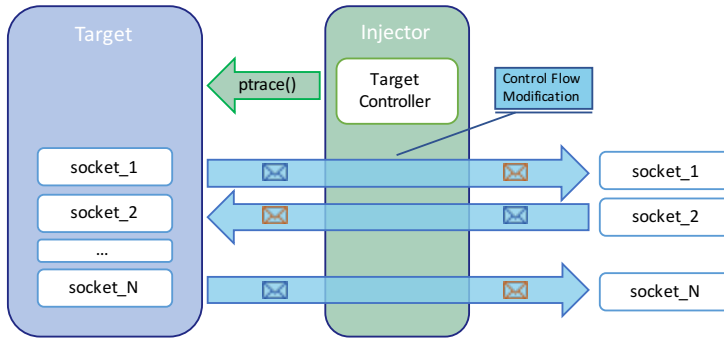


FIGURE 3.11: Unix Socket Hijacking Fault Injection Technique

the Sensors Service message and modify their content to emulate several software bugs within the Sensors Service.

UNIX Signaling

The UNIX signaling technique simply exploits the UNIX signals, which are messages sent from a process to another process to force the execution of a signal handles, and to change the state of the signaled process. For fault injection purposes, UNIX signals are used to force the premature termination of the target process, and the stall of the target process.

The injector main component is the UNIX signal emitter, which send the UNIX signals to the target.

The SIGSEGV signal is used in UNIX systems to notify an illegal memory access during a crash failure of a program. Therefore, to inject a crash failure, the UNIX signal emitter sends the SIGSEGV signal to the target process, using the `signal` system call. The SIGSEGV signal forces the same behavior of a crash caused by a memory management bug (*e.g.*, an invalid pointer). Moreover, UNIX processes can become stalled (*i.e.*, hangs) due to a synchronization or I/O bug, which leads to an indefinite wait on a synchronization primitive. To emulate a hang failure, the UNIX signal emitter sends the SIGSTOP signal. This signal pauses the execution of the target process, thus forcing the stall of the process.

Figure 3.12 shows how the injector operates.

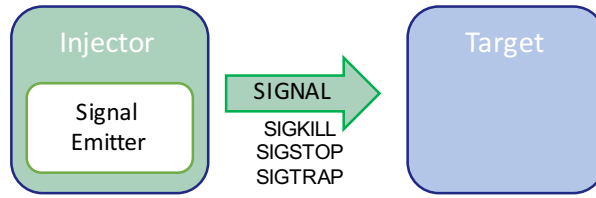


FIGURE 3.12: Unix Signaling Fault Injection Technique

3.3.2 Design and Implementation of AndroFIT

The Android Fault Injection Tool (AndroFIT) suite is designed to support a fault injection campaign of an Android device. It is a collection of scripts and tools deployed on both the workstation and the Android device under test. The Android device is connected to the workstation through an USB cable.

The AndroFIT suite, Figure 3.13, consists of the following parts:

- installation scripts: to compile the fault injector executable, to copy it on the Android device among with other libraries and scripts required for the tests, and to prepare the scripts for controlling the test;
- workstation scripts: to orchestrates the device scripts and fault injector executable, by providing the user a command-line interface;
- device scripts: to identify the version and the capabilities of the Android device under test, to generate configuration files for the fault injection experiments, and to perform the fault injection experiments;
- fault injector executable: to perform the actual fault injection, implementing the technologies discussed in Subsection 3.3.1.

AndroFIT currently supports smartphones powered by Android from version 5.0 (Lollipop) to version 7.1 (Nougat). The smartphone must have the developer mode enabled [88], the debug USB option enabled, root privileges, and a valid SIM card (to perform fault injection experiments on the phone sybsystem). On the other hand, the workstation must have the Android standard development kit (SDK) and native development kit (NDK) installed. Furthermore, a Linux-like shell is required to start AndroFIT.

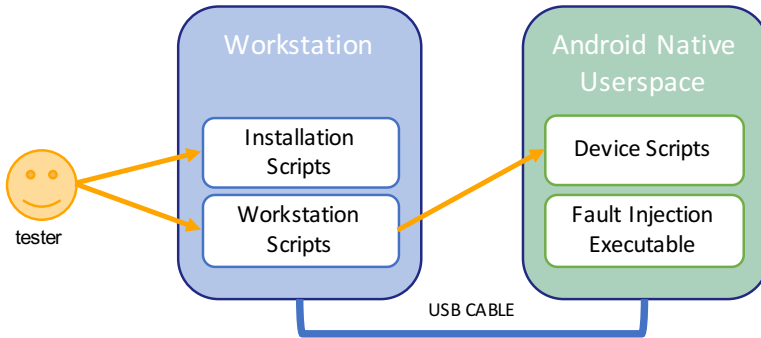


FIGURE 3.13: AndroFIT Architecture

AndroFIT has two python scripts as entry points: the injector and the experiment launcher.

The injector simply injects the fault and its syntax is:

```
Workstation# python inject.py [-h] [-d] [--version] --subsystem
SUBSYSTEM --component COMPONENT --target TARGET --failure FAILURE
{failure options} [--failure-timing {permanent,intermittent,
transient}] [--injection-start INJECTION_START] [--injection-
duration INJECTION_DURATION] [--random-seed RANDOM_SEED]
```

where

- `-h` is a optional command-line argument to print the usage and exit;
- `-d` is a optional command-line argument to add verbosity to console output;
- `--version` is a optional command-line argument to print the version and exit;
- `--subsystem` is a command-line argument to select the subsystem where to inject faults;
- `--component` is command-line argument to specify the component of the subsystem in which inject;
- `--target` is command-line argument to specify the target (*e.g.*, function, method, API) of the component in which to inject;

- `--failure` is command-line argument to specify the failure type to inject, chosen among `availability`, `timeliness`, `corruption`, `crash`, `hang` and `resource_corruption`. Others options follows according to the specific failure:
 - `unavailability` blocks and makes the target call return an error case, which can be specified by the `--unavailability-error-code` optional argument followed by the desired error code (default is `-1`);
 - `timeliness` blocks and makes the target call either delay for a specified time, `--delay-time` followed by the number of seconds to wait, or stall, `--no-response` (one of the two must be specified);
 - `corruption` intercepts and corrupts randomly one of either all the parameters of the target call, `--corrupt-all-parameters`, or the parameters specified their positions, `--parameters-positions` followed by the positions of the parameters to corrupts (default is corrupt all parameters);
 - `crash` enables crash injection only when the target is process;
 - `hang` enables hang injection only when the target is process;
 - `resource_corruption` enables resources corruption fault injection only when the target is either process or driver into a specified resource by the argument `--resource`, and may assume the values `memory` and `device_file`, for driver, or `memory`, `file`, `socket`, `binder`, and `thread`, for process.
- `--failure-timing` is an optional command-line argument to indicate the frequency of injection. The following timings are available: `permanent` (100%), `intermittent`(40%), and `transient` (10%)(default is `permanent`);
- `--injection-start` is an optional command-line argument with an integer to indicate when, in seconds, injection actually starts once the experiment starts. For example, if 2, the injections starts after 2 seconds the experiment begin (default is 0);

- `--injection-duration` is an option command-line argument with an integer to indicate how long, in seconds, is the injection. For example, if 10, the injections lasts for 10 seconds (default is 120);
- `--random-seed` is an option command-line argument with an integer used as seed for the random utility of the tool.

Despite the long list of potential arguments and parameters, the command-line tool guides the user in the selection of the necessary arguments, argument by argument. So, if the user want to inject in the camera service api version 1, but he/she does not know what are the potential targets, the user just launch the tool to have some hints, as shown below

```
Workstation# python inject.py --subsystem camera --component
camera_service_v1 --target ↵
      _ _ _ _ _
    _/ _ | _ _ _ _ _/ _ _ _ _ _/ _ _ _ _ _/
    / _ _ | / _ _ / _ _ _ _ _/ _ _ _ _ _/
    / _ _ | / _ _ _ _ _/ _ _ _ _ _/
usage: python inject.py --subsystem camera --component
camera_service_v1 --target {start_preview,stop_preview,
start_recording,stop_recording,take_picture,set_parameters,
get_parameters,send_command,notify_callback, data_callback}
python inject.py --subsystem camera --component camera_service_v1:
error: argument --target: expected one argument
Workstation#
```

The experiment launcher starts a fault injection campaign, and it automatically executes all the fault injection experiments.

A fault injection experiment, as shown in Figure 3.14, consists of two phases:

- Phase 1: a generic workload is executed to emulate user common actions;
- Phase 2: a fault is injected, by the injection techniques presented in Subsection 3.3.1, and the execution of a specific set of actions that will eventually activate the fault in the target (*e.g.*, if the fault is injected in the phone subsystem, the triggering workload consists of a phone call).

In order to have a clean device and almost-identical initial condition between experiments, the experiment launcher reboots the device between experiments. During both phases of the experiment, the experiment launcher collects failure and performance data.

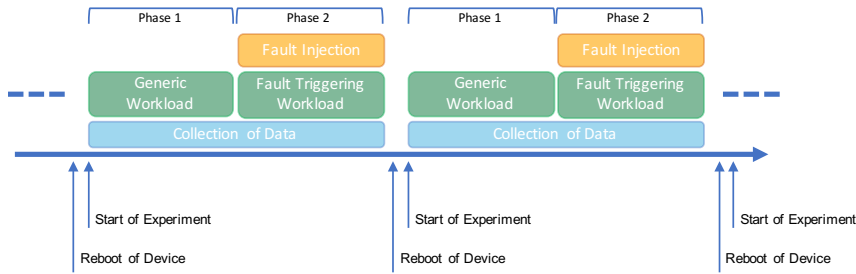


FIGURE 3.14: Execution of the Fault Injection Experiments

Figure 3.15 shows the components involved in a single experiment and the execution flow between them in order to perform the experiment.

① The entry point is the experiment launcher script that takes the campaign file as input and loads all the necessary files into the adb-connected Android device. Then, for each line in the file, ② it first starts the data collector, that uses the Android logcat and Linux proc files and ps command. ③ It starts phase 1 by starting the generic workload generator, that communicates with the Android device through adb exploiting mechanisms such as monkey, event generator and service calls. Then, ④ it starts phase 2 by starting the fault triggering workload generator and activating the proper injectors into the Android device. Finally, ⑤ data are gathered and saved in a hierarchical structure as explained further.

The main input of the experiment launcher is a file that list all the experiments that should be executed. A bunch of experiments is called experimental campaign or, simply, a campaign. Thus, this file is further referred to as the campaign file. Each line of the campaign file represent an experiment of the campaign, and is structured as

```
TRIGGER, PARAMETERS [, DESCRIPTION]
```

where TRIGGER indicates one of the potential triggering workload, such as

- camera, which opens the main camera and takes a picture;
- phone, which dials and calls a mobile phone number;
- sensors, which opens the sensors app; and

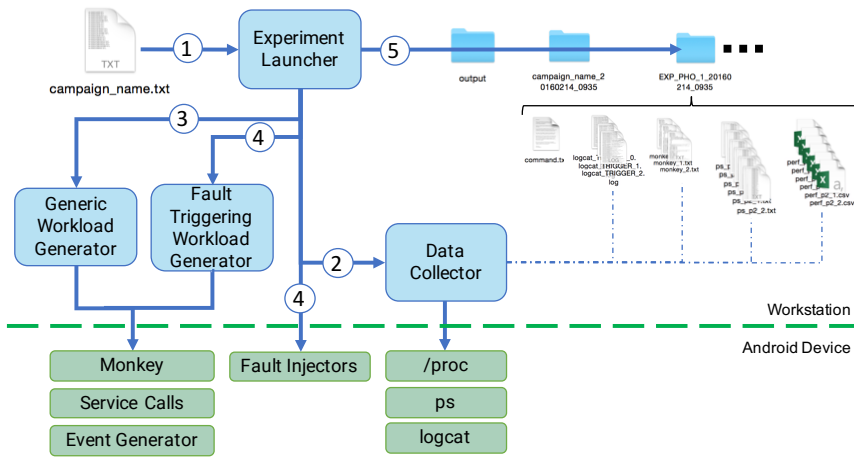


FIGURE 3.15: Flow of a Fault Injection Experiment

- user, which launches a monkey [89] script to emulate generic user inputs;

PARAMETERS have the same meanings and potential values as in the injector entry point; and DESCRIPTION is an optional argument that may represent a brief description of the experiment to be printed during the script execution on the console. The experiment launcher automatically executes each experiment in the campaign file.

The syntax to use the experiment launcher is:

```
Workstation# python experiment_launcher.py [-h] -f FILE [-n NUM] [--verbose]
```

where

- -h is an optional argument that shows an help message and exits;
- -f FILE is the only compulsory argument where FILE is the campaign file to use;
- -n NUM is an optional argument that indicates how many repetition of every experiment should be run (default value is 3);
- --verbose is an optional argument that enables more verbose console output of the script.

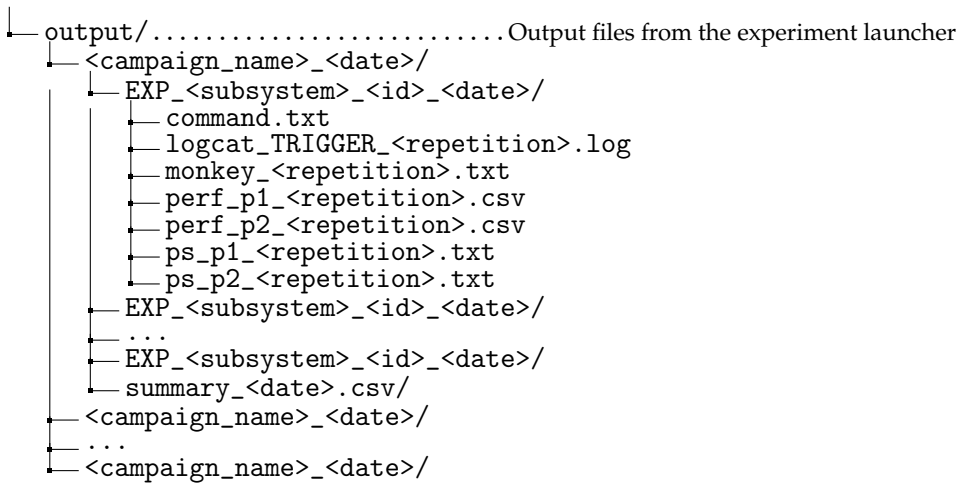


FIGURE 3.16: Output Folder Structure and Files of the Experiment Launcher

The experiment launcher generates several output files organized in a hierarchical structure. Once the experimental campaign ends, an output folder will be generated and/or populated, as follows in Figure 3.16.

Every executed experimental campaign has its own folder inside the output folder. The name of this folder is generated as

```
<campaign_name>_<date>/
```

where:

- <campaign_name> is the name of the campaign file without extension;
- <date> is the text-formatted date and time when the experimental campaign started (formatted as %Y%m%d_%H%M).

For each line of the campaign file, one or more (according to the NUM parameter) experiments are executed and their outputs are saved in the folder

```
EXP_<subsystem>_<id>_<date>/
```

where:

- `<subsystem>` represents the subsystem where the injection is performed in;
- `<id>` is the line number of the campaign file where the performed injection is specified;
- `<date>` is the text-formatted date and time when the first repetition of the experiment started (formatted as `%Y%m%d_%H%M`).

Inside this folder, there are all the outputs files of all the repetitions of a single experiment. They are:

- `command.txt`: is the line of the campaign file that indicates the injected fault producing this outputs;
- `logcat_TRIGGER_<repetition>.txt`: the logs from Android logcat, one file for each repetition;
- `monkey_<repetition>.txt`: the output of the monkey tools used in phase 1, one file for each repetition;
- `perf_p1_<repetition>.csv`: comma-separated values of performance data during phase 1, one file for each repetition;
- `perf_p2_<repetition>.csv`: comma-separated values of performance data during phase 2, one file for each repetition;
- `ps_p1_<repetition>.txt`: the output of the `ps` command executed on the Android device immediately after phase 1, one file for each repetition;
- `ps_p2_<repetition>.txt`: the output of the `ps` command executed on the Android device immediately after phase 2, one file for each repetition;

where `<repetition>` is an integer representing which repetition the files belong to (from 0 to $NUM - 1$).

These outputs are further analyzed to assess whether the injection succeeds and what are the consequences of the fault on the Android OS. During the test execution, a first analysis is performed on the logcat. The potential test outcomes, and the criteria used to obtain them, are:

- crash: a native process or a user app has crashed due to the injected fault, and the system logs a message reporting a “FATAL EXCEPTION”;
- ANR: a user app is stalled due to the injected fault, and the system generates a log message that reports an ANR condition (*i.e.*, Application Not Responding [90]);
- fatal: a high-severity error is raised by the Android OS, and the system generates log messages with a high-severity level (*i.e.*, either *assert* (A) [91] or *fatal* (F) [92];
- no failure: the Android OS is robust against the injected fault, and no significant effect is perceived.

3.4 Experimental Evaluation

This section describes how we performed an experimental evaluation on three high-end smartphones from three different vendors: Huawei P8, Samsung Galaxy S6 Edge, and HTC One M9, running Android 6.0 (Marshmallow).

We performed a fault injection campaign with AndroFIT, by targeting the 14 components in the six subsystem (Subsection 3.2.2). The implementation of the AndroFIT faultload, from the Android fault model, altered the number of actual faults to inject. On one side, we removed all the faults derived from Android 5 (Lollipop) or Android 7 (Nougat). On the other, the same output value fault in the Android fault model generates several actual faults in the AndroFIT faultload, each with a different corruption on the data³.

A summary of the results of the fault injection campaign can be found in the tests outcomes, presented in Table 3.4.

The close analysis of the experiments validates the accuracy of the AndroFIT suite. We carefully checked that the intended faults were actually injected by the tool. For example, in the case of availability faults, we found in the logs that the expected exception indeed occurred (*e.g.*, the

³the mutation operators used in AndroFIT are the same operators used by the Android Fuzzer Chizpurple, further presented in Chapter 5

ActivityNotFoundException raised by the Activity Manager). In the case of timeliness faults, we have looked at the responsiveness of the device during the tests. For example, when we inject delays in the start activity method of the Activity Manager Service, we noticed that the apps indeed take several seconds more than usual before starting, and that in some cases the UI freezes. In the case of corruption faults, we looked at the logs and found messages that told us about the corruptions. For example, when we inject corruptions at reads and writes on APK files, we found error messages by the Package Manager about incorrect APK metadata. For each subsystem, some relevant failure scenarios are presented and deeply analyzed. All the scenarios are fully reproducible and mostly belongs to a single device, *i.e.*, the Huawei P8, unless otherwise specified.

3.4.1 Fault Injection in the Phone Subsystem

For fault injection in the phone subsystem, we performed 309 experiments. Results, in Figure 3.17, presented 22 failures for Samsung, 78 failures for HTC, and 114 failures for Huawei. It is clear that, among the three vendors, the Huawei devices produced the highest number of failures. Most of these failures were “fatal errors” signaled by the phone subsystem, and, in the case of Samsung and Huawei, a few cases in which native processes crashed (mostly, the RILD).

failure scenario #1

This failure scenario considers the injection of faults between the RILD and the baseband driver and processor. AndroFIT intercepts the AT messages flowing from the baseband processor to the RILD; and corrupts them by dropping the event codes and their parameters.

The effects of fault injection are shown in Figure 3.18. The corruptions cause an incorrect internal state of the RILD, and cascade effects on the phone services, such as isms and phone_huawei, which crash. In turn, the telephony registry service crashes. Thus, the device is not able to manage phone events anymore. This failure impacts on the end-user, which is unable to perform phone calls. Even worse, the user is not informed about the problem, and the phone application becomes not responsive: when the phone stock application sends commands on behalf of the user, the

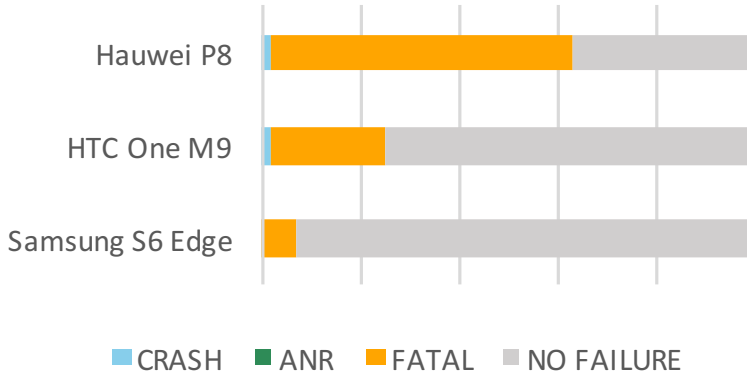


FIGURE 3.17: Fault Injection Campaign Outcomes for the Phone Subsystem

commands are simply ignored by the phone subsystem, without showing any information regarding the unavailability of the phone system.

potential reliability improvements

The presented failure scenario involves several components, and points out several opportunities for improving reliability.

The first, most important effect of the fault is the incorrect internal state of the RILD that causes the crash of phone services. Instead, it would be important for the RILD recognize violations of the AT protocol, and to handle these worst-case situations. These violations should be detected at run-time by adopting defensive programming practices, such as by checking at every step that the messages exchanges with the baseband processor follow the expected protocol. Moreover, the phone services should also be programmed defensively, by recognizing out-of-order events, and avoiding to crash in the case of these errors.

Another opportunity of improvement is in the Huawei phone stock application. It would be advisable to have mechanisms to detect that the phone subsystem is not responsive, for example by using a timeout when waiting for a response. Moreover, the application could trigger a soft restart

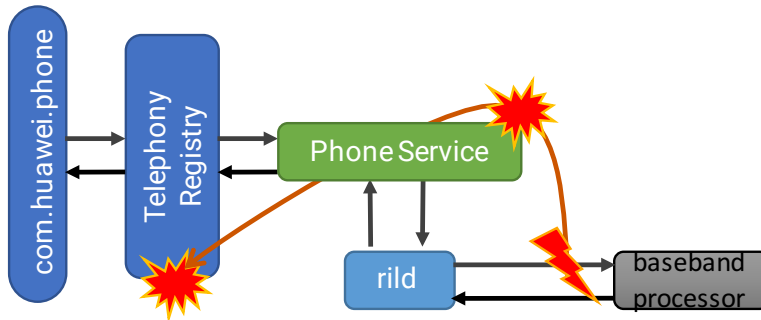


FIGURE 3.18: Analysis of the Failure Scenario #1

to mask the error state and to retry the failed operation. The phone app should also avoid to not provide any feedback to the user, since the user would have the perception of the lack of control over the device, and could get frustrated by the unsuccessful attempts to repeat the operation. Thus, in the case that these recovery mechanisms are not effective, the phone app should at least inform the user about the problem with the phone subsystem.

3.4.2 Fault Injection in the Camera Subsystem

For fault injection in the camera subsystem, we performed 111 experiments. Results, in Figure 3.19, presented 34 failures for Samsung, 19 failures for HTC, and 60 failures for Huawei. Again, the Huawei device resulted to be the most fragile, as denoted by the highest number of failures among the three vendors. Most of these failures were process crashes (mostly, crashes of the Huawei camera stock application). In few cases, the camera system reported fatal errors.

failure scenario #2

This failure scenario considers the injection of faults in the Camera Service. The Camera Service is part of the standard Android framework, and it is accessed by both third-party and stock applications. This service is exposed through a Binder API interface. This experiment injected failures of the

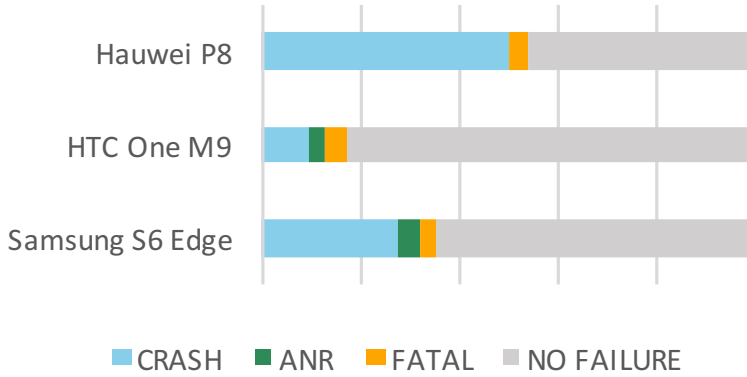


FIGURE 3.19: Fault Injection Campaign Outcomes for the Camera Subsystem

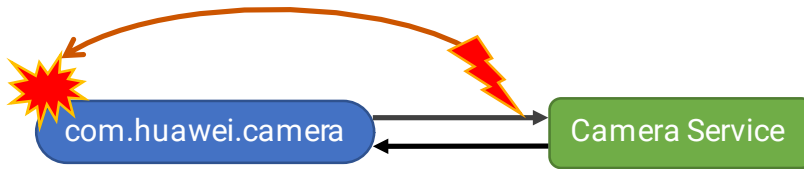


FIGURE 3.20: Analysis of the Failure Scenario #2

Camera Service by forcing the *takePicture* method to return an error to the caller.

The effects of fault injection are shown in Figure 3.20. In this scenario, the error code returned by the method generates a run-time exception. This exception is not handled by the Camera stock application, thus the Camera application is aborted by the Android Runtime. A black screen appears to the user, then followed by a pop-up message that reports the process abort. The Camera application is not restarted. However, this message does not provide any meaningful information to the use, and thus may give a bad perception of the reliability of the device.

failure scenario #3

This failure scenario considers the injection of faults between the Camera HAL and the Camera driver and hardware. In particular, the faults were injected when the media server process attempts to read from the `/dev/video` virtual device file, by forcing the operation to return an error, such as `ENOMEM` or `ENODEV`.

The effects of fault injection are shown in Figure 3.21. In this scenario, the propagated errors lead the media server to fail with a crash. It seems that the media server is not able to handle a corner case triggered by the fault injection: in the logcat, we found a fatal error message *method not implemented* logged by the media server, in the Camera HAL. The crash of the media server causes the crash of the Huawei camera stock application, since the app is not able to handle the exceptions raised by the unavailability of the media server. It is interesting to analyze how this scenario is handled by the HTC One M9 device. This is showed in Figure 3.21. In the HTC device, the camera stock application is programmed to catch the exception from the Camera Service. After the crash of the media server, both the media server and the camera app are quickly restarted, without showing any error to the user. Thus, it is able to mask the fault to the user, and to provide a better perception of device reliability.

potential reliability improvements

These scenarios unveiled noticeable failure effects (black screens, cryptic error messages) to the end-user. Thus, it is advisable for Huawei developers to further check these behaviors, and to mitigate them if possible. The analysis of the scenarios highlight that the missing exception handling by the camera stock application is a good candidate for reliability improvement. This is confirmed by the analysis of the HTC device, in which the stock app is able to catch the exception, and to mask the fault through a soft restart of the camera subsystem.

3.4.3 Fault Injection in the Sensors Subsystem

For fault injection in the sensors subsystem, we performed 108 experiments. Results, in Figure 3.22, presented 21 failures for Samsung, 16 failures for

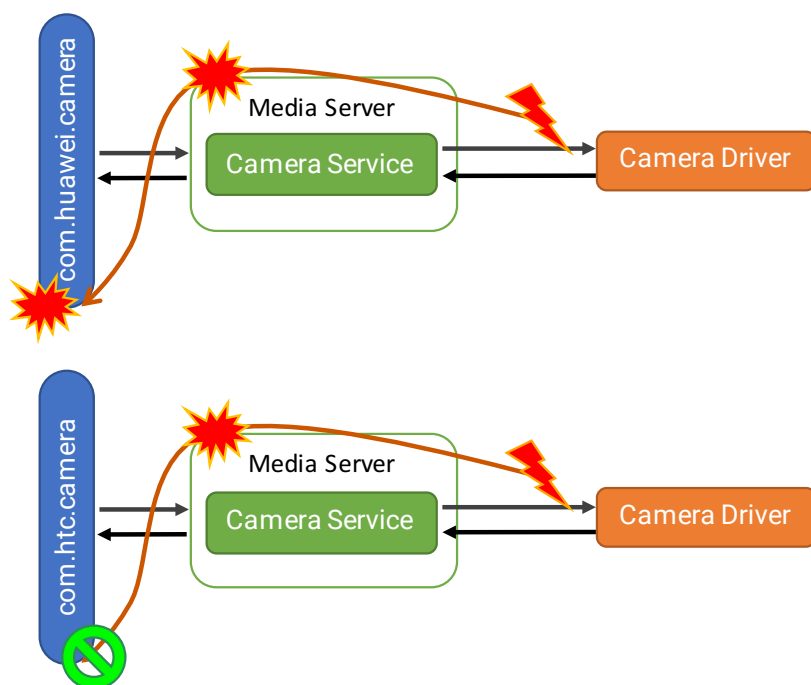


FIGURE 3.21: Analysis of the Failure Scenario #3

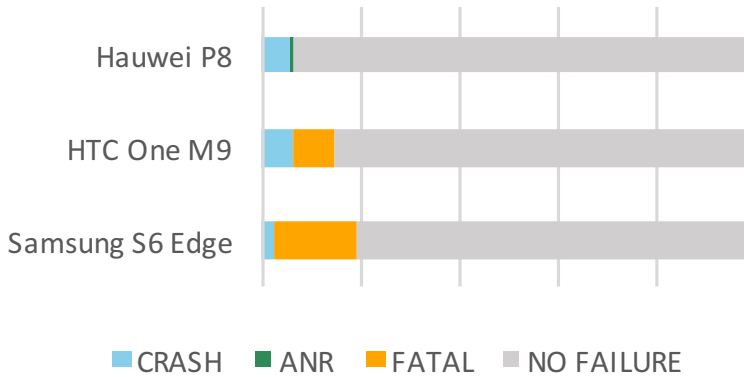


FIGURE 3.22: Fault Injection Campaign Outcomes for the Phone Subsystem

HTC, and 7 failures for Huawei. In these tests, the Huawei device was the most robust, since it exhibited the lowest number of failures, even if the numbers are very similar across the vendors. However, the Huawei devices exhibited an ANR failure that did not happen in the other devices.

failure scenario #4

This failure scenario considers the faults injected when the sensors service attempts to access the sensor devices through virtual device files (e.g., `/dev/sensor_hub`). The experiments injected errors, such as `ENOMEM` on I/O system calls.

The effects of fault injection are shown in Figure 3.23. These errors caused the crash of the sensor service. This crash has severe consequences on the system server process, which also crashes. In turn, this causes the termination of other Android services that execute inside the system server process⁴. Most notably, the failure of the system server process affects the Package Manager Service, and it causes cascading failures of the apps that require special permissions, such as Maps and Contacts.

⁴the sensors service executes within a thread of the system server process.

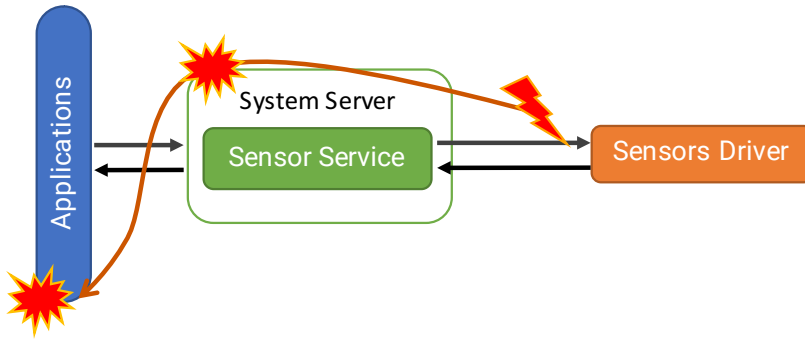


FIGURE 3.23: Analysis of the Failure Scenario #4

potential reliability improvements

This failure scenario is an example of complex problem propagation across several parts of the Android framework. In this case, the main vulnerability is the co-existence of several services inside the system server process. Thus, a fault in any service can potentially impact on all the other services. However, it is not simple to fix this design since it is rooted in the design of the Android OS. Thus, for improving reliability, it is important to avoid failures of these services at all costs, in order to prevent failures of the whole system server process. In particular, it is advisable that the Sensors Service check the successful outcome of the I/O operations on the devices. If there is any I/O error, the Sensors Service should catch the error, and should gracefully handle it and avoid the crash.

3.4.4 Fault Injection in the Activity Subsystem

For fault injection in the activity subsystem, we performed 66 experiments. Results, in Figure 3.24, presented 42 failures for Samsung, 51 failures for HTC, and 58 failures for Huawei. The number of failures has been very high for all of the three devices. These failures froze the system ui and other apps (including stock apps, such as the camera apps), which did not respond to the inputs of the users. In particular, these freezes have been caused by injected delays on key method of the Activity Manager Service (e.g., bind service).

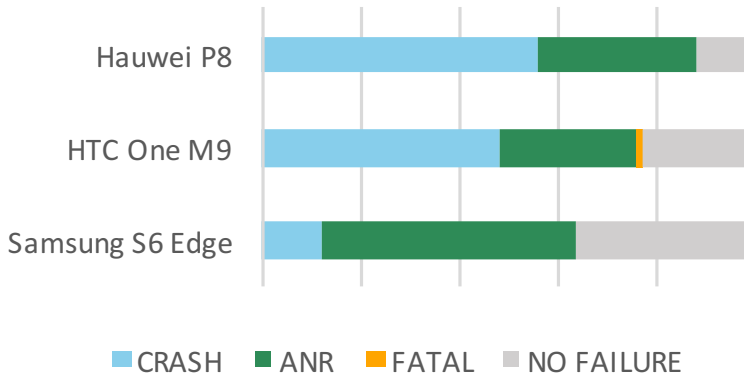


FIGURE 3.24: Fault Injection Campaign Outcomes for the Activity Subsystem

failure scenario #5

This failure scenario has low-severity, but it can still provide some improvement feedback and it is useful to understand the fault injection approach. In this scenario, a service availability fault is injected in the start activity method of the Activity Manager Service. AndroFIT forces the method to return an error code, *i.e.*, -1 .

The start activity method is mainly called by the system ui process. When a fault is injected, the system ui is unable to start a new activity. In this case, the system ui process catches the error code, and it shows a notification that tells the user *Application Not Installed*. This behavior is only a minor annoyance for the user, but it can mislead them, since the application is actually installed and the Activity Manager failed for some other reason. Thus, it would be more reasonable to display to the user a more generic error message.

failure scenario #6

This failure scenario is a case of unresponsive user interface (UI). It is important to avoid stalls of the UI, since they are clearly noticed by the end-users, and since they negatively affect the user perception of reliability.

In this scenario, AndroFIT injected a timeliness fault in the stop activity method of the Activity Manager Service. The timeliness fault delays the execution of the stop activity method by several seconds. This kind of faults can be experienced by the user as a failure because the device is overloaded, or because of a performance bug.

The system ui process becomes not responsive. If the user tries to leave the current activity, the system ui process invokes the stop activity method, but it does not care whether the operation has been delayed or whether the current activity is still open. As a result, even if the user taps on the quit activity button several times, the UI remains stuck. Clearly, this is undesirable behavior. Instead, it would be necessary the the system ui process should enforce a timeout on the operation, and detect the stall. Then, the system ui should attempt a recovery action, such as to force the termination of the activity by other means. Another possible action is to inform the user that the operation is taking more time than expected, and to invite them to be patient for more time. Another effect of the injection is the force restart of the system server process. If the user presses the show activities button, the Activity Manager Service will crash, bringing down the whole system server process.

potential reliability improvements

The injection of timeliness faults pointed out that the system ui process can often get stuck if it does not receive a timely response from the system server process. This is an important problem since the stall of the UI is clearly noticed by the user. The stalls are caused by the fragile behavior of the system ui process, which waits for a response for an indefinite amount of time, without enforcing a timeout. This is due to the fact that the system ui process excessively relies on the responsiveness of the system server process. However, this excessive trust on the system server process can expose the user to stuck UIs. Thus, it would be important to introduce additional countermeasures to handle these worst-case situations when they might happen. In particular, when possible, the developers should adopt asynchronous interactions with the system server process: that is, the system ui process should not block waiting for a response (a synchronous interaction), but it should be able to continue its execution, and to check whether the requested operation has actually been completed.

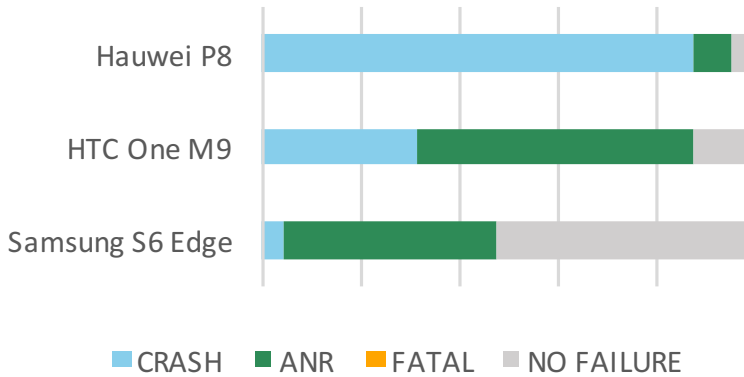


FIGURE 3.25: Fault Injection Campaign Outcomes for the Package Subsystem

3.4.5 Fault Injection in the Package Subsystem

For fault injection in the phone subsystem, we performed 63 experiments. Results, in Figure 3.25, presented 30 failures for Samsung, 45 failures for HTC, and 60 failures for Huawei. The numbers are similar to the fault injection tests in the activity subsystem. Also similar hang failures showed up when injecting delays on Package Manager Service key method, such as resolve intent.

failure scenario #7

This failure scenario presents a case of stuck UI. The stall is caused by a timeliness fault injected in the resolve intent method of the Package Manager Service. This method is used by the system to resolve which app component it should start, by reading the contents of an Intent. In this case, the failure happens when the user presses the show activities button on the bottom part of the UI. When the resolve intent method is injected with a delay, the whole UI becomes not responsive. It does not show the list of the current activities, and do not provide any feedback to the user. Even retrying to press the button does not solve the stall. Thus, it would be useful to detect

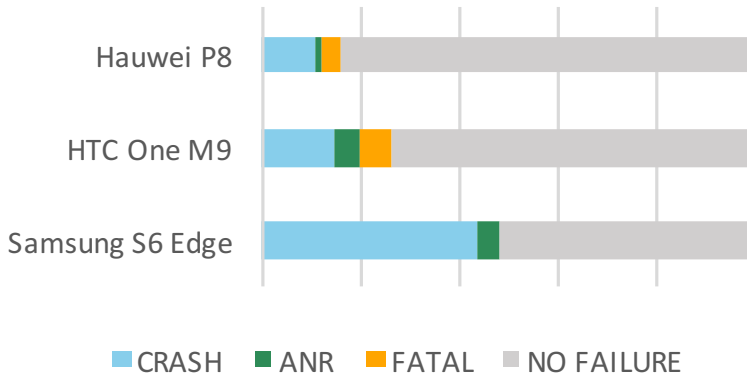


FIGURE 3.26: Fault Injection Campaign Outcomes for the Storage Subsystem

this kind of stall of the UI, in order to avoid that such worst-case scenarios lead to a poor user experience.

potential reliability improvements

This failure scenario proposed another stall of the device UI, and it actually enforce the suggestions provided in the previous subsection.

3.4.6 Fault Injection in the Storage Subsystem

For fault injection in the storage subsystem, we performed 75 experiments. Results, in Figure 3.17, presented 36 failures for Samsung, 20 failures for HTC, and 12 failures for Huawei. The Samsung device has been the one that failed the most. In most of these failures, the system processes failed because of unhandled exceptions and errors that were raised during filesystem I/O and SQL queries.

failure scenario #8

This failure scenario is unveiled when the Package Manager uses the Bionic Library to read information about apps, from the APK files of the app on the

storage. In this scenario, AndroFIT injects a failure of I/O library functions of the Bionic Library, *i.e.*, open and read. When these functions are invoked, the tool corrupts the contents of data buffers.

There are two potential cases of crashes:

- the Package Manager Service crashes in the middle of the get package info method, with the error *Package Manager has died*;
- the Package Manager service crashes because of a failure of the Android Runtime, with the error message

```
art/runtime/indirect_reference_table.cc:76] Check failed:
table_mem_map_.get() != nullptr ashmem_create_region failed
for 'indirect ref table': Not a type-writer
```

which is associated with the POSIX error code *ENOTTY*.

failure scenario #9

This failure scenario shows how the system server process crashes because of a fault in the SQLite Library. The system server process uses SQLite to store and to retrieve persistent information about the configuration of the device and about the user. In particular, the Lock Setting Service is a service that keeps the lock pattern or password data and related setting for each user. The Lock Setting Service performs the database query

```
SELECT value FROM locksettings WHERE user=? AND name=?
```

and, during this operation, AndroFIT injects an availability fault in the sqlite step operation of the SQLite Library. The fault forces the operation to return an error code (*i.e.*, `SQLITE_ERROR`).

The JNI wrapper around the SQLite library throws an exception. Unfortunately, the Lock Setting Service is unable to handle this exception, causing a fatal failure of the system server process.

potential reliability improvements

In the case of corrupted APK files, the Package Manager Service should isolate the fault only for the application affected by the corruption. Thus, the app should be aborted, or should not be started at all, without affecting

other apps. This would require to carefully check that the contents read from the APK are not corrupted, by performing checks that the data are valid. For example, by checking that strings have weird characters or are too long, or by checking that integer variables should have values within a reasonable range.

In the case of SQL queries, the system server process and the stock apps should catch any exception that might occur, and they should avoid the crash by masking the exception. In the specific case of the Lock Settings Service, the device should inform the user an alternative way to unlock the device. For example, by asking for a different PIN or password. Another approach could be to store and reuse a previous version of the database in the case of problems.

3.4.7 Lessons Learned

In this subsection, we discuss about the fault model used in the AndroFIT suite. We base the discussion on the experimental results presented in the previous subsections. The purpose of this discussion is to point out which fault modes were useful to give feedback to improve reliability, and which faults were not effective. This information will help practitioners in future efforts to perform fault injection tests on mobile devices.

According to the experimental results, the *availability* faults (*i.e.*, exceptions and error codes returned by APIs, such as Binder calls, library calls and system calls) were the ones that found vulnerabilities in the highest number of Android subsystems. In these vulnerabilities, the Android OS lacked exceptions or errors handlers, thus the exception/error was able to spread and cause the failure of Android services and applications. Since so many Android subsystems were vulnerable to these faults, it is recommended to always include this fault type in fault injection test plans. Another advantage of this fault type is that it can provide clear and easy suggestions for improving reliability: they point out the specific exceptions/errors that are not tolerated, thus the developers can mitigate them by implementing the missing exception/error handlers. This is especially important in the stock applications, as they must provide user-friendly feedback in the case of faults, in order to give a good perception of the reliability of the device.

The *timeliness* faults (*i.e.*, delays and stalls of API calls) were another frequent cause of failure of the Android OS. In particular, when the application invokes the service in a synchronous way (*i.e.*, the application stops until the service is provided), the target service causes the failure of stock apps and of the System UI. The synchronous approach is a cause of performance bottlenecks for the application, and it can cause failures if the API is delayed or stalled. The experiments tell us that the timeliness faults are effective when they are injected on the Binder APIs of Android services, since stock applications are often vulnerable to this type of faults. Moreover, the delays/stalls of UI applications must be avoided since are clearly noticed by the user, and would cause a poor quality of experience. In order to make the applications more robust against these faults, they should either adopt an *asynchronous* approach to call the service (by allowing the application to continue to be responsive even if the call is delayed/stalled); or the applications should enforce a timeout to detect the long execution time of the service, and retrying the operation, or aborting the operation and informing the user in a friendly way.

The *output value* faults (*i.e.*, a service returns wrong data, that deviates from the correct data) were effective for some specific components (the RILD socket, the AT channel and the Bionic library). For these components, the incorrect data were not correctly handled by the Android components, and caused the crash of key services. These findings point out that the corruption of protocols (such as the AT protocol) and formats (such as the APK format, and the transaction format in surface streams) can expose the Android OS to failures. Indeed, it is difficult for developers to build robust protocol/format parsers that could manage any invalid data in the protocol/format. Therefore, we recommend that output value faults should be injected into protocols and formats that are complex and tricky to parse/handle robustly. We found that even a simple approach (such as injecting random noise in these protocols/format) can be effective to highlight vulnerabilities.

Instead, we found that other components (*e.g.*, the Camera Service, the Sensors Service, and the Activity Manager) are quite insensitive to corruptions, since these services do not expose complex protocols/formats. In these cases, the injection corrupted the input/output parameters of the services (for example: in the Camera Service, parameters such as

`whitebalance=auto` are replaced with incorrect values, and numeric values are corrupted with 0, negative, MAX, or random values; in the Activity Manager, the methods return Intents with an incorrect Action field, such as `ACTION_BATTERY_CHANGED` replaced by `ACTION_POWER_CONNECTED`, or a truncated Data URI). In other cases, such as SQLite Library, the corruptions caused the SQL query results to be truncated. These injections can affect individual applications by corrupting their output (*e.g.*, the Camera application can return distorted images, or a background app service may not be loaded); but these injections do not affect the stability of the Android OS and stock apps (neither fatal exceptions nor ANRs occurred).

The resource management faults (*e.g.*, the exhaustion of memory, the inability to open files or create threads) were effective to find vulnerabilities in processes and components in the native layer. Since these parts are written in C/C++, they do not benefit from robust and automated resource management as it would be the case for the Java language, and thus they are often vulnerable to resource-related problems. Thus, we advise to inject resource management faults for testing the robustness of components and processes in the native layer. Examples of this are the RILD process and the Media Server (which hosts the Camera Service), as we found that these processes were affected by failures in the case of resource unavailability.

TABLE 3.3: Fault Injection Techniques and Target Components Map

	Binder IPC Hooking	Library Hooking	System Call Hooking	UNIX Socket Hijacking	UNIX Signaling
RILD		X	X	X	X
Baseband Driver and Processor		X	X		
Camera Service	X				X
Camera HAL		X			
Camera Driver and Hardware		X	X		
Sensors Service and HAL	X			X	X
Sensors Drivers and Devices		X	X	X	
Activity Manager Service	X				X
Package Manager Service	X				X
SQLite Library		X			
Bionic Library		X			
Volume Daemon	X			X	
Mount Service	X				X
Storage Drivers and Devices		X	X		

TABLE 3.4: Summary of the Fault Injection Campaign Outcomes

	subsystem	CRASH	ANR	FATAL	# of experiments
Samsung Galaxy S6 Edge	phone	0	0	22	309
	camera	31	5	3	111
	sensors	3	0	18	108
	activity	8	34	0	66
	package	3	27	0	63
	storage	33	3	0	75
		78	69	43	732
HTC One M9	phone	6	0	72	309
	camera	11	3	5	111
	sensors	7	0	9	108
	activity	32	18	1	66
	package	20	35	0	63
	storage	11	4	5	75
		87	60	92	732
Huawei P8	phone	6	0	108	309
	camera	56	0	4	111
	sensors	6	1	0	108
	activity	37	21	0	66
	package	55	5	0	63
	storage	8	1	3	75
		168	28	115	732

Chapter 4

Software Aging Analysis of the Android Mobile OS

When something is important enough, you do it even if the odds are not in your favor.

— Elon Musk

Software aging is the common phenomenon of gradual accumulation of errors that induces to a progressive performance degradation, and eventually to failure. Long-running software systems are the most vulnerable to software aging, such it is Android. The contributions of this work are:

- an experimental methodology, based on the Design of Experiments (DoE) approach [93] and including several statistical techniques, to investigate the software aging phenomenon in Android OS;
- the design and execution of an experimental campaign where devices from four different vendors (*i.e.*, Samsung S6 Edge, Huawei P8, LG Nexus, and HTC One M9) were stressed and highlighted that software aging does exist in Android, does depend on vendor customizations, but does not vary significantly across Android versions.

4.1 Overview

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

A common problem, whose impact on end-user quality perception is often underestimated by engineers, is **software aging** [58]. Software affected by the so-called aging-related bugs (ARBs) suffers from the gradual accumulation of errors that induces to a progressive performance degradation, and eventually to failure [23, 25, 94]. Due to such a *subtle* depletion, ARBs are difficult to diagnose and debug during testing: they appear only after a long execution and under non-easily reproducible triggering and propagation conditions. Typical examples include memory leakages, fragmentation, unreleased locks, stale threads, data corruption, and numerical error accumulation, which gradually affect the state of the environment (*e.g.*, by consuming physical memory unjustifiably). The typical solution is to try figuring out the temporal trend of the degradation, in order to act by preventive maintenance actions known as **rejuvenation**, *i.e.*, solutions to clean and restore the degraded state of the environment [58, 95, 96].

The problem is known to affect many software systems, ranging from business-critical to even safety-critical systems [15–18, 21, 22, 26, 59]. Software aging in the Android OS can potentially affect the user experience of millions of mobile products. Therefore, we conduct an experimental study to extensively investigate if and how software aging impacts the performance and reliability of mobile devices based on Android.

To investigate the phenomenon, we designed and ran a controlled experiment, grounding on a series of long-running tests, where devices from four different vendors (Samsung, Huawei, LG, and HTC) were stressed and monitored under various configurations with the aim of highlighting possible aging phenomena, to understand the conditions when they occur more severely, and to diagnose their potential source.

4.2 Experimental Methodology

To analyze software aging issues in Android, we adopt an experimental methodology based on stress testing. A stress test exercises a system with an intensive workload for a long period (typically, several hours), in order to increase the likelihood that software aging effects, such as memory leaks, accumulate over time [17, 18, 59, 97].

Moreover, we perform tests under several different conditions, as the extent of software aging effects (*e.g.*, the rate at which the device experiences performance degradation or resource depletion) varies depending on how the system is configured and exercised [59, 97]. For example, in the context of Android, different user apps may have a different impact on software aging, as they may trigger different services of the Android OS; or different Android configurations (*e.g.*, vendors or versions) may exhibit different software aging effects. However, considering all of the possible combinations of workloads and configurations leads to an extremely high number of long-running experiments, which would take an unfeasible amount of time to complete.

In order to address the problem of managing the several parameters of the tests (*e.g.*, workload, device vendor, and OS version), we adopt the **Design of Experiments** (DoE) approach [93] and derived the methodology on the basis of our preliminary study [60].

We define a set of factors (*i.e.*, the parameters of a test) and their possible values (called levels in the DoE) for designing a test plan for the Android OS. First, we identify the feasible combinations of Android devices and Android versions, since we can not install all the versions to all the devices. Then, we define a blocked, full-factorial design with regards to the other factors. Moreover, we introduce response variables to quantify the impact of a test on the target device in terms of software aging, and correlate the factors with the response variable to identify the most influential ones. We consider both **user-perceived** response variables and **system-related** response variables, which respectively reflect the responsiveness of the device as perceived by the user, and the depletion of system resources that may cause aging-related failures.

We analyze the experimental data using statistics techniques, such as:

- Mann-Kendall test to statistically assess if there is a monotonic trend in a series of the variable of interest over time [16];
- Sen's procedure to compute, in a non-parametric way, the slope of a trend [98, 99];
- Spearman's rank correlation coefficient to analyze the statistical dependence between two variables of interest [100];
- Analysis of Variance (or ANOVA) and Kruskal-Wallis/Wilcoxon hypothesis test to analyze whether the differences among two sets of experiments are statistically-significant (*i.e.*, not simply due to random variations) [101, 102];

4.2.1 User-Perceived Response Variable

To quantify software aging as perceived by users, we focus on the responsiveness of the Android OS, as it is a key design goal of this mobile OS. For example, an early design goal had been to cold-start a basic application, up to a responsive GUI, within 200 ms at most [85, 103]. Therefore, we quantify the user-perceived responsiveness by measuring the **Launch Time** (LT) of Android activities (*i.e.*, an application component that provides a GUI screen). The LT is the period between the request to start an Activity, and the appearance of the Activity on the screen, including the initialization of background and foreground elements.

We measure the LT by analyzing the logs from the Activity Manager of the Android OS, which is the service responsible for instantiate new activities and to switch among them by saving and restoring their state. The Activity Manager logs the event that triggers the start of a new Activity (denoted by the `ActivityManager` tag and the keyword `Displayed`), including the time spent for starting the Activity. We collect these logs using the Android Logcat utility [91, 92].

To get periodical samples of the LT during the experiments, we periodically (at a low frequency, every minute) terminate and restart the user applications that are used as workload. These apps need to be terminated since, otherwise, the Android OS would cache the Activities (*i.e.*, if the app has been started recently, its Activities are retrieved from a cache when the user switches again on that app) and prevent the start of new Activities,

thus providing us no information about the responsiveness of the system. Moreover, by periodically restarting the apps, we avoid that software aging effects (such as leaked memory) could accumulate inside the apps, since our focus is not to study aging of Android apps, but rather the software aging effects in the underlying Android OS.

The following line is an example of log message that shows the `MainActivity` Activity from the application `com.example.myapplication`, which took 100 ms to complete its initialization:

```
I/ActivityManager(1097): Displayed com.example.myapplication/.MainActivity: +100ms
```

After an experiment, we analyze the LT to identify any degradation of responsiveness. Ideally, if the device is free from software aging, the average LT should not vary across the experiment, since we keep fixed the workload and the test conditions during the experiment. However, we expect that software aging gradually manifests its effects during the experiment, by continuously degrading the LT of the workload apps.

To analyze the LT, we produce a time series for each experiment using the LT samples of all activities collected during the experiment, and we apply the non-parametric Mann-Kendall (MK) statistical test to check whether the time series exhibits a trend [16]. This statistical test checks the null hypothesis that there is no monotonic trend in the time series, and provides a level of significance, *i.e.*, p-value, for the likelihood that the null hypothesis is actually true in the time series. If the p-value is lower than a given α , we can reject the null hypothesis with probability, namely with a confidence, greater than $(1 - \alpha)$, which points out that the LT has been affected by a trend. We require that the confidence should be higher than 90% ($\alpha = 0.1$). Moreover, in the case that the LT exhibits a trend, we obtain the slope of such trend by applying the Sen's procedure [98,99], which is a non-parametric, robust technique that fits a linear model and computes the rate at which the samples increase over time. It simply computes the slope as the median of all slopes between paired values, and it is insensitive to outliers. This approach is often adopted in software aging studies where the system is stressed under fixed conditions, which is likely to lead to a fixed degradation rate (if any) [17,18,59,97].

4.2.2 System-Related Response Variables

To get more insights about software aging effects, we collect additional metrics that reflect resource utilization inside the Android OS. These system-related metrics include

- the memory utilization, which is the resource most exposed to software aging issue due to memory management bugs, and a scarce one for mobile devices;
- the CPU utilization, which is also exposed to software aging, *e.g.*, due to algorithmic bugs that waste CPU time on bloated data structures; and
- the garbage collection duration, which is a critical activity for the efficient use of memory.

In our analysis, we will analyze these system-related metrics to point out which are the most stressed areas of the Android OS that might be causing software aging.

Memory

We focus on memory utilization since many previous experiments on software aging effects demonstrated that this resource is the most affected one and tends to have the shortest time-to-exhaustion (TTE) [16–18, 24, 59, 97]. The Android OS uses elaborated mechanisms to manage memory, by automatically handling the lifecycle of apps (*e.g.*, collecting resources once an app is not used for a long time), by recycling processes (*e.g.*, when starting a new Activity), and by managing memory inside applications based on the ART (Android Run-Time) Java environment. Another potential cause of aging effects in memory utilization is represented by the complexity of the Android OS services, such as Activity Manager and Package Manager, that are persistent and may accumulate aging effects over time.

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

analyze memory consumption of each process of the Android OS, by periodically collecting (every 30 seconds) its **Proportional Set Size** (PSS), *i.e.*, the footprint of the process on the physical RAM (*e.g.*, not including parts of the process that do not consume memory, such as program code that has not been executed and that still resides on the storage). We focused on this metric because our previous results show that it is strongly correlated to performance degradation trends [60].

We check again whether LT degradation is related to per-process PSS metrics, by looking for trends and by checking whether these trends are correlated to LT degradation trends. For each PSS series, we perform the following two steps: (i) we test the presence of a trend (and compute its slope) using the MK test and the Sen's procedure; (ii) we compute a correlation measure between the slopes of the metric and the slopes of the median LT trend, across all experiments, using the non-parametric Spearman's rank correlation coefficient [100], since it is robust to outliers and does not make restrictive assumptions on data, contrarily to the parametric counterparts. The correlation points out whether a trend of the metric is systematically accompanied by a degradation trend of the LT.

Garbage Collection

Garbage collection (GC) is a key component of modern programming environment, as it manages dynamic memory allocations in place of the programmer (*e.g.*, freeing unused area) in order to avoid memory management bugs. However, despite it, there can still be residual software aging effects: if unused objects are still referenced by the program (*e.g.*, due to poor object handling by programmers), the GC is not able to dispose of the objects, which can accumulate over time [20,59,104]. Moreover, memory fragmentation (*e.g.*, the use of several small objects instead of a large one) and other bad programming practices (*e.g.*, frequently re-allocating objects that could instead be reused) can impact on GC and significantly degrade the performance perceived by users. If GC takes too long, the application can be *frozen* for short periods or be noticeably slowed down during GC. Therefore, we include the duration of GC among system-related metrics.

We collect information on GC from the logs of the Android OS, marked with the art tag. The ART reports on GC only in the case that the GC takes much more than usual (in particular, when the GC Pause Time exceeds 5

ms, or the GC Duration exceeds 100 ms). In such case, the log includes the event that triggered the GC (*e.g.*, the GC has been triggered in background, or it was explicitly invoked by the program as in the case of some Android OS services); the GC algorithm (as the ART support more than one); the amount of time spent for the GC; the amount of objects freed by the GC; and the available heap memory. We collect these logs as soon as they appear over the course of the experiments. These ART logs denote cases of slow GC, which are relevant for our analysis. We expect that intensive workloads, such as the ones used by our stress tests, can highlight the effects of poor memory management in Android components, which in turn can result in degraded performance.

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

CPU and memory utilization at task level

The Android OS adopts a complex multi-process and multi-threaded architecture to run its several services and components (*e.g.*, to manage a specific hardware resource or provide an API). However, the previous metrics provide information about processes, but they do not provide specific information about individual threads inside a process. This is a limitation for analyzing the Android OS, as Android runs most of its basic services (*e.g.*, camera, audio, and phone) as threads inside few processes (*e.g.*, system server and media server processes) [78].

Therefore, we introduce additional metrics to get more insights about the activity of individual services running inside threads. In the underlying Linux kernel, both processes and threads are represented as tasks. A multi-threaded process consists of a set of tasks that share the same resources (*e.g.*, virtual address space and open files). Therefore, we analyze CPU and memory utilization metrics for individual tasks. These metrics point out which tasks are mostly active during the onset of software aging effects, and are a potential root cause of software aging.

We obtain task-level metrics from proc filesystem of the Linux kernel. In particular, we use the virtual files `schedstat` and `stat` files that are exposed

by the kernel (in the directory `/proc/TASK_PID/`) to provide information on scheduling and memory usage of each task. These metrics include the number of minor and major page faults (*i.e.*, the task requires new code or data, thus denoting higher memory activity), and execution time spent in user-space and kernel-space, which respectively point out the CPU and I/O activity of the task. We periodically sample these task-level metrics (every 30 seconds).

To identify critical tasks, we compute trends for each metric and for each task using the MK test and the Sen's procedure. Then, we count the number of cases in which a metric exhibited a statistically-significant trend for the task, at a confidence level of 90%. The higher the count, the higher the likelihood that the metric evolves with software aging effects, thus revealing a potential relationship between a task and software aging of the device.

4.2.3 Factors and Levels

We consider several factors to cover different configurations and workloads in the experimental plan. We define 5 factors and obtain the test plan by applying the DoE on the levels of these factors. Factors and levels are summarized in Table 4.1.

In our analysis, we assess whether these 5 factors contribute to the severity of software aging, in order to provide context about which conditions are more problematic. We apply the one-way ANOVA technique [101] to assess which factors impact the response variable in a statistically-significant way. In order to use a non-parametric ANOVA and be robust to potential non-normal distribution of errors, we use the Kruskal-Wallis/Wilcoxon hypothesis test [102]. The null hypothesis, in this case, is that the factor does not impact the response variable, and the p-value indicates again the confidence in rejecting this hypothesis. We conclude that a factor impacts the response variable if the level of confidence is higher than 90%, *i.e.*, the p-value is less than 0.1.

Device (DEV)

Experiments are performed on different Android devices from different vendors, each with its own software configuration and customizations. The

TABLE 4.1: Factors and Levels for Android Software Aging Analysis

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

Android devices in our experimental setup represent the levels for the DEV factor. We conducted experiments on high-end smartphones from four different vendors; thus, we have four levels for the DEV factor, labeled as *HTCONEM9*, *HUAWEIP8*, *LGNEXUS*, and *SAMSUNGS6EDGE*.

Version (VER)

The Android devices can execute different versions of the Android OS. The Android OS versions available for a device determine the levels for the VER factor. It is worth noting that not every level in the DEV factor can be combined with every level in the VER factor, because some devices do not support older or newer versions of the Android OS (for example, some devices may only support Android 5 and 6, while other may only support Android 6 and 7). In total, we have three levels for the VER factor: *ANDROID5*, *ANDROID6*, and *ANDROID7*.

Application Set (APP)

In our experiments, we use different sets of applications as workload to exercise the Android OS. These apps are selected to be representative of typical usage scenarios (including browsing, making photos, dialing, chatting), and counts of both stock apps and third-party apps. We include popular Android applications, which are listed Table 4.1. These apps have been installed on all Android devices used in the experiments. We organized applications in two groups, which represent the two levels of the APP factor: European applications (*EU*), and Chinese applications (*CHINA*), which are obtained respectively from the European version of the Google app market, and from Chinese app markets.

Workload Events (EVENTS)

Our workload generator (based on the Android *monkey* tool) produces a set of events to interact with the apps and the Android device. The events include: application switch, touch, motion, trackball, and navigation events. The events are generated randomly, and their probability of occurrence is configured by the EVENTS factor, that varies across three levels: *MIXED1*, where half of the events are application switches; *MIXED2*, where half of

the events are touches; *MIXED3*, where half of the events are navigation events. In every level, the other half of the events are of the remaining types, and are selected according to a uniform random distribution.

Storage Space Usage (STO)

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

4.2.4 Experimental plan

We defined an experimental plan by considering different combinations of the levels and factors presented in the previous subsection. In turn, the experimental plan can be divided in three sets. The full experimental plan includes 72 experiments, for a total of 18 days of experimentation. All the experiments are listed in Table 4.2 and sorted from the oldest to the newest Android version. The three test plans are blocked full-factorial designs, in which one factor is fixed (*i.e.*, the Android device or version, according our research questions), while we vary all the other parameters.

TABLE 4.2: Experimental plan of the case study

ID	DEV	VER	APP	EVENTS	STO
EXP1	HUAWEIP8	ANDROID5	EU	MIXED1	NORMAL
EXP2	HUAWEIP8	ANDROID5	EU	MIXED1	FULL
EXP3	HUAWEIP8	ANDROID5	EU	MIXED2	NORMAL
EXP4	HUAWEIP8	ANDROID5	EU	MIXED2	FULL
EXP5	HUAWEIP8	ANDROID5	EU	MIXED3	NORMAL
EXP6	HUAWEIP8	ANDROID5	EU	MIXED3	FULL
EXP7	HUAWEIP8	ANDROID5	CHINA	MIXED1	NORMAL
EXP8	HUAWEIP8	ANDROID5	CHINA	MIXED1	FULL
EXP9	HUAWEIP8	ANDROID5	CHINA	MIXED2	NORMAL
EXP10	HUAWEIP8	ANDROID5	CHINA	MIXED2	FULL

Continued on next page

Table 4.2: Experimental plan of the case study – *continued from previous page*

ID	DEV	VER	APP	EVENTS	STO
EXP11	HUAWEIP8	ANDROID5	CHINA	MIXED3	NORMAL
EXP12	HUAWEIP8	ANDROID5	CHINA	MIXED3	FULL
EXP13	HUAWEIP8	ANDROID6	EU	MIXED1	NORMAL
EXP14	HUAWEIP8	ANDROID6	EU	MIXED1	FULL
EXP15	HUAWEIP8	ANDROID6	EU	MIXED2	NORMAL
EXP16	HUAWEIP8	ANDROID6	EU	MIXED2	FULL
EXP17	HUAWEIP8	ANDROID6	EU	MIXED3	NORMAL
EXP18	HUAWEIP8	ANDROID6	EU	MIXED3	FULL
EXP19	HUAWEIP8	ANDROID6	CHINA	MIXED1	NORMAL
EXP20	HUAWEIP8	ANDROID6	CHINA	MIXED1	FULL
EXP21	HUAWEIP8	ANDROID6	CHINA	MIXED2	NORMAL
EXP22	HUAWEIP8	ANDROID6	CHINA	MIXED2	FULL
EXP23	HUAWEIP8	ANDROID6	CHINA	MIXED3	NORMAL
EXP24	HUAWEIP8	ANDROID6	CHINA	MIXED3	FULL
EXP25	HTCONEM9	ANDROID6	EU	MIXED1	NORMAL
EXP26	HTCONEM9	ANDROID6	EU	MIXED1	FULL
EXP27	HTCONEM9	ANDROID6	EU	MIXED2	NORMAL
EXP28	HTCONEM9	ANDROID6	EU	MIXED2	FULL
EXP29	HTCONEM9	ANDROID6	EU	MIXED3	NORMAL
EXP30	HTCONEM9	ANDROID6	EU	MIXED3	FULL
EXP31	HTCONEM9	ANDROID6	CHINA	MIXED1	NORMAL
EXP32	HTCONEM9	ANDROID6	CHINA	MIXED1	FULL
EXP33	HTCONEM9	ANDROID6	CHINA	MIXED2	NORMAL
EXP34	HTCONEM9	ANDROID6	CHINA	MIXED2	FULL
EXP35	HTCONEM9	ANDROID6	CHINA	MIXED3	NORMAL
EXP36	HTCONEM9	ANDROID6	CHINA	MIXED3	FULL
EXP37	LGNEXUS	ANDROID6	EU	MIXED1	NORMAL
EXP38	LGNEXUS	ANDROID6	EU	MIXED1	FULL
EXP39	LGNEXUS	ANDROID6	EU	MIXED2	NORMAL
EXP40	LGNEXUS	ANDROID6	EU	MIXED2	FULL
EXP41	LGNEXUS	ANDROID6	EU	MIXED3	NORMAL
EXP42	LGNEXUS	ANDROID6	EU	MIXED3	FULL
EXP43	LGNEXUS	ANDROID6	CHINA	MIXED1	NORMAL
EXP44	LGNEXUS	ANDROID6	CHINA	MIXED1	FULL
EXP45	LGNEXUS	ANDROID6	CHINA	MIXED2	NORMAL
EXP46	LGNEXUS	ANDROID6	CHINA	MIXED2	FULL
EXP47	LGNEXUS	ANDROID6	CHINA	MIXED3	NORMAL
EXP48	LGNEXUS	ANDROID6	CHINA	MIXED3	FULL
EXP49	SAMSUNG56EDGE	ANDROID6	EU	MIXED1	NORMAL
EXP50	SAMSUNG56EDGE	ANDROID6	EU	MIXED1	FULL

Continued on next page

Table 4.2: Experimental plan of the case study – *continued from previous page*

ID	DEV	VER	APP	EVENTS	STO
EXP51	SAMSUNGS6EDGE	ANDROID6	EU	MIXED2	NORMAL
EXP52	SAMSUNGS6EDGE	ANDROID6	EU	MIXED2	FULL
EXP53	SAMSUNGS6EDGE	ANDROID6	EU	MIXED3	NORMAL
EXP54	SAMSUNGS6EDGE	ANDROID6	EU	MIXED3	FULL
EXP55	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED1	NORMAL
EXP56	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED1	FULL
EXP57	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED2	NORMAL
EXP58	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED2	FULL
EXP59	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED3	NORMAL
EXP60	SAMSUNGS6EDGE	ANDROID6	CHINA	MIXED3	FULL
EXP61	SAMSUNGS6EDGE	ANDROID7	EU	MIXED1	NORMAL
EXP62	SAMSUNGS6EDGE	ANDROID7	EU	MIXED1	FULL
EXP63	SAMSUNGS6EDGE	ANDROID7	EU	MIXED2	NORMAL
EXP64	SAMSUNGS6EDGE	ANDROID7	EU	MIXED2	FULL
EXP65	SAMSUNGS6EDGE	ANDROID7	EU	MIXED3	NORMAL
EXP66	SAMSUNGS6EDGE	ANDROID7	EU	MIXED3	FULL
EXP67	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED1	NORMAL
EXP68	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED1	FULL
EXP69	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED2	NORMAL
EXP70	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED2	FULL
EXP71	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED3	NORMAL
EXP72	SAMSUNGS6EDGE	ANDROID7	CHINA	MIXED3	FULL

The first set (EXP13~EXP60) covers all of the DEV levels, and keeps the VER factor to *ANDROID6*, since Android 6 Marshmallow is the only version that can be installed on all the devices, allowing us to study the impact of software aging across devices from different vendors (and all other factors with the same level). The second set of experiments (EXP1~EXP24) fixes the DEV factor to *HUAWEIP8*, and varies the VER factor between *ANDROID5* and *ANDROID6*. The third set (EXP49~EXP72), instead, fixes DEV to *SAMSUNGS6EDGE* and the VER to either *ANDROID6* or *ANDROID7*. These last two sets are used to study the impact of software aging across different versions of the Android OS. In each set, with the sole exception of the fixed factor, we consider every possible combination of the levels, leading to a full factorial design. Based on our preliminary experiments [60],

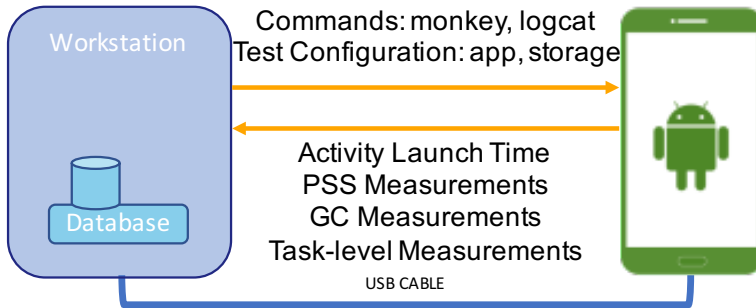


FIGURE 4.1: The Experimental Android Testbed

we calibrated the duration of each experiment to 6 hours, as this duration suffices to point out software aging effects.

The devices are controlled and monitored using the Android Debug Bridge (ADB) utility (which is a non-intrusive, dedicated channel through the USB port for debugging purposes). User inputs are provided with the monkey tool, which is a workload generator that randomly generates UI events. The events are generated at a high frequency ($500ms$) to stress the device, and follow the random profile of the *EVENTS* factor. The experimental testbed is showed in Figure 4.1.

4.3 Results

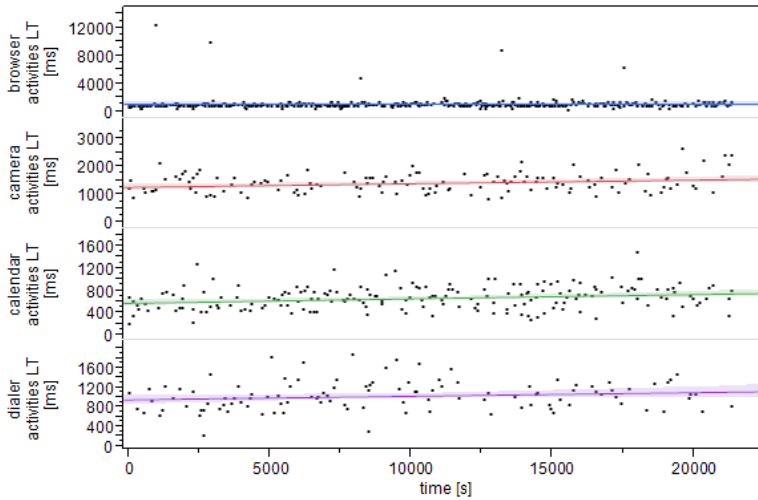
We analyze software aging phenomenon using the metrics and the experimental plan presented in the previous section. Thus, we conclude the analysis with a more detailed study of software aging symptoms.

4.3.1 Software aging across Android vendors

This subsection analyzes software aging across device vendors, by fixing the Android OS to version 6.

Analysis of Launch Time

The Launch Time is a direct indicator of software aging effects experienced by the user. The analysis shows that most of the experiments (33 out of 48)

FIGURE 4.2: Groups Activities Launch Time for *EXP39*

presents a statistically-significant aging trend in the LT series from all the activities started during the experiment. On average, the All Activities LT trend across all the experiments has been $1.76\text{E-}2$ ms/s (with an estimated degradation of 380ms, on average, of the launch time after 6 hours of testing), with a maximum of $1.19\text{E-}1$ ms/s in the worst case (estimated degradation of 2.5 seconds after 6 hours). Moreover, at the end of some experiments, the device were so bloated to be unusable, as they reacted to user inputs with very long delays (e.g., several seconds). Figure 4.2 provides examples of LT trends for a subset of activities from the experiment *EXP39*, in which the activities have been divided among browser, camera, calendar, and dialer.

We performed the one-way ANOVA technique on the experiments with the Android version fixed to 6, to assess whether the differences between the samples are statistically significant, and which are the factors that impact on the observed **LT trends**. These factors are showed in Figure 4.3, which provides the LT trends that were measured in this first subset of experiments. According to the ANOVA, the main factor that determines statistically-significant differences in the Launch Time is the device vendor (*DEV*), with a confidence of 99%. We found that the experiments with *HUAWEIP8* yielded the lowest trends, while the *SAMSUNGS6EDGE* yielded the highest ones. Another statistically-significant difference is in the *APP* factor, with a

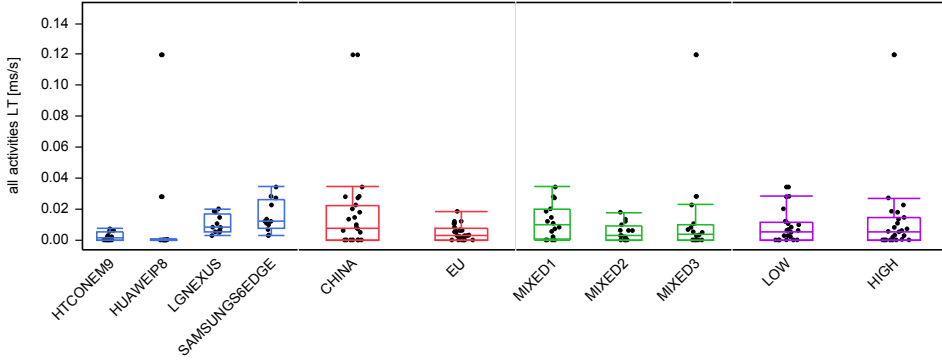


FIGURE 4.3: Distribution of the Launch Time Trends, with all vendors and fixed to Android 6 (EXP13~EXP60)

confidence of 85%, where the *CHINA* applications have a worse impact than *EU* in terms of LT. This result suggests that software aging in the Android OS depends on the workload, which can stress different services and sub-systems of the Android OS depending on user applications; moreover, the customizations from the Android vendors have also influence on software aging.

Analysis of Memory Usage

While the LT give indication of software aging effects directly perceived by users, the memory usage provides more insights about the underlying cause of these issues, since memory often suffers from leaks, fragmentation, and thrashing [104]. Based on the results of our preliminary work [60], we focus the analysis on the *PSS* metric collected for four key processes of the Android OS, namely the *System Server*, *Media Server*, *System UI*, and *Surface Flinger*. These processes play an important role in the Android OS:

- The *System Server* is the first Java process that starts at Android OS boot and initializes the rest of the Android Framework. It runs the majority of system services, such as the *Activity Manager*, which manages the life cycle of applications and their activities, and the *Package Manager*, which manages installed packages and security permissions.

- The *Media Server* is the process that host most of the media related services, e.g. *Audio Flinger*, *Media Player Service*, *Camera Service*, and *Audio Policy Service*.
- The *System UI* is the process that composes notifications, device status, and device navigation buttons as system bar elements in specific screen areas.
- The *Surface Flinger* process receives window layers (surfaces) from multiple sources (*System UI* included), combines them, and displays them on the screen.

We again performed the one-way ANOVA, using the PSS of these processes. Figure 4.4 shows the distribution of PSS trends from the experiments. We found that these processes exhibit increasing trends of the PSS over the experiments. The *System Server* is the process with the highest trends. Moreover, the *DEV* and the *APP* factors exhibit statistically-significant differences of the the *System Server*, with a confidence of 99%: these trends are especially high in the case of the *SAMSUNGS6EDGE*, and of the group of *CHINA* apps. Instead, the *EVENTS* and *STO* factors do not have a statistically-significant impact.

However, the results for the remaining processes (*Media Server*, *Surface Flinger*, *System UI*) must be interpreted with caution, as in some cases they even exhibit negative trends. Instead, the trends for the *System Server* were always positive. This behavior to the use of media (e.g., playing videos or using the camera) by the workload, which require these processes to temporarily allocate more memory: if the random workload uses media more in the first part of the experiment, these processes will also use more memory in the first part rather than the last part, leading to an apparent decreasing trend, regardless of software aging phenomena.

We cross-check this interpretation of the results by jointly analyzing the *PSS* and the *LT* metrics. We compared the memory consumption trend of the processes with the corresponding *LT* trends of the experiments, by correlating these two metrics using the *Spearman's rank correlation*. The correlation provides an index that points out whether the two metrics tend to vary in the same way: for example, a positive correlation means that higher values of one metric are accompanied by a higher values of the other one.

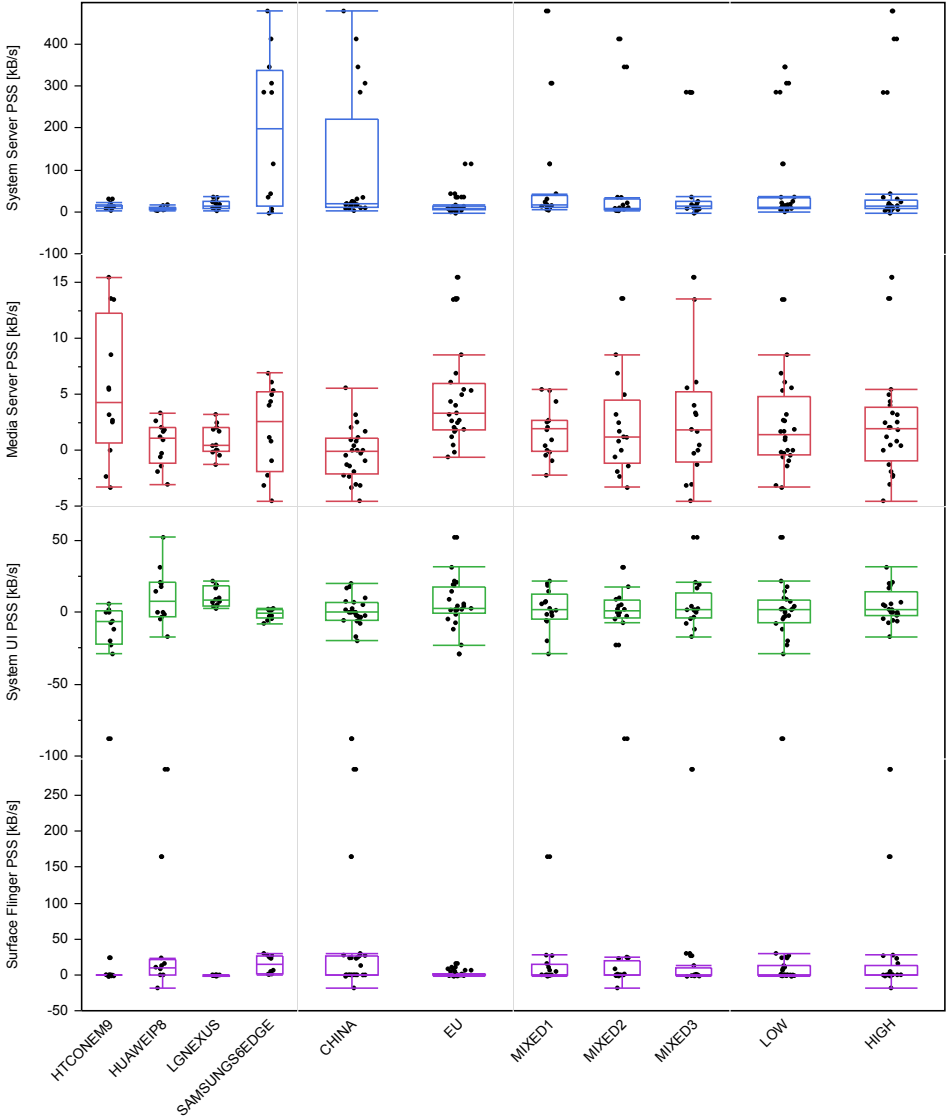


FIGURE 4.4: PSS Trends Distributions: EXP13~EXP60 (Android 6)

TABLE 4.3: Spearman Correlation Coefficients between All Activities LT Trends and PSS Trends of Android System Processes

PROCESS	SPEARMAN COEFFICIENT	P-VALUE
system (System Server)	0.6481	6.0548e-05
mediaserver (Media Server)	-0.5641	0.0009
com.android.systemui (System UI)	-0.0306	0.87
surfaceflinger (Surface Flinger)	0.6125	0.0001

Table 4.3 shows the results of the correlation. Indeed, the memory consumption of the *System Server* exhibits a noticeable (and statistically significant) positive correlation with the LT, meaning that high LT trends (i.e., quicker performance degradation) occur at the same time of high PSS trends (i.e., quicker inflation of the memory consumption). Instead, the other processes exhibit a less significant correlation, which is even negative in two cases. Thus, the memory consumption of the *Media Server*, *System UI*, and *Surface Flinger* does not seem a possible cause of the performance degradation (the LT trends). Instead, the increasing memory consumption of the *System Manager* (which has an important role in starting and managing activities through the *Activity Manager* and *Package Manager*) is a potential symptom of software aging, that we further investigate later in this section.

4.3.2 Software aging across Android versions

We analyze software aging across different versions of the Android OS, by looking for differences both between the Android versions 5 and 6 (by locking the *DEV* factor to *HUAWEIP8*), and between the Android versions 6 and 7 (by locking the *DEV* factor to *SAMSUNGS6EDGE*).

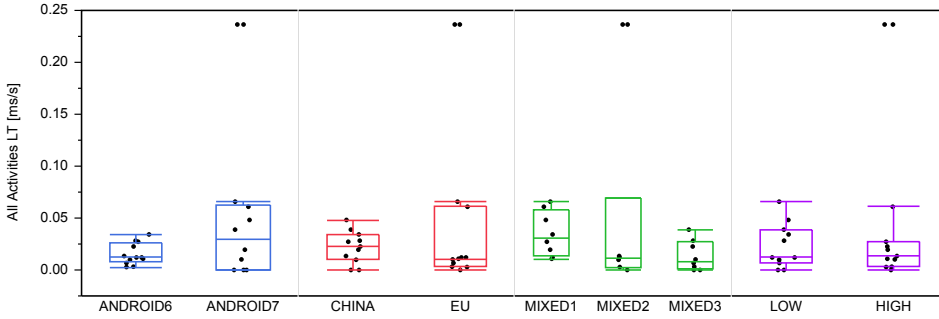


FIGURE 4.5: Launch Time Trends Distributions: EXP49~EXP72 (Samsung S6 Edge)

Analysis of Launch Time

In the case of the *SAMSUNGS6EDGE* device, we again consistently observed aging trends also for *ANDROID7*, which are showed in Figure 4.5. Overall, the average LT trend across all of the *SAMSUNGS6EDGE* experiments has been $3.01\text{E-}2$ ms/s. We estimate that the LT at the end of the tests (i.e., 6 hours) degrades, on average, by 650.89ms compared to the LT at the beginning of the test. The maximum LT had been $2.36\text{E-}1$ ms/s in the worst case, with an estimated degradation of LT of 5.1 seconds after 6 hours of testing.

In the case of the *HUAWEIP8* device, we also notice aging trends in both the versions, as showed in Figure 4.6. The average LT trend across all the experiments has been $1.35\text{E-}2$ ms/s, with an estimated degradation of 291.62ms after 6 hours of testing. The maximum LT trend has been $1.19\text{E-}1$ ms/s in the worst case, with an estimated degradation of 2.6 seconds after 6 hours.

Comparing *ANDROID6* with *ANDROID7*, the LT trends show only small differences with respect to the mean values, and a slightly higher variability of the trends for *ANDROID7*. The differences between *ANDROID5* and *ANDROID6* are apparently more noticeable, with a reduction of the LT trends in favor of *ANDROID6*. We performed the one-way ANOVA on these two sets of experiments, to assess whether the differences between the samples from different Android versions were statistically significant. According to the ANOVA, none of the factors (including the Android OS

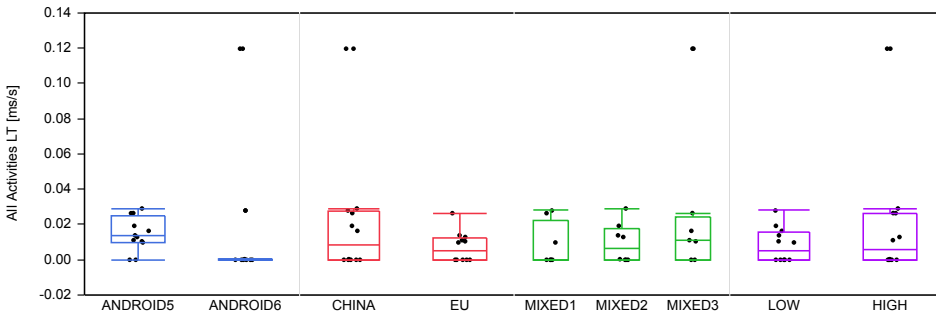


FIGURE 4.6: Launch Time Trends Distributions:
EXP1~EXP24 (Huawei P8)

version) has a statistically-significant impact on the LT trends, both for the *SAMSUNGS6EDGE* and the *HUAWEIP8*. According to these results, we conclude that the software aging effects on the performance neither improve nor worsen across different Android OS releases, as the LT trends do not exhibit significant variations. This result suggests that the revisions to the Android OS are not addressing the areas of the OS that are affected by software aging, and that Android vendors need to invest more effort to address this neglected problem.

Analysis of Memory Usage

Figure 4.7 and Figure 4.8 show the PSS trends for the four key processes, respectively in the case of *HUAWEIP8* and *SAMSUNGS6EDGE*. According to the ANOVA, in all processes, there were statistically-significant differences (with confidence levels greater than 90%) between the trends of different Android OS versions. In particular, in the case of *HUAWEIP8* (i.e., the transition from *ANDROID5* to *ANDROID6*), the PSS trends for the *System Server* process gets worse; instead, in the case of the *SAMSUNGS6EDGE* (i.e., the transition from *ANDROID6* to *ANDROID7*), the PSS trends for the *System Server* exhibit an improvement. Considering the results of the previous analysis on LT trends, it seems that the magnitude of LT trends is not impacted by these variations of the PSS trends (i.e., the LT trends are steady even if the PSS trends are different). This result suggests that it is not the quantity of memory consumption that influences the performance degradation, but

rather the way the memory is used, in terms of fragmentation, frequency of allocations, or the adoption of bad programming practices (§ 4.2.2). We analyze this aspect in more detail in the next sections.

4.3.3 Analysis of process internals

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

Analysis of Garbage Collection

We further analyze Android processes from the point of view of memory management, by considering the time spent for garbage collections, namely the *GC Pause Time* (i.e., the period that the process is suspended during the GC) and the *GC Duration* (i.e., the total duration, including both the GC that executes when the process is suspended, and the GC that executes in parallel with the program).

We performed a trend analysis on these GC metrics for each process. The results were grouped by different collection types [105]: in particular, in our experiments only two GC types produced more than 100 samples and exhibited a trend with confidence higher than 90%, namely:

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

To investigate how much these processes are affected by a degeneration of GC activity, we analyzed how often a trend occurs in the GC times of each process. Then, we rank the processes according to the number of experiments in which the process exhibited a trend. Figure 4.9 reports these ranks, focusing on the topmost 5 processes.

This analysis confirmed that the *System Server* is the process that most frequently showed a trend of the GC times. Compared to our preliminary

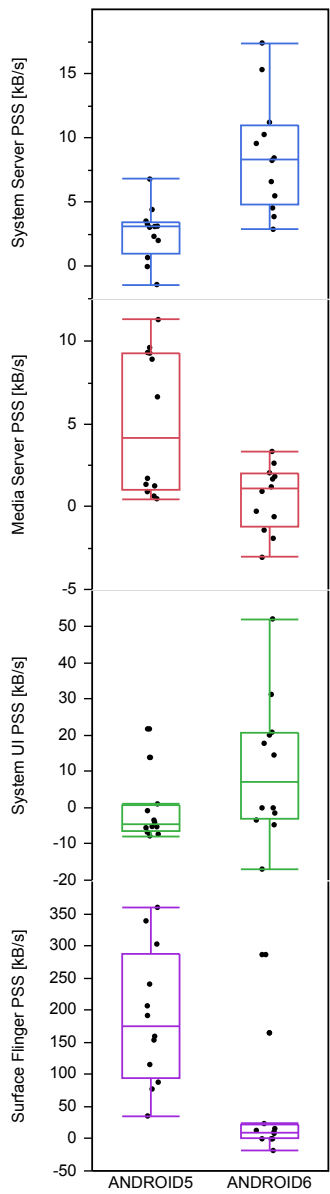


FIGURE 4.7: PSS Trends Distributions: EXP1~EXP24 (Huawei P8)

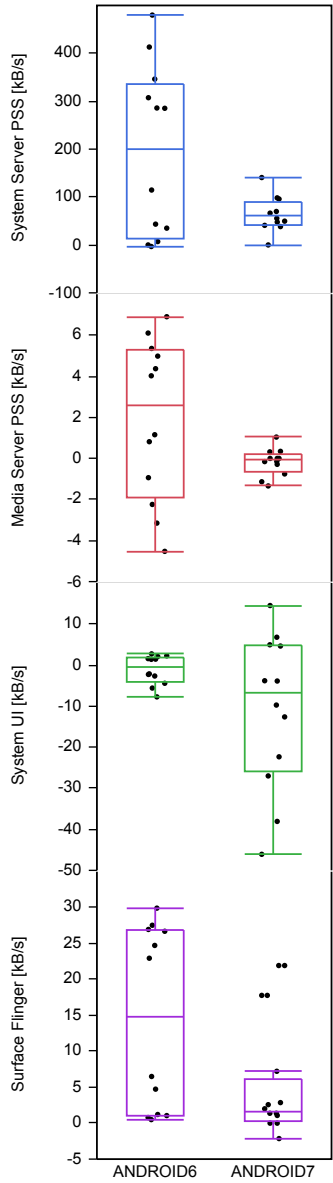


FIGURE 4.8: PSS Trends Distributions: EXP49~EXP72 (Samsung S6 Edge)

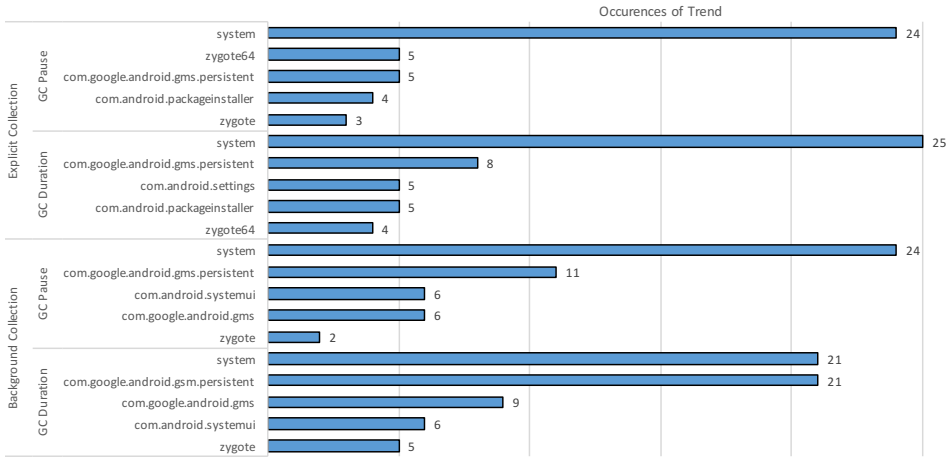


FIGURE 4.9: Occurrences of GC metric trend: EXP1~EXP72

study [60], the *System UI* process has a lower rank, giving space to the `com.google.android.gms.persistent` process, *i.e.*, the persistence layer of the Google Play services. These results suggest that these processes heavily use heap memory, and that they are exposed to performance degradation due to the inflation and fragmentation of the heap, increasing the overhead of garbage collections and slowing down or suspending the threads. Moreover, monitoring the GC times of these processes is another useful indicator to detect performance degradation in the Android OS (*e.g.*, for software rejuvenation purposes).

Analysis of Tasks

We perform a trend analysis on task-level metrics, to identify which tasks of the Android OS exhibit an increasing CPU utilization and cause virtual memory pressure over time (in terms of CPU ticks and minor/major page faults), which are symptoms of software aging that can be attributed to specific tasks. According to the previous analysis, we focus on task-level metrics for the *System Server*, *System UI*, and *Surface Flinger*. For each of these processes and for each experiment, we applied the Mann-Kenall test and Sen's procedure (both with a confidence level of 90%) on the time series of *major faults*, *minor faults*, *kernel time*, and *user time*. Some tasks exhibit a

statistically-significant increasing trend for several metrics, and we rank the tasks according to how many times a trend occurs for the specific task across all of the experiments. To relate the tasks to Android subsystems and to better understand them, we grouped the tasks according to the Android service or subsystem they belong to, according to their names and to our analysis of the Android AOSP source code. For example, the *ACTIVITY* group in *System Server* consists of four threads related to the Activity Manager, namely *ActivityManager*, *ActivityManager_2*, *ActivityManger_3*, and *HwActivityManag*. The value of each group is computed by averaging the trends count of each task of the group. The groups ranked among the top-10 are presented in Figure 4.10.

The most of the occurrences showed up in some specific groups. For example, in the *System Server* we have:

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

These groups and the others in Figure 4.10 represent tasks that are especially stressed during the experiments, and that exhibit higher and higher CPU utilization over time. Software aging effects can either be caused by them (e.g., their subsystem is affected by software aging), or may be propagated through them (e.g., the tasks are slowed down by other subsystems on which they depend on, and that are also causing or propagation aging effects). In both cases, these groups point out areas of the Android OS that may be targeted by software rejuvenation: in particular, it is advisable to focus software rejuvenation in the *System Server*, either at process-level (in

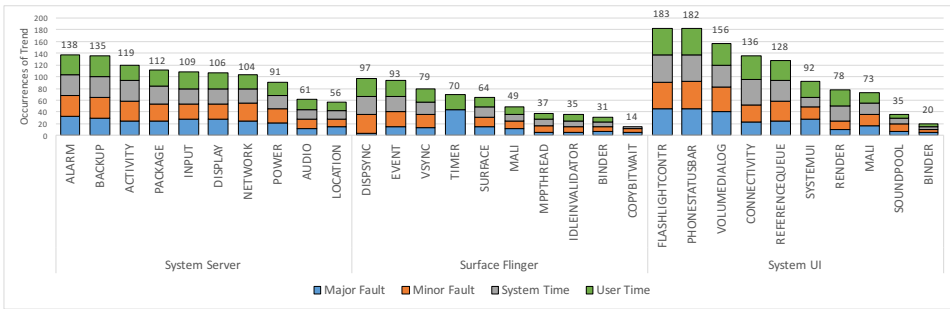


FIGURE 4.10: Occurrences of task metric trend:
EXP1~EXP72

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

Chapter 5

Chizpurfle: A Gray-Box Android Fuzzer for Vendor Service Customizations

An expert is a man who has made all the mistakes which can be made, in a narrow field.

— Niels Bohr

Fuzzing is a well-established and effective software testing technique to identify weaknesses in fragile software interfaces by injecting invalid and unexpected inputs. Fuzzing can be extremely useful in finding software bugs in Android services, particularly in closed-source vendor customizations. The contributions of this work are:

- a novel *gray-box* fuzzing tool for Android devices, namely Chizpurfle, to address the gap in the spectrum of mobile fuzzers, and to improve the effectiveness of fuzzing on vendor customizations;
- an experimental evaluation campaign for Chizpurfle on the Samsung S6 Edge smartphone running Android 7 (Nougat), fuzzing 2272 vendor-specific service methods and finding two bugs, with the realization that Chizpurfle improves the depth of testing compared to the black-box approach, by increasing the test coverage by 2.3 times on average and 7.9 times in the best case.

5.1 Overview

This chapter presents the third aspect of this thesis: fuzz testing. Android comes in different flavors, depending on which vendor is implementing it. Nowadays, more than 20 original equipment manufacturers (OEMs), including but not limited to Samsung, HTC, Huawei, Motorola, and LG, base their devices on the Android Open Source Project (AOSP) [9]. Hardware capabilities are not the only factor that support the customers' choice. Software customizations play a key role in this aspect, making user experience unique and more enjoyable. Unfortunately, these customizations often introduce new software defects, which are vendor-specific. Because they are proprietary, vendor customizations are not integrated in the open-source Android and do not benefit from the feedback loop of the whole ecosystem. Thus, they are less scrutinized than the core AOSP code-base, and their vulnerabilities take significantly more time to be patched. Indeed, the Google Android security team publishes a monthly security bulletin [106] with new and patched security vulnerabilities, but it has to refer the users to vendor-specific security bulletins such as the ones by LG [107], Motorola [108], and Samsung [109]. It is worth noting that vendors customizations consist of code running with special privileges, thus exacerbating the security issues¹.

Fuzzing is a well-established and effective software testing technique to identify weaknesses in fragile software interfaces by injecting invalid and unexpected inputs. Fuzzing was initially conceived as a *black-box* testing technique, using random or grammar-driven inputs [35]. More recently, *white-box* techniques have been leveraging information about the program internals (such as the test coverage) to steer the generation of fuzz inputs, either by instrumenting the source code or by running the target code in a virtual machine [38, 39]. Unfortunately, these tools are not applicable to proprietary Android services, since vendors are not willing to share their source code, and since virtual machine environments (*e.g.*, device emulators) do not support the execution of these proprietary extensions.

This chapter introduces a novel *gray-box* fuzzing tool, namely **Chizpurfle**, to address the gap in the spectrum of mobile fuzzers, and to improve

¹For example, recent devices based on Qualcomm chipsets suffer from a vulnerability in the Qualcomm service API that allows privilege escalation and information disclosure [12].

the effectiveness of fuzzing on vendor customizations. Similarly to recent white-box approaches, Chizpurfle leverages test coverage information, while avoiding the need for recompiling the target code, or executing it in a special environment. The tool has been designed to be deployed and run on unmodified Android devices, including vendor customization of the Android OS. The tool leverages a combination of dynamic binary instrumentation techniques (such as software breakpoints and just-in-time code rewriting) to obtain information about the block coverage. Moreover, Chizpurfle is able to guide fuzz testing only on the vendor customizations, by automatically extracting the list of vendor service interfaces from an Android device.

We validated the applicability and performance of the Chizpurfle tool by conducting a fuzz testing campaign on the vendor customizations of the Samsung Galaxy S6 Edge, running Android 7 (Nougat). It came out that Chizpurfle improves the depth of testing compared to the black-box approach, by increasing the test coverage by 2.3 times on average and 7.9 times in the best case, with a performance overhead that is comparable to existing dynamic binary instrumentation frameworks. Moreover, we discuss two bugs found in privileged services during these evaluation experiments.

Chizpurfle can fit mainly three vendor usage scenarios: vendor may want to apply a lighter approach than white-box fuzzing because of very complex target systems that make it difficult to re-compile instrumented source code; vendor wants to take into accounts all the potential actions an attacker can undertake, demystifying *security through obscurity*; or some vendor extension can be closed-source code from sub-providers. The tool also provides **future research** a platform for experimenting with fuzz testing techniques (such as evolutionary algorithms) based on coverage-based feedback.

5.2 Chizpurfle

This section includes further motivations for the realization of Chizpurfle, and its design and some implementation details.

5.2.1 Motivations

When a vendor delivers a new smartphone on the market, it includes several customizations of the *vanilla* Android, the open source software stack from the AOSP. Unlike the AOSP, customizations are usually closed source and undocumented, and vary among vendors. Vendors' software customizations are focused on three areas:

- device drivers: they support proprietary hardware components of the smartphone;
- stock applications: they are pre-installed on the smartphone along with the default AOSP stock applications;
- system services: they enhance the Android OS with additional APIs for both stock and third-party applications.

We focus on the third type of customizations, *i.e.*, system services, because they usually run as privileged processes (thus, they have a major potential impact on robustness and security); they are directly exposed to (potentially buggy and malicious) user applications; they provide wrappers to lower-level interfaces, such as device drivers; and they represent a large part of vendor customizations.

In order to understand the extent of deployment of vendor customizations, we conducted a preliminary analysis of system services from vendor customizations in three commercial smartphones, namely the HTC One M9, the Huawei P8 Lite, and the Samsung Galaxy S6 Edge. We extracted the services interfaces on the three devices and on their corresponding Android AOSP versions, using the same techniques of the *Chizpurfle* tool (that are further discussed in §5.2.2), and compared the two lists.

Table 5.1 reports the results of this analysis. The first row is the version of the Android Platform running on each device. The second row is the number of services found only on the device, but not in the corresponding AOSP; in the third and forth rows, this number is split between Java and C services. The next two rows refer only to the Java-implemented services, of which we could retrieve the methods signatures through Java Reflection. The fifth row considers the common Java services, present in both AOSP and vendor devices, that have new methods in the vendor version. Finally, the last row shows how many new methods are present in the vendor

TABLE 5.1: Vendors' Smartphone Customizations on System Services

	Huawei P8 Lite	HTC One M9	Samsung Galaxy S6 Edge
Android version	5.0	6.0	7.0
# new services	30	7	82
# new C services	13	2	20
# new Java services	17	5	62
# extended Java services	15	25	52
# new Java methods	325	166	2,272

services that do not exist in the AOSP. Figure 5.1, instead, visualizes the portions of the three smartphones services, split between unmodified AOSP services and vendor customizations (both new and extended services). Our analysis shows that there is a significant number of customized services and vendor-specific methods. Moreover, most of these services execute in the context of privileged processes (such as System Server and Media Server processes), where any failure can have a severe impact the whole OS.

The large vulnerability surface and high privilege of proprietary services motivate the need for specialized tools to evaluate their robustness. To achieve its full potential, fuzz testing needs to guide the generation of inputs according to test coverage, as demonstrated by empirical experience in several security-critical contexts [38, 67]. However, the lack of source code for proprietary services, and the inability to run these proprietary extensions on a device emulator, defy the strategies for profiling coverage that are adopted by existing fuzzing tools.

5.2.2 Design

The Chizpurfle tool architecture is presented in Figure 5.2. It includes six software modules running on the target Android device, that are implemented in Java and C/C++. These modules cooperate to profile the target system service and to generate fuzz inputs according to test coverage. We

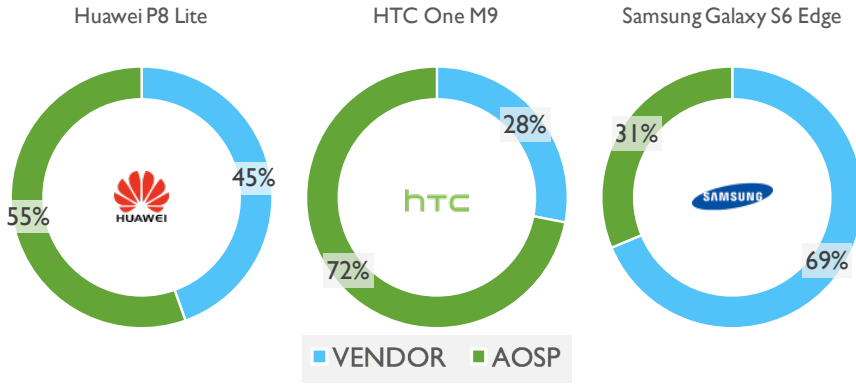


FIGURE 5.1: AOSP and Vendor services.

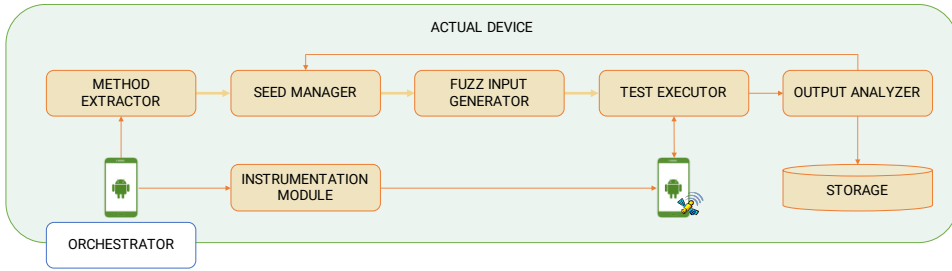


FIGURE 5.2: Overview of the Architecture of Chizpurfle

designed Chizpurfle to be as less intrusive as possible, and to only require root permissions for few debug operations discussed further.

The **Methods Extractor** produces a list of system services and their methods, marking the custom vendor services as described in Section 5.2.1. It also provides a map between services and their hosting processes. The **Seed Manager** iterates over the custom vendor services and methods, and it provides initial inputs (*i.e.*, seeds) for testing them. The **Fuzz Input Generator** takes a seed (either the initial seed, or any previous worthwhile input) and generates new actual inputs for the target method, by applying fuzzing operators to the values of method parameters. Then, the **Test Executor** applies the fuzzed inputs to the target service, while the **Instrumentation Module** keeps track of the test coverage. The outcomes of the test are collected, analyzed, and saved by the **Output Analyzer**. It also provides

feedback to the Seed Manager with seeds for the next test iteration. Finally, the **Orchestrator** provides a simple user interface for Chizpurfle.

Orchestrator

The Orchestrator is the only part of Chizpurfle that runs outside the target Android device (*i.e.*, on the workstation), that loads and controls the other modules using the Android Debug Bridge (ADB) [110] through an USB connection. Chizpurfle minimizes the amount of interactions through ADB, since this connection is notoriously unstable, and we could not rely on it due to potential side effects of fuzzing. Thus, Chizpurfle is detached from the ADB shell process right after it is started, in order to avoid any issue related to the ADB connection. Test data are recorded on a local file on the device and later pulled from the target device by the Orchestrator; the Orchestrator periodically checks the progress of fuzz tests by briefly connecting with ADB and inspecting the logs of Chizpurfle.

We also need to prevent the early termination of Chizpurfle in the case of crashes of system processes. If Chizpurfle ran as a standard Android app, it would be bound to Zygote, which is a daemon process that serves as parent for all Android processes, and which provides a pristine copy of the Android Runtime environment for its children through copy-on-write mechanism. When the Zygote dies, all children processes die as well. Thus, we run Chizpurfle modules in a distinct Android Runtime from the Zygote, that is launched by the `app_process` command (the same command that starts Zygote at boot). This enables Chizpurfle to keep working and gather data even if key system processes fail due to software bugs in vendor customizations.

Method Extractor

The Method Extractor gets the list of services from the Service Manager in a vendor-customized Android device, and it compares them with a blueprint of the AOSP with the same Android version.

The Android OS provides a service-oriented architecture to manage its several services, where the Service Manager keep a list of all the registered services (*cfr.* Section A.3). The Method Extractor queries the Service

Manager on the target device to get the list of all registered services, including customizations. By iterating on these names, it retrieves the list of service descriptors. In case of Java-implemented services (supported by the current version of the tool), a service descriptor is the string name of the Java Interface that is implemented by that system service (e.g., the package manager service implements the `android.content.pm.IPackageManager` Java Interface). Then, Java Reflection API is used to inspect the definition of the interfaces, and to get the signatures of the methods in the service. The methods that are not in the AOSP are marked as *vendor customizations* and considered for testing.

Another task of the Method Extractor is to map every service to the system process that hosts that service. This mapping is obtained by hooking calls to the Service Manager, before the services are registered. In particular, we focus on invocations of the function

```
static int svc_can_register(const uint16_t *name, size_t name_len,
                          pid_t spid, uid_t uid)
```

where `spid` is the PID of the process that wants to register the service named `name`. The functions of Service Manager are hooked by copying a breakpoint handler in the memory address space of the process and by modifying the symbol table to hijack function invocations (the technique to modify the memory of the target process is further discussed in the Instrumentation Module). We force the system services to be published again (thus invoking the Service Manager) by restarting the Zygote process, which in turn forces the restart of system processes and their services. If the method returns 1, then the service has been correctly registered, and the Methods Extractor retrieves the name of the process and saves the mapping.

Instrumentation Module

The Instrumentation Module interacts with the process that runs the target service, in order to collect information about the test coverage. We designed the Instrumentation Module by taking into account the following requirements:

- it must be able to intercept the execution of branches by the target service, in order to identify any new code block covered by the test;

- it has to attach to system processes that are already running, since the life cycle of Android services (including vendors' ones) cannot be directly controlled by external tools such as *Chizpurfle*, and since most of these service are already running since the boot of the target device; and
- it should be able to instrument proprietary services on the actual device (which is the goal of this study), thus excluding any approach that recompiles the source code or that runs in an emulated environment.

We initially explored both hardware and software solutions to measure coverage. Hardware solutions typically take advantage of special CPU features for debugging purposes, such as performance counters. The ARM processors (the CPU family also adopted in Android devices) provide the CoreSight on-chip trace and debug utility to trace the execution of program [111]. However, this specific feature is not mandatory for ARM CPUs, and it is not available on the CPUs typically used in Android devices. Thus, we could not use the hardware support from the CPU, since this solution could not be applied on commercial devices.

We then focused on software-based solutions, which typically have a higher run-time overhead, but they can also provide more flexibility and have less requirements about the underlying hardware. In particular, we based our design on the `ptrace` system call of the Linux kernel: it allows a debugger process (in our context, the Instrumentation Module) to inspect and to write on the memory address space and CPU registers of the debuggee (in our context, the process that runs the target system service). Typically, debugging tools use `ptrace` to install software breakpoints, by replacing an instruction of the debugged program with another instruction that stops the program and triggers a breakpoint handler function.

We leverage the `ptrace` mechanism to profile the target code through dynamic binary rewriting, which is a general technique used by virtual machine interpreters. The program is divided in basic blocks, which are small groups of sequential machine instructions that end with a branch. When the exit branch is reached, the control flow is returned to the interpreter, which retrieves the next basic block, applies some transformations (such as just-in-time compilation and instrumenting the final branch instruction) and moves the control flow to the block; or the exit branch directly jumps to the next basic block if it has already been processed and cached. In our

context, we apply the same principle to keep track of which code blocks are executed, in order to compute the test coverage.

Figure 5.3 shows the instrumentation and tracing mechanism used by Chizpurfle. The Instrumentation Module injects into the target process a small C library by using `ptrace`; then, before restoring the execution of the traced process, it starts a new thread in the process to run the library code, which starts the **stalker server**. This server opens a local socket to talk back with the Instrumentation Module. At the beginning of a test campaign, Chizpurfle sends a message over this socket to enable the tracing of any thread in the target process. Then, the stalker server rewrites the current code block; from this point on, the code blocks will return the control flow to the injected library, which will rewrite the next code block that will be executed by the target. For every rewritten block, the tool adds instructions to log the memory address of the code block, in order to record that the block has been covered. The list of the addresses of covered code blocks is collected by the stalker server in a global data structure. At the end of testing, Chizpurfle sends a message to disable logging, and to let the stalker send back to Chizpurfle the list of code blocks that have been covered.

In the current version of Chizpurfle, we implemented this approach using the Frida framework [112]. Frida is a generic dynamic instrumentation toolkit that provides basic facilities for dynamic binary rewriting, in order to let developers to insert probes in a program for debugging and reverse-engineering purposes. We have ported Frida to 64-bit ARM processors in order to let it run on actual Android devices, and we extended the code rewriting process to trace the coverage of code blocks.

Seed Manager

The Seed Manager is in charge of providing seeds (*i.e.*, initial inputs for the target service) to the Fuzz Input Generator. The Seed Manager manages a priority queue of seeds to be fuzzed, which are ordered with respect to their score π . This score is assigned by the Output Analyzer (as discussed later), after that the seed has been submitted to the target, and that the coverage for the input has been measured. The score π represents the number of new blocks executed by the traced process. If π is greater than zero, the seed is fed back to the Seed Manager to be further fuzzed in subsequent tests.

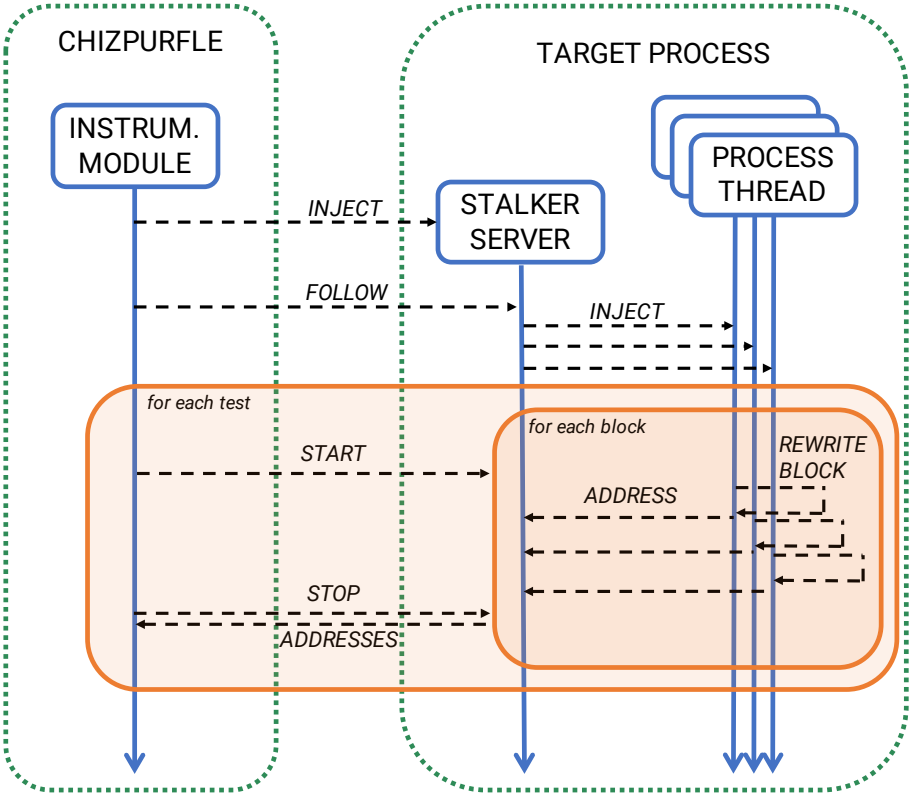


FIGURE 5.3: Chizpurfle Instrumentation and Tracing Mechanism

This workflow represents the cornerstone for applying evolutionary algorithms to drive fuzz testing towards deeper testing of the target service. To select the next seed from the priority queue, we adopt an exploitation-based constant schedule, where a seed is not used more than once [113]. The termination criterion of Chizpurfle is to stop when all seeds have been consumed from the queue, and no more seeds are available for further fuzzing. Moreover, Chizpurfle represents a basis for applying several algorithms for fuzz testing, *e.g.*, by changing or tuning the queue scheduling policy and the termination criterion. This is a valuable opportunity for research on fuzzing in mobile devices, as the heuristics and algorithms adopted by existing tools (such as AFL) have evolved over the years on the basis of empirical experience and experimentation with alternative approaches, which is facilitated by tools such as Chizpurfle.

At the beginning of a fuzz testing campaign for a target method, the Seed Manager creates a new initial seed with empty (for primitive types) or null (for object types) values. This initial seed is not mutated, but immediately submitted as test input. This input will trigger the target method to cover an initial set of π code blocks; then, the input is immediately fed back to the Seed Manager to be used as first actual seed with score π . The steps to fuzz a vendor service method are summarized in Algorithm 1.

Fuzz Input Generator

The Fuzz Input Generator receives a seed to be mutated, and generates inputs for the Test Executor. Several inputs are obtained from the same seed, by applying different fuzz operators. The number of new inputs to generate is proportional to the score π of the seed, and the fuzz operators are selected according to the types of the parameters of the target method. We implemented in Chizpurfle a rich library of fuzz operators, including operators that are often adopted in existing fuzzing tools (including the ones in Section 2.3). For each parameter type, the fuzz operators are:

- Primitive types (boolean, byte, char, double, float, integer, long, short): substitute with a random value, substitute with the additive identity (0), substitute with the multiplicative identity (1), substitute with the maximum value, substitute with the minimum value, add a random

Algorithm 1 fuzzing a vendor service method

Input: Service s , Method m , Process pid

```

1: parameters = createInitialSeed( $s$ ,  $m$ )
2: outputs = executeTest( $s$ ,  $m$ , parameters,  $pid$ )
3: analyzedOutputs = analyzeAndSave(outputs)
4: priorityQueue = {}
5: priorityQueue.push(parameters, analyzedOutputs. $\pi$ )
6: repeat
7:   parameters,  $\pi$  = priorityQueue.pop()
8:   for  $i = 1$  to  $\pi$  do
9:     parameters' = mutate(parameters)
10:    outputs = executeTest( $s$ ,  $m$ , parameters',  $pid$ )
11:    analyzedOutputs = analyzeAndSave(outputs)
12:    if analyzedOutputs. $\pi > 0$  then
13:      priorityQueue.push(parameters', analyzedOutputs. $\pi$ )
14:    end if
15:  end for
16: until priorityQueue == {}

```

delta, subtract a random delta, substitute with a special character (only for char);

- Strings: substitute with a random string, substitute with a very long random string, truncate string, add random substring, remove random substring, substitute random character from string with special character, substitute with empty string, substitute with null;
- Arrays and Lists: substitute with array of random length and items, remove random items, add random items, apply fuzz operator on a item value according to its type, substitute with empty array, substitute with null;
- Objects: substitute with null, invoke constructor with random parameters, apply fuzz operator on a field value according to its type.

For Object types, the Fuzz Input Generator provides additional ad-hoc fuzzers for important specific classes defined by the Android OS. For example, the `android.content.Intent` class has a specific fuzzer that injects into the fields of an Intent (such as actions, categories and extras) special values that have a meaning for the Intent (e.g., `ACTION_MAIN` and `ACTION_CALL` for the Intent actions) [114]; and the fuzzer for the `android.content.ComponentName` class takes into account which components are installed on the target device, in order to use and to mutate valid component names during fuzz testing. For all the other classes, a generic object fuzzer uses the Java Reflection API to create new objects using the class constructor with random parameters, and to invoke setter methods of the class to place random values in the fields of the object.

The Fuzz Input Generator keeps a list of all the inputs generated so far, in order not to submit again the same input to the test executor. Seeds are mutated by using a random number generator to select fuzz operators and to guide them (e.g., new values replacing the previous ones are selected randomly). These probabilities are tunable using a configuration file.

Test Executor

The Test Executor performs tests on the Android device, by invoking the service method with the input provided by the Fuzz Input Generator. It

generates a proxy for that service using the *IBinderObject* associated to the target service. Before invoking the target method, it flushes the logs collected by the Android OS (the logcat, which is a global collector for log messages produced both by user applications and system processes [92]). Then, Chizpurfle sends the start message to the stalker server in the target process and calls the target method. Any potential exception thrown by the service is caught, so that the Test Execution is not aborted in the case of service failures. After the method call, it sends another message to the stalker to stop the tracing, and retrieves logs from the logcat. The steps undertaken by the Test Executor are summarized in Algorithm 2.

Algorithm 2 execute test

Input: Service *s*, Method *m*, Parameters *p*, Process *pid*

Output: Outputs *o*

- 1: flushLogcat()
 - 2: startBranchTracing(*pid*)
 - 3: **try:** call(*s*, *m*, *p*)
 - 4: **catch e:** o.setException(*e*)
 - 5: o.branches = stopBranchTracing()
 - 6: o.logs = stopLogcat()
-

Output Analyzer

The Output Analyzer parses the outputs produced by the Test Executor, and stores the information and results of the tests on a file on the target device.

This component analyzes the logs to identify any failure that has been triggered by the fuzzing test. A failure is detected using the following criteria:

- A/F messages: the system generates log messages with a high-severity level (either assert (A) or fatal (F)) [91,92]; such messages are never generated in failure-free conditions, and should be considered as failure symptoms;
- ANR messages: the system generates a log message that reports an ANR condition (*i.e.*, Application Not Responding) [90]; this condition

denotes that the fuzzed input from the *Test Executor* propagated and triggered a long-running operation or an indefinite wait on the main thread of some process;

- FATAL messages: the system logs a message reporting a “FATAL EXCEPTION”, which denotes an uncaught exception on the service side.

It must be noted that we focus on errors logged by system processes rather than the Test Executor; since the Test Executor stimulates the system service with invalid input, it is correct for the service to raise exceptions and not to provide any service to the Test Executor. Thus, we do not consider these exceptions as failure symptoms as they indicate the correct handling of wrong inputs.

Another check for failure detection is made when the Test Executor retrieves the Binder proxy for the tested service. Chizpurfle registers a callback, using the `linkToDeath` of the *IBinder* API for the service [115], to receive a notification if the Binder object of the service is not available. This happens when the process that hosts the target service dies.

The Output Analyzer component also analyzes the list of block addresses reported by the Instrumentation Module. It keeps trace of all blocks covered by tests so far, and compares them with the block addresses of the current test. If new blocks are detected, the test input is assigned a score π , and the new blocks are added to the list of covered blocks.

The outcomes of this analysis, along with general information about the test inputs and the tested service, are saved on a file. If the input receives a non-zero π score, the input is sent to the Seed Manager for the next iteration of the fuzzing loop. The steps of the Output Analyzer are summarized in Algorithm 3.

Further Optimizations

When we initially applied the Chizpurfle tool to the Samsung Galaxy S6 Edge, we needed to address an important technical problem: the system services (including the ones from vendors’ customizations) execute in the context of a few system processes, along with dozens of other threads, such

Algorithm 3 analyze and save results

Input: Outputs o , DeathRecipient r **Output:** AnalyzedOutput ao

```

1:  $ao = o$ 
2: if ("FATAL" or "ANR" in  $ao.logs.message$ ) or ("F" or "A" in
    $ao.logs.level$ ) then
3:    $ao.hasFailures = true$ 
4: end if
5: if  $ao.deathRecipient.deathNotified$  then
6:    $ao.serviceDead = true$ 
7: end if
8:  $newBranches = ao.branches \ getExecutedBranches()$ 
9: if  $size(newBranches) > 0$  then
10:   $addExecutedBranches(newBranches)$ 
11:   $ao.\pi = size(newBranches)$ 
12: end if
13:  $saveToFile(ao)$ 

```

as the `system_server` process, which contains about 160 threads. Unfortunately, instrumenting all these threads at the same time causes a high overhead, that would slow down the execution of the fuzz tests.

We enabled Chizpurfle to avoid instrumenting threads that are unrelated to the target service being tested. We base this approach on a simple, yet effective heuristic to detect unrelated threads: for all the services running in the context of the same process of the target service, we tokenize the name of the service, and retain the tokens that belong only to that specific service (for example, in the case of `CocktailBarService`, we retain the tokens `Cocktail` and `Bar`); then, we get the names of the threads of the process, using the `comm` entry in the `proc` filesystem; finally, we identify the threads whose name include the tokens of services different than the one under testing (for example, we exclude the `CocktailBarVisi` thread when testing services different than the `CocktailBarService`). The associations between threads and services can be easily reviewed by Chizpurfle's users before starting the testing campaign. This heuristic reduces the run-time overhead of the instrumentation and avoids threads that are likely unrelated to the service

under testing.

We did another minor optimization to avoid few false positives that happened during the tests. During our preliminary tests, some false positives occurred when the Android device reached a low battery level, that caused the Android OS to switch to *battery-saver mode*. This change, together with the workload of fuzz tests, slowed down the smartphone, and caused spurious ANRs in processes not related to the service under testing. We prevented these false positives by periodically checking the battery level and pausing the tests if the level is too low. We carefully checked and reproduced all the other failures described in next sections, to assure that our results are free from false positives.

5.3 Experimental Evaluation

We applied Chizpurfle to a well-known commercial smartphone, the Samsung Galaxy S6 Edge. Before testing, we updated this device with the most recent Android OS officially released by Samsung based on Android 7 (Nougat). First, we perform a fuzz testing campaign on all the service methods introduced by Samsung. Then, we perform additional tests to evaluate the performance overhead and the test coverage, compared to a pure black-box approach.

5.3.1 Bugs in Samsung Customizations

Chizpurfle detected 2,272 service methods from Samsung customizations. In this first experimental campaign, Chizpurfle performed 34,645 tests on these methods. The tool reported that 9 tests caused failures, which are summarized in Table 5.2. We executed again the tests, and we found that the failures were reproducible. Then, we analyzed the failure messages reported on the logs, which include uncaught exceptions and the stack trace at the time of the failures. Despite the source code not being available, we notice that the failures affected high-privilege system processes, and were caused by 2 distinct bugs (respectively, the first 4 failures, and the other 5 failures).

TABLE 5.2: Failures Detected by Chizpurfle

	ID	INPUT	FAILURE
spengestureservice injectInputEvent	7	{0, -2147483648, array of android.view.InputEvent objects with a null item, false, NULL}	FATAL EXCEPTION: mainProcess: com.android.systemui, PID: 12884 java.lang.NullPointerException: Attempt to invoke virtual method 'long an- droid.view.InputEvent.getTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDis- patcher.dispatchInputEventInjection (SmartClipRemoteRequestDis- patcher.java:201)[...]
spengestureservice injectInputEvent	22	{-715676118, -1, array of android.view.InputEvent objects with a null item, false, NULL}	FATAL EXCEPTION: mainProcess: com.android.systemui, PID: 4025 java.lang.NullPointerException: Attempt to invoke virtual method 'long an- droid.view.InputEvent.getTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDis- patcher.dispatchInputEventInjection (SmartClipRemoteRequestDis- patcher.java:201)[...]
spengestureservice injectInputEvent	162	{0, 91, array of android.view.InputEvent objects with a null item, false, NULL}	!@*** FATAL EXCEPTION IN SYSTEM PROCESS: android.ui java.lang.NullPointerException: Attempt to invoke virtual method 'long an- droid.view.InputEvent.getTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDis- patcher.dispatchInputEventInjection (SmartClipRemoteRequestDis- patcher.java:201)[...]

Continued on next page

Table 5.2: Failures Detected by Chizpurfle – *continued from previous page*

	ID	INPUT	FAILURE
spengestureservice injectInputEvent	186	{-188, 91, array of android.view.InputEvent objects with a null item, true, NULL}	!@*** FATAL EXCEPTION IN SYSTEM PROCESS: android.ui java.lang.NullPointerException: Attempt to invoke virtual method 'long android.view.InputEvent.getTime()' on a null object reference at com.samsung.android.content.smartclip. SmartClipRemoteRequestDis- patcher.dispatchInputEventInjection (SmartClipRemoteRequestDis- patcher.java:201)[...]
voip callInVoIP	54	{???9??\u001a??b\u0004A\ "1???HanI???\u0017??014? \u001a\u0006?Fu??UN [...] }	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 23452 an- droid.database.sqlite.SQLiteException: near "\", \"": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='u000e?? [...]
voip callInVoIP	55	{??_??\u0010_\u0001\bK) ?}??t'??R?G}T<T\u0001?\u 001b?????N?d?V??Z\u [...] }	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 24643 an- droid.database.sqlite.SQLiteException: near "\"???\": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='001?\u0 [...]
voip callInVoIP	72	{??y\u0014?~?\u0011??E\ u0007\u000b?'%?\u0016 yD\u0018??9t?i\u000 [...] }	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 25500 an- droid.database.sqlite.SQLiteException: unrecognized token: \"'??9??\u001a?????b?VN6g?, ~" (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='??9 [...]
voip callInVoIP	86	{??o??\bF?%\u0003?#, ?? ?t\u001a??9?~??Z\$??J\u0 016?\u0011\u0018?\u [...] }	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 32445 an- droid.database.sqlite.SQLiteException: near "\"???\": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number='?\u0011 [...]

Continued on next page

Table 5.2: Failures Detected by Chizpurfle – *continued from previous page*

	ID	INPUT	FAILURE
voip callInVoIP	105	{7??L6?I?{<81>?P! : ?\u00k05?\/?G~\u0003?#\u0000k??+c\u0016?\u001eA2 [...]}	FATAL EXCEPTION: mainProcess: com.samsung.android.incallui, PID: 5745 an- droid.database.sqlite.SQLiteException: near \"@?d???\": syntax error (code 1): , while compiling: SELECT reject_number FROM reject_num WHERE reject_number=?0Q?@b}W\u000e [...]

The first bug was found in the service `spengestureservice`, hosted by the System Server process. The bug was triggered by the method `injectInputEvent`. To understand the role of this method, we analyzed the AOSP, and found a similar method (with the same name and minor differences in the method signature) provided by the `InputManager` class of AOSP, which handles input devices such as keyboards. This method “injects an input event into the event system on behalf of an application”. It is likely that the method with the same name in the `spengestureservice` performs the same operation for input events from the *S Pen* in Samsung devices.

One of the input parameters for this method is an array of `android.view.InputEvent` objects, which is an abstract class for representing input events from hardware components. During the fuzz testing campaign, Chizpurfle detected a `FATAL EXCEPTION` when this array is non-null and non-empty, and at least one of its elements is null (instead, the service does not fail if the array is simply null or empty). This input causes the service to throw a `NullPointerException` that is not caught, causing a crash. We found that this bug is fully reproducible. The bug can have two different effects on the Android OS, depending on which process will consume the injected events from the Input Manager. If the events are consumed by the process `com.android.systemui`, the uncaught exception triggers the restart of the process, and a black screen of the user interface for a few seconds. If the events are consumed by `android.ui`, which is a thread of the `system_server` process, the fuzzed inputs has a higher impact: it crashes the `system_server` and causes a restart of the whole Android device. Several

method calls with exactly the same parameters values can be arbitrarily managed in both ways.

The second bug was triggered up when fuzzing the method `callInVoIP` of the Samsung's voip service. The method likely is used to place a call with *Samsung WE VoIP* app [116], a stock application that provides voice-over-IP for corporate users. The method takes as input parameter a string that represents a SIP address URI (e.g., `sip:1-999-123-4567@voip-provider.example.net`). Chizpurfle found that input strings that include specific SQL control expressions (similarly to single quotes in SQL injection) trigger an uncaught `SQLiteException` by the `com.samsung.android.incallui` process. This process is a customized version of the `com.android.incallui` process of the AOSP, which handles the UI that appears during a call, providing several on-screen functions. The uncaught exception crashes the `com.samsung.android.incallui` process, cutting off any ongoing call.

5.3.2 Comparison with Black-Box Fuzzing

We compared Chizpurfle with the black-box approach, to provide a baseline for evaluating our gray-box approach. We first analyze the performance overhead of Chizpurfle, *i.e.*, the relative slow-down of fuzz testing when applying the gray-box approach. The overhead includes the time for generating inputs and profiling the coverage of the tests. During the whole test campaign on the Samsung Galaxy S6 Edge, Chizpurfle measured the overall time spent for executing the test. An individual test takes on average 6.65 seconds, while testing a whole method takes on average 527.60 seconds.

To get the test duration that would be obtained with black-box fuzzing, we performed a second round of tests by disabling both the Chizpurfle's Seed Manager and Instrumentation Module (the two distinctive elements of gray-box testing). This usage mode of Chizpurfle (denoted as Chizpurfle^{BB}) is equivalent to perform black-box fuzzing, without neither collecting coverage nor using coverage for selecting the test inputs. In Chizpurfle^{BB}, the inputs are instead generated randomly. For each target method, we used Chizpurfle^{BB} by applying the same number of inputs that were also generated by the gray-box Chizpurfle for that method.

By comparing the time to run Chizpurfle^{BB} with the time to run the gray-box Chizpurfle, we obtain a performance slow-down per service of 11.97x on average. To put this number into context, we must consider that

the performance slow-down is inline with other tools for dynamic binary instrumentation. For example the Valgrind framework (which also uses dynamic binary rewriting for complex analyses, such as finding memory leaks and race conditions), when applied on the SPEC CPU 2006 benchmark [117], causes an average slow-down of 4.3x when the program is simply executed on the Valgrind virtual machine; and an average slow-down of 22.1x when performing memory leak analysis. Such overhead when running tests is rewarded by a higher bug-finding power, and it is in many cases accepted by developers as shown by the widespread adoption of Valgrind in automated regression test suites in open-source projects [24]. In our context, the slow-down still allows the Android system to execute without any noticeable side effect, thus preserving the intended behavior of the test cases. Figure 5.4 shows the performance overhead for the two services previously discussed (voip and spengestureservice), and for other 10 randomly-chosen custom vendor services, which cover the 10% of all the custom methods.

We then evaluate the gain, in terms of test coverage (the higher, the better), obtained by applying gray-box fuzzing instead of black-box fuzzing, given the same time budget T available for both forms of fuzz testing. To measure the test coverage of black-box fuzzing on the vendor customization, the only possible approach is to apply the Instrumentation Module of Chizpurfle (but without using the Seed Manager, in order to fuzz inputs in a random way). We denote this mode as Chizpurfle^{BB+COV}.

However, we need to take into account that code instrumentation slows down the execution of the black-box tests, and thus simply applying Chizpurfle^{BB+COV} for the same amount of wall-clock time of the gray-box Chizpurfle would unfairly penalize the black-box approach. Therefore, to obtain a fair estimate of the test coverage for black-box fuzzing, we compensate for the slow-down due to instrumentation by granting it a higher time budget than gray-box fuzzing. The time budget is obtained by multiplying the time budget of gray-box fuzzing for the slow-down due to instrumentation (while 11.97x is the average slow-down according to the experiments discussed above, here we applied to each method its slow-down factor).

On average, Chizpurfle covers 2.3x more code than the black box approach. The gain in terms of test coverage is shown in Figure 5.5 (which focuses on the same services analyzed in Figure 5.4).

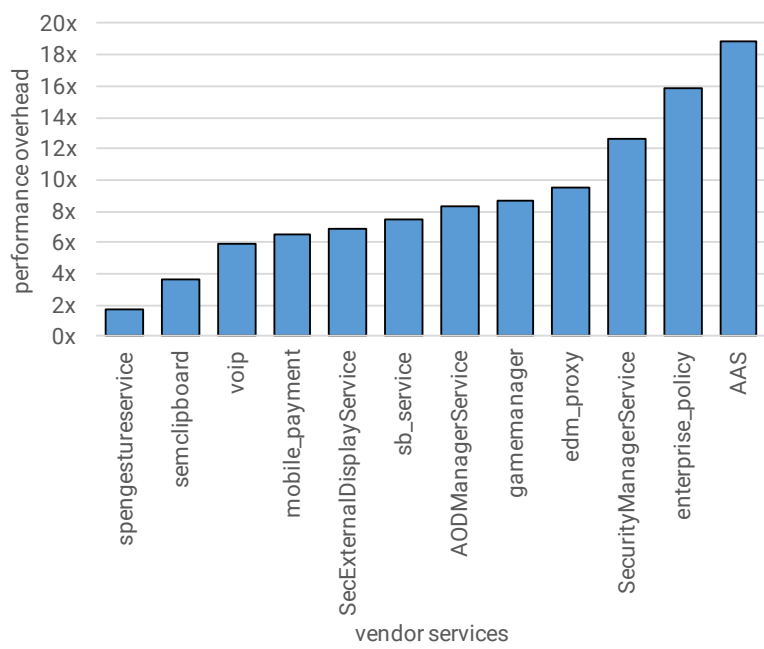


FIGURE 5.4: Performance Overhead of Chizpurfle

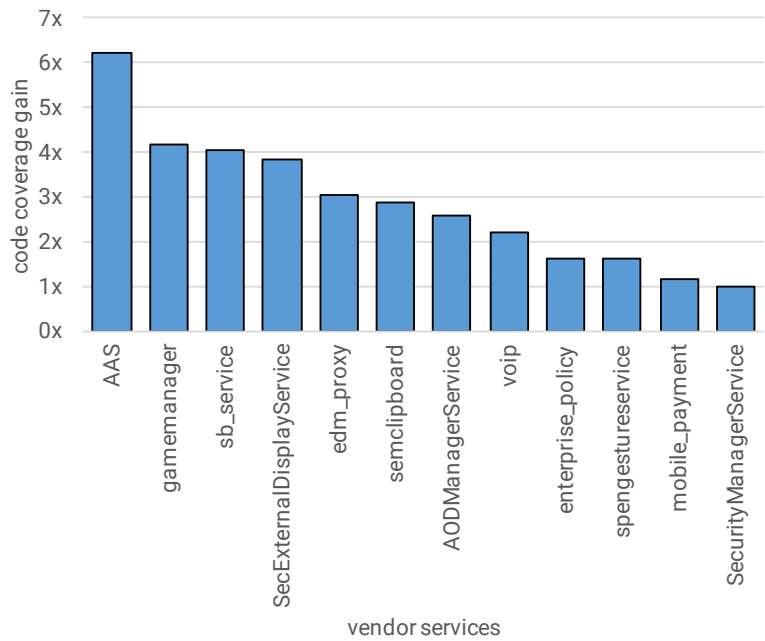


FIGURE 5.5: Code Coverage Gain of Chizpurple

By looking at the code coverage gain per method (see Figure 5.6), we noticed that Chizpurfle was more effective on those methods that take complex data in inputs, such as *semclipboard*'s method *updateFilter* takes as input an object of type *android.sec.clipboard.IClipboardDataPasteEvent* for managing clipboard data. Instead, in the case of simpler methods, such as getters and setters, the gray-box approach has a minor impact on test coverage.

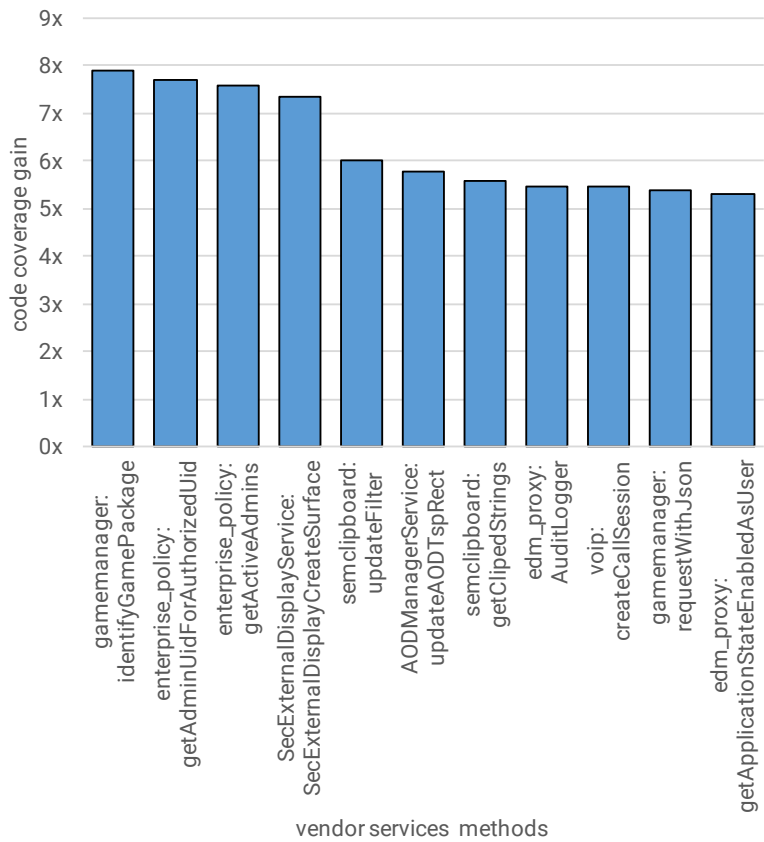


FIGURE 5.6: Code Coverage Gain of Chizpurfle per Method

Chapter 6

Conclusion And Future Directions

Projects we have completed demonstrate what we know — future projects decide what we will learn.

— Dr. Mohsin Tiwana

This thesis presented novel methods and experimental procedures to assess the dependability of mobile OS, specifically Android OS, fundamental for further improvements.

6.1 Fault Injection Testing

I presented the **SIR methodology**, which is based on the analysis of software interfaces and resource of the components. I applied the methodology on 14 components from 6 different subsystems of Android (*i.e.*, phone, camera, sensors, activity, package, and storage subsystems). This results in a Android Fault Model for these subsystems that counts more than 870 potential faults for the Android OS.

The extracted fault model is the base for the fault load used by **AndroFit**, the Android Fault Injection Tool, which design and implementation is presented in this thesis. AndroFit faces several technical problems to inject component failures in the Android OS, and to orchestrate a series of consecutive fault injection experiments.

AndroFit proved its effectiveness with an **experimental campaign**, executing fault injection experiments on three different Android smartphones

running Android 6 (Marshmallow): Huawei P8, Samsung Galaxy S6 Edge, and HTC One M9. It automatically executed more than 700 experiment on the three devices, for a total of 2196 experiments. Results show the differences on the dependability approach among vendors. I also presented some reliability improvements that can be applied by developers, and discussed some lesson learned that can be useful to future mobile OS fault injection practitioners.

The Android Fault Model and AndroFit could and should be extended to all the Android components, that requires a greater effort of reverse engineering and source code analysis. Fortunately, AndroFit is designed with maintenance in mind so that porting to next Android versions or new components should be done with little effort. Furthermore, the SIR methodology is valid for other mobile OS as well, such as iOS, and fault injection could be enabled also for this technologies. AndroFit can be part of the first **fault injection testing tools family** for mobile platforms.

6.2 Software Aging Analysis

We analyzed **software aging** issues in the Android OS, by performing a large experimental study across several devices, versions, and test configurations. From the stress tests, we obtained a large amount of data that provided us useful insights. Software aging effects in the vast majority of the tests, which thus confirm that the Android OS is indeed affected by software aging effects. Software aging impacted on the responsiveness of the device, as confirmed by the increasing, statistically-significant trends of the launch time of Android activities. Moreover, increases of the memory consumption of key processes of the Android OS.

The analysis considered devices from four leading Android vendors (Samsung, Huawei, LG, and HTC), and software aging occurs consistently across these four vendors. Thus software aging is **not limited to specific Android devices**. Moreover, the software aging effects are exacerbated by the specific Android vendors, as they apply customizations to the basic Android OS. Furthermore, the workload is another factor that significantly contributes to the extent of software aging (*e.g.*, by stressing the parts of the Android OS that are affected by aging-related bugs), as in our tests the

Chinese applications revealed higher software aging trends of the Android OS.

Similarly, the analysis of the presence and the variability of software aging across different versions of the Android OS considered the three most recent Android releases at the time of writing, *i.e.*, Android 5 (Lollipop), Android 6 (Marshmallow), and Android 7 (Nougat). All the considered Android versions are affected by software aging, thus pointing out that this problem is **not limited to specific Android versions**, but that the problem permeates the Android OS. Moreover, tests did not show an improvement of the Android OS over time, as the most recent Android release shows aging trends that are comparable to the previous one (*i.e.*, there are no statistically-significant differences). This finding remarks the need for more extensive tests to fix aging-related bugs, and for software rejuvenation solutions to mitigate the effects of the (unavoidable) aging-related bugs that get shipped with the products.

A final analysis of metrics inside the Android OS pointed out the possible causes of the software aging problems. The software aging trends are accompanied by a statistically-significant increase of the **memory consumption of key Android processes**. In particular, the memory consumption of the *System Server* (an Android OS process that runs many of the basic services of this system, including the *Activity Manager* for starting user applications) is significantly correlated with the performance degradation trends. The detailed analysis of these processes, by looking at garbage collection and task-level metrics, pointed out that the *System Server* spends more and more time on garbage collection during the experiments, which is a strong symptom that the memory utilization becomes more fragmented and bloated (*i.e.*, burdened by unnecessary objects) over time. Moreover, the task-level analysis identified the subsystems of the Android framework that exhibit increasing trends in terms of CPU utilization and virtual memory, which point out that these components are the ones most exercised in our experiments and are the possible source of the software aging effects.

Overall, these experiments point out the need for better testing and software rejuvenation to counteract software aging. Monitoring the memory utilization and garbage collection as in our tests (*e.g.*, by sampling the PSS and collecting logs from the ART) provides useful information to detect the onset of software aging (*i.e.*, a measurement-based approach). In turn, when

the software aging effects becomes too large (*e.g.*, compared to aging-free conditions after the boot of the device), the Android OS should trigger software rejuvenation at a convenient time (*e.g.*, restart selected services of the Android framework while the device is not used). The components identified in our analysis (such as the Activity Manager) are potential candidates for applying **software rejuvenation**.

6.3 Fuzz Testing

Chizpurfle filled the gap in the mobile fuzzers: it is a novel gray-box fuzzer designed to test custom system services from Android vendors. This tool exploits dynamic binary instrumentation to measure test coverage and to drive the selection of fuzz inputs. The experimental results on a commercial Android device from Samsung showed that the gray-box approach can discover relevant bugs, that it has a reasonable overhead, and that it can increase the test coverage compared to the black-box approach.

The gray-box fuzzing represents a promising approach for testing proprietary Android services in more depth. The Chizpurfle tool represents a **valuable opportunity for research** on fuzzing in mobile devices, by allowing to experiment with different heuristics for evolutionary fuzzing (*e.g.*, for determining when to stop fuzzing, for prioritizing seeds, and for selecting fuzz operators), as happened for similar fuzzing tools that were applied in different context than mobile devices. Another possible extension of Chizpurfle is to include support for system services implemented in C; since there is not reflection API, other reverse engineering techniques should be used in order to extract the method signatures.

6.4 Further Discussion

On August 21, 2017, Google released to the public **Android 8 (Oreo)** [118]. This version is a major release that introduces several changes, particularly for the vendor-related code. It better formalizes the role of the HAL interfaces, by introducing the HAL Interface Description Language (HIDL) similar to AIDL (Section A.2). The HAL will implements communication in a binderized mode, *i.e.*, by exploiting the Binder IPC mechanisms. As this communication dramatically increases binder traffic, several improvements

are designed to keep binder IPC fast. First of all, two new binder contexts backed by two new device file are added: the `/dev/hwbinder` to enable communication between framework/vendor processes and between vendor processes with HIDL, and the `/dev/vndbinder` to enable communication between vendor processes with AIDL. They left the `/dev/binder` only for communication between framework processes with AIDL. Another improvement for generic binder IPC is the scatter-gather optimization, which eliminates the need of Parcel objects. This re-architect of Android changes significantly the way vendor customizations are implemented, and poses new challenges and research opportunity on the dependability study of Android vendor customizations.

Mobile devices can be considered enablers for a larger and even more pervasive computing revolution, the **Internet of Things**. IoT allow object to be sensed or controlled remotely, creating a deep integration of the physical world into computer systems. Sensors and actuators exploit IoT to create smart homes, intelligent transportation, smart cities, smart manufacturing, and more. These technologies exacerbate the problem of undependable embedded devices, whereas smartphones are only tip of the iceberg. Google already entered the field with Android Things [119], a reduced Android release to build connected devices. This dissertation focuses mainly on mobile devices, but it can be a notable start point for the analysis of software dependability of the whole IoT.

If we live in a software-based environment, better it is reliable.

— Antonio Ken Iannillo

Appendix A

Android Insights

You know nothing, Jon Snow.

— Ygritte

This appendix presents Android and its internal, as a reference for this thesis. It is based on books study [78,85], websites visits [9,120], source code inspections [86], and reverse engineering on actual devices. Concepts in this appendix are valid from Android 5 (Lollipop) and updated to Android 7 (Nougat).

A.1 Android Architecture

An Android System consists of several layers, as shown in Figure [A.1](#).

Hardware

The Android device hardware on which the Android Software Stack is installed. It includes all the physical resources, such as CPU, RAM and battery, embedded in a single device with a great variety of sensors and other devices, *e.g.*, touchscreen, camera, Wi-Fi antenna and accelerometer.

Linux Kernel

The whole Android platform is based on the Linux kernel. Similar to other Linux distributions, Android applies its own patch on the *vanilla* kernel available from the Linux Kernel Archives. However, Android cannot be

The official website [120] defines Android as an OS, but if we visit the official website of the AOSP [9], Android refers to both “an open source software stack for a wide range of mobile devices and a corresponding open source project led by Google”. In this thesis, we use the following definitions:

- Android OS: the core OS built upon the Linux Kernel through which all devices resources are accessed;
- Android Framework: the collection of libraries and classes with the common goal of providing a base on which to build applications (apps) that extend the Android OS;
- Android Software Platform, or Android Platform, or Android Software Stack, or Android Stack: the open source software stack that provides all the necessary software to run a wide array of devices (it includes the Android OS and Android Framework);
- Android System: the combination of an Android device hardware and the Android Software Platform that runs on it;
- Android Ecosystem: the system of interdependence between users, developers, and equipment makers of the Android System.

Unless specified, the use of the term **Android** alone refers to the Android System.

considered a proper Linux distribution. Indeed, whereas in any Linux distribution a practitioner can substitute its kernel with another Linux kernel with little to no impact on the rest of the distribution’s components, Android’s user-space components run only an *Androidized* kernel or not at all. These unique features stem from specific kernel modules, namely Androidisms, such as the binder driver, wakelocks, low-memory killer, anonymous shared memory, alarm, logger, paranoid networking, and RAM console [78].

Hardware Abstraction Layer

The hardware abstraction layer (HAL) can be considered as a library loader, which loads vendor-specific hardware libraries that access and use the hardware components. An HAL module provides standard interfaces to

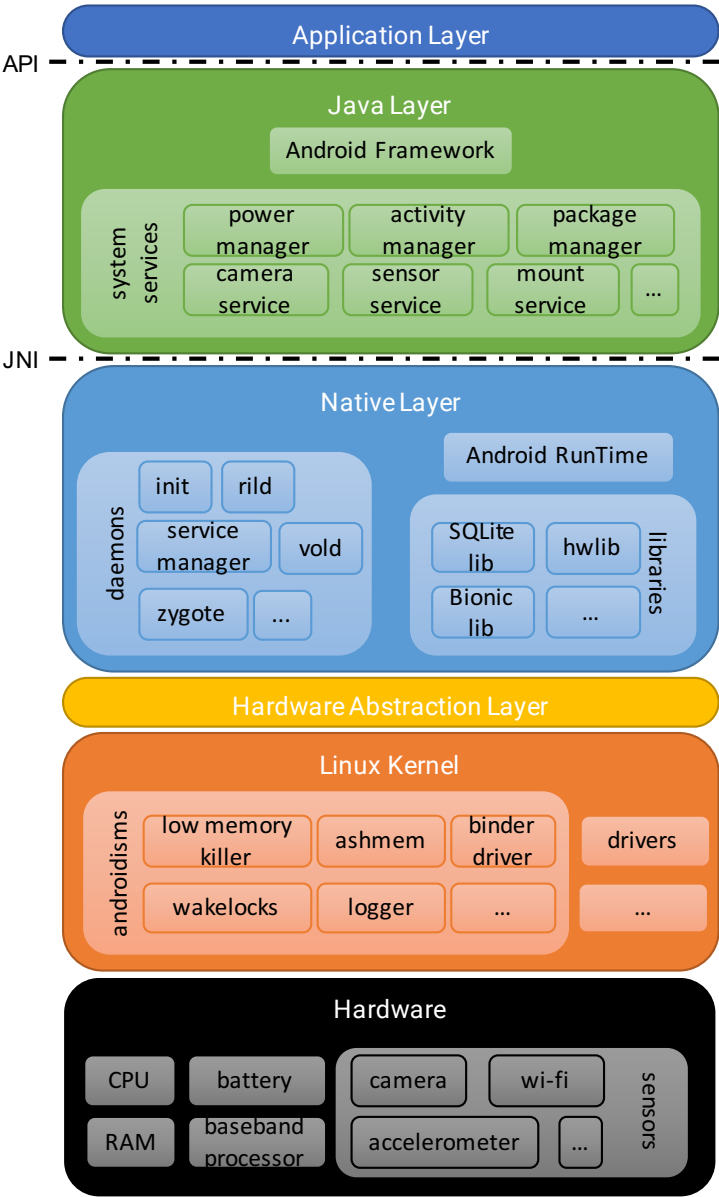


FIGURE A.1: Android System Architecture

the higher layers, independent from the specific hardware and drivers implementation, and it is loaded by the Android System at the appropriate time.

Native Layer

The native layer is implemented in C and C++ and consists of the Android Runtime, Native Libraries, and Native Daemons.

The Android Runtime (ART), introduced in Android 5 (KitKat), is the environment that allows Java-based and Android-specific code to be executed and to interface with the rest of the system. Every app runs in its own process and with its own instance of ART. ART executes bytecode optimized for minimal memory footprint, namely DEX code.

Native Libraries are written in C and C++ and their functionalities are available within the Android Platform. They include, among others, the Android implementation of the C library, namely Bionic.

The Native Daemons are key processes that continue to run throughout the lifetime of the system and provide essential services. The most important one is the `init` process. This process is the first, and only, user-space process started by the kernel and, then, it is responsible for spawning all other processes and services in the system.

Java Layer

The Java Layer is implemented in Java and consists of System Services and Android Framework.

The System Services are modular components that cooperate to manage all the features of the entire Android Platform. They essentially provide software interfaces, through the binder, to make an object-oriented OS built on top of Linux kernel.

The Android Framework is a collection of Java libraries that exposes all the Android OS features through Application Programming Interfaces (API). These APIs are the main instruments a developer needs to create Android apps. The Binder Inter-Process Communication (IPC) mechanisms, [Section A.2](#), allows the Android Framework to cross the process boundaries and to interact with the System Services.

The Java Layer can communicate with the Native Layer (mostly Native Libraries) also through the Java Native Interfaces (JNI).

Application Layer

The Application Layer includes applications or apps. The apps extend the functionalities of the device and they are either developed by the manufacturers for the specific device (stock apps) or downloaded from a market such as Google Play (third-party apps).

A.2 Binder IPC

Android's Binder Inter-Process Communication (IPC) mechanism enables remote method invocations from one Android process to another. It is of paramount importance since Android is designed on a strong process isolation principle, between both apps and different components of the Android platform itself. The Binder architecture consists of three parts.

- the **Binder Kernel Module**: it's a special kernel module that expose a device file, namely `/dev/binder`. It is also referred to as the Binder driver. It implements the remote procedure call (RPC) model, *i.e.*, the sending process submits an operation to the kernel that is executed in the receiving process. A message in the Binder Kernel Module is referred to as a transaction, which identifies the sender and the receiver, it determines the target operation, and it contains the complete data to exchange.
- the **Binder User-Space API**: it is a user-space object-oriented library (implemented both in Java and C/C++) that provides an API simpler to use than the Binder Kernel Module. It actually wraps the `ioctl` invocations defining Binder Proxies, used by the client to invoke a remote method, Binder Objects, implemented by the server to receive transactions from remote clients, and Parcel, a container for reading and writing data exchanged in a Binder transaction.
- the **Binder Interfaces**: a high-level interface-based programming model, that simplifies the Binder IPC hiding all the internal mechanisms. Once defined an interface with the Android Interface Definition

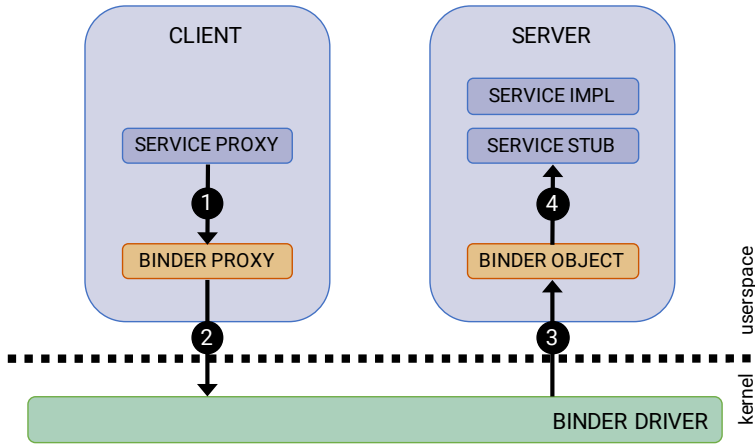


FIGURE A.2: Binder IPC Iteration Between Two Android Processes

Language (AIDL), stub and proxy can be automatically created by the `aidl` command-line tool. The proxy marshals the method call in to a Parcel and calls the underlying Binder Proxy. The stub is called by the Binder Object and decodes the Parcel into the appropriate method and arguments to call.

Figure A.2 depicts the flow of data to implement a remote method call between an Android process that acts as a client (*i.e.*, the caller) and one that acts as a server (*i.e.*, the callee), through the Binder IPC mechanism. Let's suppose that the client already obtained the Binder ID or handle of the server (this aspect is covered in Section A.3). **1** It first marshals the transaction objects into Parcels and **2**, through the Binder User-Space API, calls an `ioctl` syscall on the `/dev/binder` device file. The data is transferred to the kernel, and the Binder Driver looks up the required service to obtain its address space. The data is then copied to the server's address spaces and **3** the Binder driver wakes up a server worker thread to handle the request. **4** The server unmarshals the Parcels, check for client's permissions, and performs the requested service. Once the server computes a response, it is marshaled and sent back to the Binder driver, which dispatch it to the client in turn.

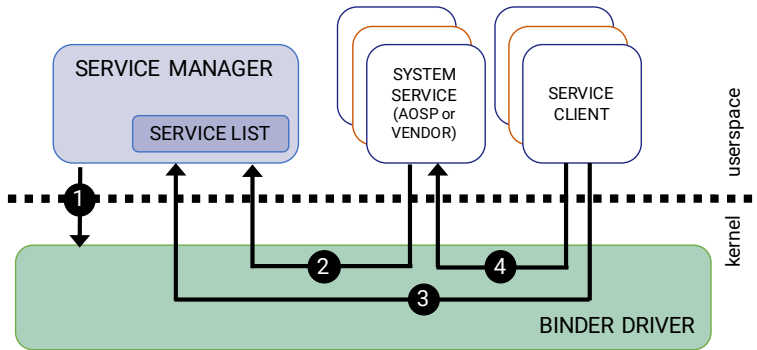


FIGURE A.3: Android Services and Service Manager

A.3 Service Manager

The Android OS provides a service-oriented architecture to manage its several services, as shown in Figure A.3. A process in the system should have a handle to the system service it wants to invoke through Binder. The Service Manager should be considered as a registry of all services available, and it is started by the `init` daemon before any other service to properly provide this functionality. Indeed, at boot time, ① the Service Manager registers itself as the **context manager**, by sending a special message to the Binder Driver. The Service Manager has a known handle, the Binder ID 0, so that any process in the system is able to communicate through Binder to it. Then, ② a service provider publishes its services by sending a message through the Binder driver to the Service Manager. When a client application wants to contact a service, ③ it first queries the Service Manager with the service name, and then ④ invokes the service directly through Binder with the handle the Service Manager provided.

The Service Manager is also invoked by some command-line utilities, such as `dumpsys`. This utility dumps the status of a single or all system services, obtaining the list of all services and the single handle querying the Service Manager. Once it get the handle, `dumpsys` invokes the service's dump function to dump its status and displays it on the terminal.

Appendix B

Android Fault Model

If you want something new, you have to stop doing something old.

— Peter F. Drucker

This appendix shows the Android Fault Model, presented and used as a reference in subsection [3.2.2](#).

- phone subsystem:
 - Table B.1: [RILD Fault Model](#)
 - Table B.2: [Baseband Driver and Processor Fault Model](#)
- camera subsystem:
 - Table B.3: [Camera Service Fault Model](#)
 - Table B.4: [Camera HAL Fault Model](#)
 - Table B.5: [Camera Driver and Hardware Fault Model](#)
- sensors subsystem:
 - Table B.6: [Sensor Service and HAL Fault Model](#)
 - Table B.7: [Sensors Drivers and Devices Fault Model](#)
- activity subsystem:
 - Table B.8: [Activity Manager Service Fault Model](#)
- package subsystem:

- Table B.9: [Package Manager Service Fault Model](#)
- storage subsystem:
 - Table B.10: [SQLite Library Fault Model](#)
 - Table B.11: [Bionic Library Fault Model](#)
 - Table B.12: [Mount Service Fault Model](#)
 - Table B.13: [Volume Daemon Fault Model](#)
 - Table B.14: [Storage Drivers and Hardware Fault Model](#)

TABLE B.1: RILD Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The RILD crashes	resource management	PROCESS & THREADS	A software fault causes RILD to crash	PERMANENT
The RILD hangs	resource management	PROCESS & THREADS	A software fault causes RILD to stall	PERMANENT TRANSIENT
The RILD cannot allocate memory (due to software aging)	resource management	MEMORY	RILD leaks memory due to a software aging issue	PERMANENT TRANSIENT
The RILD cannot allocate files (due to software aging)	resource management	FILE	RILD leaks file descriptors due to a software aging issue	PERMANENT TRANSIENT
The RILD drops or cannot open sockets/pipes	resource management	PIPE	The RILD uses pipes to enable communication between threads	PERMANENT TRANSIENT
The RILD drops or cannot open the rild socket	resource management	SOCKET	The RILD uses a socket to communicate with the framework	PERMANENT TRANSIENT
The RILD reads from the RILD socket with a high latency, and event delivery is postponed	timeliness	RECEIVE PHONE COMMAND ON RILD SOCKET	A read operation on the socket is delayed	PERMANENT INTERMITTENT TRANSIENT
The RILD is unable to read from the RILD socket	timeliness	RECEIVE PHONE COMMAND ON RILD SOCKET	A read operation on the socket receives no reply	PERMANENT INTERMITTENT TRANSIENT
The RILD tries to read from the RILD socket but it fails	availability	RECEIVE PHONE COMMAND ON RILD SOCKET	A read operation on the socket fails and returns an error code	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.1: RILD Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The RILD reads a corrupted request from the RILD socket	output value	RECEIVE PHONE COMMAND ON RILD SOCKET	A read operation gets corrupted data	PERMANENT INTERMITTENT TRANSIENT
The RILD is unable to write into the RILD socket	timeliness	SEND PHONE EVENTS ON RILD SOCKET	A write operation on the socket is never performed	PERMANENT INTERMITTENT TRANSIENT
The RILD sends operation through the RILD socket with a high latency, and response delivery is postponed	timeliness	SEND PHONE EVENTS ON RILD SOCKET	A write operation on the socket is delayed	PERMANENT INTERMITTENT TRANSIENT
The RILD tries to write on the RILD socket but it fails	availability	SEND PHONE EVENTS ON RILD SOCKET	A write operation on the socket fails and returns an error code	PERMANENT INTERMITTENT TRANSIENT
The RILD writes a corrupted response on the RILD socket	output value	SEND PHONE EVENTS ON RILD SOCKET	A write operation sends corrupted data	PERMANENT INTERMITTENT TRANSIENT
The RILD is unable to write AT command to the baseband processor	timeliness	WRITE AT COMMAND TO MODEM	AT commands to the baseband processor are lost	PERMANENT INTERMITTENT TRANSIENT
The RILD writes to the baseband processor with a high latency	timeliness	WRITE AT COMMAND TO MODEM	AT commands to the baseband processor are delayed	PERMANENT INTERMITTENT TRANSIENT
The RILD tries to write AT data to the baseband processor but it receives an error	availability	WRITE AT COMMAND FROM MODEM	A write operation on the AT channel fails with an error code	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.1: RILD Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The RILD writes corrupted AT data to baseband processor	output value	WRITE AT COMMAND TO MODEM	AT data is corrupted	PERMANENT INTERMITTENT TRANSIENT
The RILD is unable to read AT command from the baseband processor	timeliness	READ AT COMMAND FROM MODEM	AT commands to the baseband processor are lost	PERMANENT INTERMITTENT TRANSIENT
The RILD reads from the baseband processor with a high latency	timeliness	READ AT COMMAND FROM MODEM	AT commands to the baseband processor are delayed	PERMANENT INTERMITTENT TRANSIENT
The RILD tries to read AT data from baseband processor but it receives an error	availability	READ AT COMMAND FROM MODEM	A read operation on the AT channel fails with an error code	PERMANENT INTERMITTENT TRANSIENT
The RILD reads corrupted AT data from baseband processor	output value	READ AT COMMAND FROM MODEM	AT data is corrupted	PERMANENT INTERMITTENT TRANSIENT

TABLE B.2: Baseband Driver and Processor Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The phone driver is unable to allocate memory	resource management	MEMORY	The device driver is unable to allocate resources (memory, I/O regions) due to a software bug in the kernel, an overload, or an error raised by the hardware.	PERMANENT TRANSIENT

Continued on next page

Table B.2: Baseband Driver and Processor Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The phone driver cannot be accessed	resource management	DEVICE FILE	The virtual device file (for example, /dev/appvcom*, /dev/sdm*) cannot be accessed due to the lack of read/write permissions	PERMANENT TRANSIENT
Protocol error in the device driver, causing the abort of an I/O operation	availability	SEND PHONE SIGNAL / RECEIVE PHONE SIGNAL	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The hardware device always returns an error.	PERMANENT INTERMITTENT TRANSIENT
Protocol error in the device driver, causing the corruption of an I/O operation	output value	SEND PHONE SIGNAL / RECEIVE PHONE SIGNAL	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations is not performed.	PERMANENT INTERMITTENT TRANSIENT
Protocol error in the device driver, causing the stall of an I/O operation	timeliness	SEND PHONE SIGNAL / RECEIVE PHONE SIGNAL	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations takes longer time than expected.	PERMANENT INTERMITTENT TRANSIENT

TABLE B.3: Camera Service Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
Media server hang	resource management	PROCESS & THREADS	The camera is available before and during the operation. Randomly in time, the media server hangs.	PERMANENT TRANSIENT
Media server crash	resource management	PROCESS & THREADS	The camera is available before and during the operation. Randomly in time, the media server crashes.	PERMANENT
The media server cannot allocate memory (due to software aging)	resource management	MEMORY	The media server is affected by software aging, leaking memory. The media server is unable to allocate memory.	PERMANENT TRANSIENT
The media server cannot open files (due to software aging)	resource management	FILE	The media server is affected by software aging, leaking file descriptors. The media server is unable to open file descriptors.	PERMANENT TRANSIENT
The media server cannot open sockets or pipes (due to software aging)	resource management	SOCKET / PIPE	The media server is affected by an overload, that saturates the sockets and pipes. The media server is unable to open new pipe or socket for interprocess communication.	PERMANENT TRANSIENT
The media server cannot access to the Binder	resource management	BINDER OBJECT	The media server is affected by an overload, thus saturating the Binder communication buffers. The media server fails when attempting to send a message.	PERMANENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
Media server can't start the thread pool	resource management	PROCESS & THREADS	The mediaserver uses a thread pool to manage the threads for its services. This fault hampers the initialization of this thread pool.	PERMANENT
The ICamera processes the start preview request after a high delay	timeliness	ICAMERA STARTPREVIEW	A user requests to see a preview of their subject before clicking the shutter, but the request takes a long time.	PERMANENT INTERMITTENT TRANSIENT
The ICamera does not respond to the start preview request	timeliness	ICAMERA STARTPREVIEW	A user requests to see a preview of their subject before clicking the shutter, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera fails in sending a start preview request generating an error	availability	ICAMERA STARTPREVIEW	A user requests to see a preview of their subject before clicking the shutter, but the request fails with an error.	PERMANENT INTERMITTENT TRANSIENT
The ICamera processes the stop preview request after a high delay	timeliness	ICAMERA STOPPREVIEW	A user requests to stop the preview displaying on the screen, but the request takes a long time.	PERMANENT INTERMITTENT TRANSIENT
The ICamera does not respond to the stop preview request	timeliness	ICAMERA STOPPREVIEW	A user requests to stop the preview displaying on the screen, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera processes the start recording request after a high delay	timeliness	ICAMERA STARTRECORDING	A user requests to start recording a video with the camera, but the request is accepted after a long period of time	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICamera does not respond to the start recording request	timeliness	ICAMERA. STARTRECORD- ING	A user requests to start recording a video with the camera, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera fails in sending a start recording request generating an error	availability	ICAMERA. STARTRECORD- ING	A user requests to start recording a video with the camera, but the request fails with an error.	PERMANENT INTERMITTENT TRANSIENT
The ICamera processes the stop recording request after a high delay	timeliness	ICAMERA.STO- PRECORDING	A user requests to stop recording the video with the camera, but the request takes a long time.	PERMANENT INTERMITTENT TRANSIENT
The Icamera does not respond to the stop recording request	timeliness	ICAMERA.STO- PRECORDING	A user requests to stop recording the video with the camera, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera processes the take picture request after a high delay	timeliness	ICAMERA. TAKEPICTURE	A user requests to take a picture with the camera, but the request takes a long time.	PERMANENT INTERMITTENT TRANSIENT
The ICamera does not respond to the take picture request	timeliness	ICAMERA. TAKEPICTURE	A user requests to take a picture with the camera, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera fails in sending a take picture request generating an error	availability	ICAMERA. TAKEPICTURE	A user requests to take a picture with the camera, but the request fails with an error.	PERMANENT INTERMITTENT TRANSIENT
The ICamera corrupts the set parameters	output value	ICAMERA.SET- PARAMETERS	A user requests to set the parameters (exposure, color balance, focus, effects) of the camera, but the request is corrupted.	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICamera processes the set parameters request after a high delay	timeliness	ICAMERA. SET- PARAMETERS	A user requests to set the parameters (exposure, color balance, focus, effects) of the camera, but the request takes a long time.	PERMANENT INTERMITTENT TRANSIENT
The ICamera does not respond to the set parameters request	timeliness	ICAMERA. SET- PARAMETERS	A user requests to set the parameters (exposure, color balance, focus, effects) of the camera, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera fails in sending a set parameters request generating an error	availability	ICAMERA. SET- PARAMETERS	A user requests to set the parameters (exposure, color balance, focus, effects) of the camera, but the request fails with an error.	PERMANENT INTERMITTENT TRANSIENT
The ICamera returns a corrupted version of the parameters of the camera	output value	ICAMERA. GET- PARAMETERS	A user requests to get the parameters (exposure, color balance, focus, effects) of the camera, but the request is corrupted.	PERMANENT INTERMITTENT TRANSIENT
The ICamera processes the get parameters request after a high delay	timeliness	ICAMERA. GET- PARAMETERS	A user requests to get the parameters (exposure, color balance, focus, effects) of the camera, but the request takes a long time.	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICamera does not respond to the get parameters request	timeliness	ICAMERA. GET-PARAMETERS	A user requests to get the parameters (exposure, color balance, focus, effects) of the camera, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera fails in sending a get parameters request generating an error	availability	ICAMERA. GET-PARAMETERS	A user requests to get the parameters (exposure, color balance, focus, effects) of the camera, but the request fails with an error.	PERMANENT INTERMITTENT TRANSIENT
The ICamera processes the send command request after a high delay	timeliness	ICAMERA. SENDCOM-MAND	A user requests to send a generic command (smooth zoom, display orientation, sounds, face detection) to the camera, but the request takes a long time.	PERMANENT INTERMITTENT TRANSIENT
The ICamera does not respond to the send command request	timeliness	ICAMERA. SENDCOM-MAND	A user requests to send a generic command (smooth zoom, display orientation, sounds, face detection) to the camera, but the request never receives a response.	PERMANENT INTERMITTENT TRANSIENT
The ICamera fails in sending a send command request generating an error	availability	ICAMERA. SENDCOM-MAND	A user requests to send a generic command (smooth zoom, display orientation, sounds, face detection) to the camera, but the request fails with an error.	PERMANENT INTERMITTENT TRANSIENT
The ICameraClient returns a corrupted notification type and parameters	output value	ICAMERA-CLIENT. NOTIFYCALL-BACK	A user wait for a notification from the camera (shutter event), but the request is corrupted	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICameraClient notify an event after a high delay	timeliness	ICAMERA-CLIENT-NOTIFYCALL-BACK	A user wait for a notification from the camera (shutter event), but the request reply is never received	PERMANENT INTERMITTENT TRANSIENT
The ICameraClient does not notify an event	timeliness	ICAMERA-CLIENT-NOTIFYCALL-BACK	A user wait for a notification from the camera (shutter event), but the request reply takes a long time	PERMANENT INTERMITTENT TRANSIENT
The ICameraClient returns corrupted data	output value	ICAMERA-CLIENT-DATACALL-BACK	A user wait for a notification with data (metadata, raw image) from the camera (shutter event), but the request is corrupted	PERMANENT INTERMITTENT TRANSIENT
The ICameraClient notify a data event after a high delay	timeliness	ICAMERA-CLIENT-DATACALL-BACK	A user wait for a notification with data (metadata, raw image) from the camera (shutter event), but the request reply takes a long time	PERMANENT INTERMITTENT TRANSIENT
The ICameraClient does not notify a data event	timeliness	ICAMERA-CLIENT-DATACALL-BACK	A user wait for a notification with data (metadata, raw image, jpeg image) from the camera (shutter event), but the request reply is never received	PERMANENT INTERMITTENT TRANSIENT
The ICameraService returns a corrupted camera info	output value	ICAMERASERVICE-GETCAMERAINFO	A user wants to retrieve the information of the cameras, but the Camera Service provides the corrupted informations	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICameraService processes the camera info request with a great delay	timeliness	ICAMERASER- VICE. GETCAM- ERAINFO	A user wants to retrieve the information of the cameras, but the Camera Service replies with a great delay	PERMANENT INTERMITTENT TRANSIENT
The ICameraService does not respond to the camera info request	timeliness	ICAMERASER- VICE. GETCAM- ERAINFO	A user wants to retrieve the information of the cameras, but the Camera Service does not reply	PERMANENT INTERMITTENT TRANSIENT
The ICameraService fails in getting camera info and returns an error	availability	ICAMERASER- VICE. GETCAM- ERAINFO	A user wants to retrieve the information of the cameras, but the Camera Service returns an error code	PERMANENT INTERMITTENT TRANSIENT
The ICameraService returns a corrupted CameraDevice	output value	ICAMERASER- VICE. CONNECTDE- VICE	A user wants to retrieve the CameraDevice connected to the requested cameras, but the Camera Service provides a corrupted CameraDevice	PERMANENT INTERMITTENT TRANSIENT
The ICameraService process the connect device request with a great delay	timeliness	ICAMERASER- VICE. CONNECTDE- VICE	A user wants to retrieve the CameraDevice connected to the requested cameras, but the Camera Service replies with a great delay	PERMANENT INTERMITTENT TRANSIENT
The ICameraService does not respond to the connect device request	timeliness	ICAMERASER- VICE. CONNECTDE- VICE	A user wants to retrieve the CameraDevice connected to the requested cameras, but the Camera Service does not reply	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICameraService fails in connecting the device and returns an error	availability	ICAMERASERVICE. CONNECTEDDEVICE	A user wants to retrieve the CameraDevice connected to the requested cameras, but the CameraService returns an error code	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceUser returns a corrupted lastFrameNumber	output value	ICAMERADEVICEUSER. SUBMITREQUEST	A user wants to submit a request to the CameraDevice, the ICameraDeviceUser submits the request but it returns an incorrect lastFrameNumber	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceUser submits the request with a great delay	timeliness	ICAMERADEVICEUSER. SUBMITREQUEST	A user wants to submit a request to the CameraDevice, but the ICameraDeviceUser submit the request with a great delay	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceUser does not actually submit the request	timeliness	ICAMERADEVICEUSER. SUBMITREQUEST	A user wants to submit a request to the CameraDevice, but the ICameraDeviceUser submit does not submit the request and does not notify any error	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceUser fails in submitting a request and returns an error	availability	ICAMERADEVICEUSER. SUBMITREQUEST	A user wants to submit a request to the CameraDevice, but the ICameraDeviceUser fails in submission and return an error	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceUser processes the create stream request with a great delay	timeliness	ICAMERADEVICEUSER. CREATESTREAM	A user wants to create a stream (input/output stream) for the camera with the CameraDevice, but the ICameraDeviceUser creates the stream with a great delay	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICameraDeviceUser does not respond to the create stream request	timeliness	ICAMERADE- VICEUSER. CREATESTREAM	A user wants to create a stream (input/output stream) for the camera with the CameraDevice, but the ICameraDeviceUser creates does not create the stream and does not notify any error	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceUser fails in processing the create stream request and returns an error	availability	ICAMERADE- VICEUSER. CREATESTREAM	A user wants to create a stream (input/output stream) for the camera with the CameraDevice, but the ICameraDeviceUser fails in the stream creation and return an error	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceCallbacks provides a corrupted information while notifying an the capture started event	output value	ICAMERADE- VICECALL- BACKS. ONCAP- TURESTARTED	The CameraDevice should be notified the capture started event with related information (timestamp and extras), but the ICameraDeviceCallbacks send corrupted info with the binder transaction	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceCallbacks provides capture started notification with a great delay	timeliness	ICAMERADE- VICECALL- BACKS. ONCAP- TURESTARTED	The CameraDevice should be notified with a camera capture started event, but the ICameraDeviceCallbacks sends the binder transaction with a great delay	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICameraDeviceCallbacks does not notify the capture started event	timeliness	ICAMERADE- VICECALL- BACKS. ONCAP- TURESTARTED	The CameraDevice should be notified with a camera capture started event, but the ICameraDeviceCallbacks does not send the binder transaction	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceCallbacks provides a corrupted information while notifying an the result received event	output value	ICAMERADE- VICECALL- BACKS. ONRESULTRE- CEIVED	The CameraDevice should be notified the result received event with related information (metadata and extras), but the ICameraDeviceCallbacks send corrupted info with the binder transaction	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceCallbacks provides result received notification with a great delay	timeliness	ICAMERADE- VICECALL- BACKS. ONRESULTRE- CEIVED	The CameraDevice should be notified with a camera result received event, but the ICameraDeviceCallbacks sends the binder transaction with a great delay	PERMANENT INTERMITTENT TRANSIENT
The ICameraDeviceCallbacks does not notify the result received event	timeliness	ICAMERADE- VICECALL- BACKS. ONRESULTRE- CEIVED	The CameraDevice should be notified with a camera result received event, but the ICameraDeviceCallbacks does not send the binder transaction	PERMANENT INTERMITTENT TRANSIENT
The ICameraServiceListener provides a corrupted information while notifying the status of a camera	output value	ICAMERASER- VICELISTENER. ONSTA- TUSCHANGED	The CameraService should be notified the status changed event of a specific camera, but the ICameraServiceListener sends corrupted info with the binder transaction	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.3: Camera Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The ICameraServiceListener provides a status change event with a great delay	timeliness	ICAMERASER- VICELISTENER. ONSTA- TUSCHANGED	The CameraService should be notified with a status changed event, but the ICameraServiceListener sends the binder transaction with a great delay	PERMANENT INTERMITTENT TRANSIENT
The ICameraServiceListener does not notify the status changed event	timeliness	ICAMERASER- VICELISTENER. ONSTA- TUSCHANGED	The CameraService should be notified with a status changed event, but the ICameraServiceListener does not send the binder transaction	PERMANENT INTERMITTENT TRANSIENT

TABLE B.4: Camera HAL Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The camera HAL process the frame buffer enable/disable request with a great delay	timeliness	CAMERA_ DEVICE. CAMERA_ DEVICE_OPS_ T. STOREMETA- DATAIN- BUFFERS	Request the camera HAL to store meta data or real data in the video buffers for a recording session, but the request is sent with a great delay	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.4: Camera HAL Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The camera HAL does not process the frame buffer enable/disable request	timeliness	CAMERA_DEVICE. CAMERA_DEVICE_OPS_T. STOREMETA-DATAIN-BUFFERS	Request the camera HAL to store data in the video buffers for a recording session, but the request never receives a response	PERMANENT INTERMITTENT TRANSIENT
		CAMERA_DEVICE. CAMERA_DEVICE_OPS_T. STOREMETA-DATAIN-BUFFERS	Request the camera HAL to store meta data or real data in the video buffers for a recording session, but it returns an error	PERMANENT INTERMITTENT TRANSIENT
The camera HAL returns an error on the frame buffer enable/disable request	availability	CAMERA_DEVICE. CAMERA_DEVICE_OPS_T. PREVIEW_OPS_T. ENQUEUE_BUFFER	The camera fill the buffers of the preview stream with data, but this data are corrupted	PERMANENT INTERMITTENT TRANSIENT
		CAMERA_DEVICE. CAMERA_DEVICE_OPS_T. PREVIEW_OPS_T. ENQUEUE_BUFFER	The camera wants to insert in the preview stream the frames for the preview, but it does it with a great delay	PERMANENT INTERMITTENT TRANSIENT
The camera corrupts the buffers of the preview stream	output value	CAMERA_DEVICE. CAMERA_DEVICE_OPS_T. PREVIEW_OPS_T. ENQUEUE_BUFFER	The camera fill the buffers of the preview stream with data, but this data are corrupted	PERMANENT INTERMITTENT TRANSIENT
		CAMERA_DEVICE. CAMERA_DEVICE_OPS_T. PREVIEW_OPS_T. ENQUEUE_BUFFER	The camera wants to insert in the preview stream the frames for the preview, but it does it with a great delay	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.4: Camera HAL Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The camera does not signal the production of new frames in the preview stream	timeliness	CAMERA_DEVICE.	The camera wants to insert in the preview stream the frames but the consumer is not signaled for the new frames into the buffer.	PERMANENT INTERMITTENT TRANSIENT
		CAMERA_PREVIEW_OPS_T. ENQUEUE_BUFFER		
The camera HAL returns corrupted info on the created buffer on the allocate stream request	output value	CAMERA2_DEVICE.	Request the camera HAL to allocate a new input stream, which will use the buffers allocated for an existing output stream, but it returns a corrupted output	PERMANENT INTERMITTENT TRANSIENT
		CAMERA2_DEVICE_OPS_T. ALLOCATE_STREAM		
The camera HAL process the allocate stream request with a great delay	timeliness	CAMERA2_DEVICE.	Request the camera HAL to allocate a new input stream for use, which will use the buffers allocated for an existing output stream, but the request is sent with a great delay	PERMANENT INTERMITTENT TRANSIENT
		CAMERA2_DEVICE_OPS_T. ALLOCATE_STREAM		
The camera HAL does not process the allocate stream request	timeliness	CAMERA2_DEVICE.	Request the camera HAL to allocate a new input stream for use, which will use the buffers allocated for an existing output stream, but the request never receives a response	PERMANENT INTERMITTENT TRANSIENT
		CAMERA2_DEVICE_OPS_T. ALLOCATE_STREAM		

Continued on next page

Table B.4: Camera HAL Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The camera HAL returns an error on the allocate stream request	availability	CAMERA2_DEVICE.	Request the camera HAL to allocate a new input stream for use, which will use the buffers allocated for an existing output stream, but it returns an error	PERMANENT INTERMITTENT TRANSIENT
		CAMERA2_DEVICE_OPS_T.ALLOCATE_STREAM		
The camera corrupts the buffers of the input/output stream	output value	CAMERA2_DEVICE.	The camera fill the buffers of the input/output stream with data, but this data are corrupted	PERMANENT INTERMITTENT TRANSIENT
		CAMERA2_STREAM_OPS_T.ENQUEUE_BUFFER		
The camera signals the production of new frames in the input/output stream with a great delay	timeliness	CAMERA2_DEVICE.	The camera wants to insert in the input/output stream the frames for the input/output, but it does it with a great delay	PERMANENT INTERMITTENT TRANSIENT
		CAMERA2_STREAM_OPS_T.ENQUEUE_BUFFER		
The camera does not signal the production of new frames in the input/output stream	timeliness	CAMERA2_DEVICE.	The camera wants to insert in the input/output stream the frames for the input/output, but the consumer is not signaled for the new frames into the buffer.	PERMANENT INTERMITTENT TRANSIENT
		CAMERA2_STREAM_OPS_T.ENQUEUE_BUFFER		

Continued on next page

Table B.4: Camera HAL Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The camera HAL process the configure streams request with a great delay	timeliness	CAMERA3_ DEVICE.	Request the camera HAL to reset the HAL camera device processing pipeline and set up new input and output streams, but the request is sent with a great delay	PERMANENT INTERMITTENT TRANSIENT
		CAMERA3_ DEVICE_OPS_ CONFIGURE_ STREAMS		
The camera HAL does not process the configure streams request	timeliness	CAMERA3_ DEVICE.	Request the camera HAL to reset the HAL camera device processing pipeline and set up new input and output streams, but the request never receives a response	PERMANENT INTERMITTENT TRANSIENT
		CAMERA3_ DEVICE_OPS_ CONFIGURE_ STREAMS		
The camera HAL returns an error on the configure streams request	availability	CAMERA3_ DEVICE.	Request the camera HAL to reset the HAL camera device processing pipeline and set up new input and output streams, but it returns an error	PERMANENT INTERMITTENT TRANSIENT
		CAMERA3_ DEVICE_OPS_ CONFIGURE_ STREAMS		
The camera corrupts the buffers of the input/output stream	output value	CAMERA3_ DEVICE.	The camera fill the buffers of the input/output stream with data, but this data are corrupted	PERMANENT INTERMITTENT TRANSIENT
		CAMERA3_ DEVICE_OPS_ PROCESS_ CAPTURE_ REQUEST		

Continued on next page

Table B.4: Camera HAL Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The camera signals the production of new frames in the input/output stream with a great delay	timeliness	CAMERA3_DEVICE. CAMERA3_DEVICE_OPS_T. PROCESS_CAPTURE_REQUEST	The camera wants to insert in the input/output stream the frames for the input/output, but it does it with a great delay	PERMANENT INTERMITTENT TRANSIENT
		CAMERA3_DEVICE. CAMERA3_DEVICE_OPS_T. PROCESS_CAPTURE_REQUEST	The camera wants to insert in the input/output stream the frames for the input/output, but the consumer is not signaled for the new frames into the buffer.	PERMANENT INTERMITTENT TRANSIENT

TABLE B.5: Camera Driver and Hardware Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The Camera Driver is unable to allocate resources (memory, I/O regions)	resource management	DEVICE FILE	The virtual device file is not accessible, because of a bug in the kernel, in the driver, or in user-space device management utilities	PERMANENT TRANSIENT

Continued on next page

Table B.5: Camera Driver and Hardware Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The virtual device file (e.g., /dev/video0) cannot be accessed due to the lack of read/write permissions	resource management	MEMORY	The device driver is unable to allocate resources (memory, I/O regions) due to a software bug in the kernel, an overload, or an error raised by the hardware. This fault causes the availability of the hardware device.	PERMANENT TRANSIENT
Protocol error in the device driver, causing the abort of an I/O operation	availability	READ CAMERA DATA	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The hardware device status is reset or is hung	PERMANENT INTERMITTENT TRANSIENT
Protocol error in the device driver, causing the corruption of an I/O operation	output value	READ CAMERA DATA	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations are aborted	PERMANENT INTERMITTENT TRANSIENT
Protocol error in the device driver, causing the stall of an I/O operation	timeliness	READ CAMERA DATA	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations are stalled	PERMANENT INTERMITTENT TRANSIENT

TABLE B.6: Sensor Service and HAL Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
System Server hangs including all threads	resource management	PROCESS & THREADS	The System Server is affected by a critical bug that causes the termination of the System Server as a whole	PERMANENT TRANSIENT
System Server crashes, including all threads	resource management	PROCESS & THREADS	The System Server is affected by a critical bug that causes the stall of the System Server as a whole	PERMANENT
System server cannot allocate memory	resource management	MEMORY	The System Server is affected by software aging and leaks memory. It is unable to allocate more memory.	PERMANENT TRANSIENT
System server cannot open files	resource management	FILES	The System Server is affected by software aging and leaks file descriptors. It is unable to open files.	PERMANENT TRANSIENT
Sensor Service crashes	resource management	PROCESS & THREADS	Sensor Service thread crashes due to unhandled exception	PERMANENT
Sensor service hangs	resource management	PROCESS & THREADS	Sensor Service thread is not responsive due to software bugs, such as a deadlock	PERMANENT TRANSIENT
Sensor Service can't open or drops sockets	resource management	SOCKET	The service is not able to open a socket or abruptly closes the already opened sockets	PERMANENT TRANSIENT
Sensor Service drops the Binder Object	resource management	BINDER OBJECT	The service is not able to use the communication facilities offered by the binder	PERMANENT TRANSIENT

Continued on next page

Table B.6: Sensor Service and HAL Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The SensorService does not return the list of available sensors	availability	GET SENSORS LIST	The Sensor Service cannot provide the list of the available sensors.	PERMANENT INTERMITTENT TRANSIENT
The SensorService returns an incorrect list of available sensors	output value	GET SENSORS LIST	The Sensor Service provides a wrong list of the available sensors.	PERMANENT INTERMITTENT TRANSIENT
The SensorService fails to establish a sensor channel	availability	ESTABLISH SENSOR CHANNEL	The Sensor Service is unable to create an Event Channel.	PERMANENT INTERMITTENT TRANSIENT
The SensorService omits to forward sensor events	timeliness	SEND SENSOR EVENTS	The Sensor Service does not update sensor data.	PERMANENT INTERMITTENT TRANSIENT
The SensorService delays the forwarding of sensor events	timeliness	SEND SENSOR EVENTS	The Sensor Service does not timely update sensor data.	PERMANENT INTERMITTENT TRANSIENT
The SensorService corrupts sensor events	output value	SEND SENSOR EVENTS	The Sensor Service incorrectly updates sensor data.	PERMANENT INTERMITTENT TRANSIENT

TABLE B.7: Sensors Drivers and Devices Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The virtual device file (/dev) cannot be accessed due to the lack of read/write permissions	resource management	DEVICE FILE	A concurrency issue (caused by a software bug, or an incorrect event timing from the hardware) causes a deadlock of kernel threads. The driver is unable to perform any operation.	PERMANENT TRANSIENT
			The device driver is unable to allocate resources (memory, I/O regions) due to a software bug in the kernel, an overload, or an error raised by the hardware. This fault causes the availability of the hardware device.	PERMANENT TRANSIENT
The Sensor Driver is unable to allocate memory (dynamic memory, I/O regions)	resource management	MEMORY		
Protocol error in the device driver, causing the abort of an I/O operation	availability	READ DATA	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The hardware device status is reset or is hung	PERMANENT INTERMITTENT TRANSIENT
Protocol error in the device driver, causing the corruption of I/O data	output value	READ DATA	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations are aborted	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.7: Sensors Drivers and Devices Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE / RESOURCE	DESCRIPTION	PERSISTENCE
Protocol error in the device driver, causing the stall of an I/O operation	timeliness	READ DATA	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations are stalled	PERMANENT INTERMITTENT TRANSIENT

TABLE B.8: Activity Manager Service Fault Model

NAME	FAILURE MODE	SERVICE / RESOURCE	DESCRIPTION	PERSISTENCE
The AM service hangs	resource management	PROCESS & THREADS	The service crashes due to some software faults	PERMANENT TRANSIENT
The AM service crashes	resource management	PROCESS & THREADS	The service stalls due to some software faults	PERMANENT
The AM service can't open or drops sockets and pipes	resource management	SOCKET / PIPE	The service is not able to open a socket/pipe or abruptly closes the already opened socket/pipe	PERMANENT TRANSIENT
The AM service drops a binder object	resource management	BINDER OBJECT	The service is not able to use the communication facilities offered by the binder	PERMANENT TRANSIENT
The AM service does not respond to the start activity request	timeliness	START ACTIVITY	The service is not able to start a new activity component	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.8: Activity Manager Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The AM service processes the start activity request with a great delay	timeliness	START ACTIVITY	The service takes too much time to start a new activity component	PERMANENT INTERMITTENT TRANSIENT
The AM service fails in starting a new activity and throw an exception	availability	START ACTIVITY	The service starts a different activity component from the expected one	PERMANENT INTERMITTENT TRANSIENT
The AM service does not respond to the finish activity request	timeliness	FINISH ACTIVITY	The service is not able to stop a activity component	PERMANENT INTERMITTENT TRANSIENT
The AM service processes the finish activity request with a great delay	timeliness	FINISH ACTIVITY	The service takes too much time to stop an activity component	PERMANENT INTERMITTENT TRANSIENT
The AM service fails in sending a finish activity request and throws an exception	availability	FINISH ACTIVITY	The service stops a different activity component from the expected one	PERMANENT INTERMITTENT TRANSIENT
The AM service returns a corrupted "sticky" intent	output value	REGISTER RECEIVER	The service register a receiver but it get a corrupted sticky intent (Intents that are "sticky" stay around after the broadcast has finished, to be sent to any later registrations)	PERMANENT INTERMITTENT TRANSIENT
The AM service does not respond to the register receiver request	timeliness	REGISTER RECEIVER	The service is not able to register a new broadcast receiver for a specified intent	PERMANENT INTERMITTENT TRANSIENT
The AM service processes the register receiver request with a great delay	timeliness	REGISTER RECEIVER	The service takes too much time to register a broadcast receiver for a specified intent	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.8: Activity Manager Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The AM service fails in sending a register receiver request and throws an exception	availability	REGISTER RECEIVER	The service registers the broadcast receiver for a different intent from the requested	PERMANENT INTERMITTENT TRANSIENT
The AM service does not respond to the broadcast intent request	timeliness	BROADCAST INTENT	The service is not able to broadcast an intent to the Android system	PERMANENT INTERMITTENT TRANSIENT
The AM service processes the broadcast intent request with a great delay	timeliness	BROADCAST INTENT	The service takes too much time to broadcast an intent to the Android system	PERMANENT INTERMITTENT TRANSIENT
The AM service fails in sending a broadcast intent request and throws an exception	availability	BROADCAST INTENT	The service broadcasts a different intent from the requested one to the Android system	PERMANENT INTERMITTENT TRANSIENT
The AM service does not respond to the bind service request	timeliness	BIND SERVICE	The service is not able to bind an activity to the requested service component	PERMANENT INTERMITTENT TRANSIENT
The AM service processes the bind service request with a great delay	timeliness	BIND SERVICE	The service takes too much time to bind an activity to the requested service component	PERMANENT INTERMITTENT TRANSIENT
The AM service fails in starting a new activity and throws an exception	availability	BIND SERVICE	The service bind an activity to a different service component from the requested one	PERMANENT INTERMITTENT TRANSIENT

TABLE B.9: Package Manager Service Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The PM service hangs	resource management	PROCESS & THREADS	The service crashes due to some software faults	PERMANENT TRANSIENT
The PM service crashes	resource management	PROCESS & THREADS	The service stalls due to some software faults	PERMANENT
The PM service can't open or drops sockets and pipes	resource management	SOCKET / PIPE	The service is not able to open a socket/pipe or abruptly closes the already opened socket/pipe	PERMANENT TRANSIENT
The PM service drops a binder object	resource management	BINDER OBJECT	The service is not able to use the communication facilities offered by the binder	PERMANENT TRANSIENT
The PM service does not respond to the get component info request	timeliness	GET COMPONENT INFO	The service is not able to retrieve the information of an installed component	PERMANENT INTERMITTENT TRANSIENT
The PM service processes the get component info request with a great delay	timeliness	GET COMPONENT INFO	The service takes too much time to retrieve the information of an installed component	PERMANENT INTERMITTENT TRANSIENT
The PM service fails in sending the get component info request and throws an exception	availability	GET COMPONENT INFO	The service cannot send information of an installed component	PERMANENT INTERMITTENT TRANSIENT
The PM service takes corrupted information of a component	output value	GET COMPONENT INFO	The service retrieves corrupted information of an installed component	PERMANENT INTERMITTENT TRANSIENT
The PM service does not respond to the check permission request	timeliness	CHECK PERMISSION	The service is not able to check a permission for an installed application	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.9: Package Manager Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE / RESOURCE	DESCRIPTION	PERSISTENCE
The PM service processes the check permission request with a great delay	timeliness	CHECK PERMISSION	The service takes too much time to check a permission for an installed application	PERMANENT INTERMITTENT TRANSIENT
The PM service always return permission grantedNot applicable	output value	CHECK PERMISSION	The service returns incorrect permission grant / deny decisions	PERMANENT INTERMITTENT TRANSIENT
The PM service does not respond to the resolve intent request	timeliness	RESOLVE INTENT	The service is not able to resolve an intent to the associated components	PERMANENT INTERMITTENT TRANSIENT
The PM service processes the resolve intent request with a great delay	timeliness	RESOLVE INTENT	The service takes too much time to resolve an intent to the associated components	PERMANENT INTERMITTENT TRANSIENT
The PM service fails in sending the resolve intent request and throws an exception	availability	RESOLVE INTENT	The service resolve an intent to the wrong associated components	PERMANENT INTERMITTENT TRANSIENT
The PM service returns corrupted resolve info	output value	RESOLVE INTENT	The service returns corrupted resolve information	PERMANENT INTERMITTENT TRANSIENT
The PM service does not respond to the install package request	timeliness	INSTALL PACKAGE	The service is not able to install a new package	PERMANENT INTERMITTENT TRANSIENT
The PM service processes the install package request with a great delay	timeliness	INSTALL PACKAGE	The service takes too much time to install a new package	PERMANENT INTERMITTENT TRANSIENT

TABLE B.10: SQLite Library Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The SQLite is unable to open the database file (e.g., due to incorrect permissions)	availability	OPEN DATABASE	SQLite returns an error when trying to open the database file	PERMANENT INTERMITTENT TRANSIENT
The SQLite fails to parse and build an SQL query due to a bug	availability	QUERY	SQLite returns an error when trying to prepare SQL database operations	PERMANENT INTERMITTENT TRANSIENT
SQLite returns an error when trying to execute SQL database operations	availability	QUERY	SQLite returns an error when performing an SQL database operations	PERMANENT INTERMITTENT TRANSIENT
SQLite is very slow during the execution of SQL database operations	timeliness	QUERY	SQLite is very slow during the execution of SQL database operations	PERMANENT INTERMITTENT TRANSIENT
SQLite returns incomplete data to the caller application	output value	QUERY	SQLite encounters an algorithmic bug (e.g., an incorrect usage of database cursors) that make the return values incomplete	PERMANENT INTERMITTENT TRANSIENT
SQLite leaks database locks, and is unable to acquire a lock	resource management	FILE	Locking operation failures in SQLite	PERMANENT INTERMITTENT TRANSIENT
Excessive growth of the SQLite database, caused by an overload or a buggy application	resource management	FILE	SQLite is unable to insert data in the database	PERMANENT INTERMITTENT TRANSIENT
SQLite corrupts the physical file containing the database with random noise	resource management	FILE	The contents of the SQLite database are corrupted	PERMANENT INTERMITTENT TRANSIENT

TABLE B.11: Bionic Library Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The bionic library can't open a file	availability	OPEN	A process cannot open a file, thus it does not get a handle from the OS	PERMANENT INTERMITTENT TRANSIENT
The bionic library delays in opening a file	timeliness	OPEN	A process open a file, but it get a handle from the OS after a great delay	PERMANENT INTERMITTENT TRANSIENT
The bionic library hangs while opening a file	timeliness	OPEN	A process open a file, but it never get a handle or any response from the OS.	PERMANENT INTERMITTENT TRANSIENT
The bionic library open the file with the wrong flags	output value	OPEN	A process open a file, but it get the handle of the file opened with the wrong flags.	PERMANENT INTERMITTENT TRANSIENT
The bionic library is returned with a wrong file descriptor	output value	OPEN	A process cannot open a file, but it get an incorrect handle from the OS.	PERMANENT INTERMITTENT TRANSIENT
The bionic library fails in reading a file	availability	READ	A process cannot read a file, and it receives an error from the function	PERMANENT INTERMITTENT TRANSIENT
The bionic library delays in reading a file	timeliness	READ	A process read a file, but it returns the read buffer after a great delay	PERMANENT INTERMITTENT TRANSIENT
The bionic library hangs while reading a file	timeliness	READ	A process read a file; but it never returns	PERMANENT INTERMITTENT TRANSIENT
The bionic library read the file with the wrong lenght	output value	READ	A process read a file, but it returns the read buffer with the wrong lenght	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.11: Bionic Library Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The bionic library is returned with a corrupted read buffer	output value	READ	A process read a file, but it returns a corrupted read buffer	PERMANENT INTERMITTENT TRANSIENT

TABLE B.12: Mount Service Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The Mount Service is stalled	resource management	PROCESS & THREADS	The service stalls due to some software faults	PERMANENT TRANSIENT
The Mount Service is crashed	resource management	PROCESS & THREADS	The service crashes due to some software faults	PERMANENT TRANSIENT
The Mount Service drops the Binder Object	resource management	BINDER OBJECT	The service is not able to use the communication facilities offered by the binder	PERMANENT TRANSIENT
The MountService cannot send command to the Volume Daemon	availability	SEND COMMAND TO VOLD	The MountService tries to make requests to the Volume Daemon, but an error is returned and the transaction is not completed	PERMANENT INTERMITTENT TRANSIENT
The MountService send a corrupted command to the Volume Daemon	output value	SEND COMMAND TO VOLD	The MountService tries to make requests to the Volume Daemon, but the request is corrupted	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.12: Mount Service Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The commands from the MountService to the Volume Daemon are delayed	timeliness	SEND COMMAND TO VOLD	The MountService tries to make requests to the Volume Daemon, but the commands are actually transmitted and handled after a long delay	PERMANENT INTERMITTENT TRANSIENT
The MountService misses an event notification from the Volume Daemon	availability	RECEIVE EVENT NOTIFICATION FROM VOLD	There is a change of state of a volume, but the MountService does not notify the change	PERMANENT INTERMITTENT TRANSIENT
The MountService corrupt an event notification from the Volume Daemon	output value	RECEIVE EVENT NOTIFICATION FROM VOLD	There is a change of state of a volume, but the MountService notifies another event	PERMANENT INTERMITTENT TRANSIENT
The MountService is slow at handling an event notification from the Volume Daemon	timeliness	RECEIVE EVENT NOTIFICATION FROM VOLD	There is a change of state of a volume, but the MountService notifies the change after a long delay	PERMANENT INTERMITTENT TRANSIENT

TABLE B.13: Volume Daemon Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
Volume Daemon hang	resource management	PROCESS &THREADS	The Volume Daemon becomes stalled due to a bug (e.g., a deadlock)	PERMANENT TRANSIENT
Volume Daemon crash	resource management	PROCESS &THREADS	The Volume Daemon crashes because of a bug (e.g., a memory management bug)	PERMANENT TRANSIENT

Continued on next page

Table B.13: Volume Daemon Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The Volume Daemon cannot open files(due to software aging)	resource management	FILE	The Volume Daemon leaks file descriptors due to a software aging issue	PERMANENT TRANSIENT
The Volume Daemon can't open or drops sockets and pipes	resource management	SOCKET / PIPE	The service is not able to open a socket/pipe or abruptly closes the already opened socket/pipe	PERMANENT TRANSIENT
Event notifications (volume inserted, mounted, ...) over the Netlink connection are lost	availability	RECEIVE NOTIFICATION FROM STORAGE	The Volume Daemon cannot retrieve events from the kernel	PERMANENT INTERMITTENT TRANSIENT
Event notifications (volume inserted, mounted, ...) over the Netlink connection are delayed	timeliness	RECEIVE NOTIFICATION FROM STORAGE	The Volume Daemon retrieves events from the kernel after a long delay	PERMANENT INTERMITTENT TRANSIENT
Event notifications (volume inserted, mounted, ...) over the Netlink connection are corrupted	output value	RECEIVE NOTIFICATION FROM STORAGE	The Volume Daemon retrieves incorrect events from the kernel, and volumes become not accessible	PERMANENT INTERMITTENT TRANSIENT
The Volume Daemon is unable to retrieve the volume configuration from vold.fstab	availability	GET CONFIGURATION (FTAB)	The Volume Daemon is unable to retrieve information about the volumes, which become not accessible	PERMANENT INTERMITTENT TRANSIENT
The Volume Daemon retrieves the volume configuration from vold.fstab with a long delay	timeliness	GET CONFIGURATION (FTAB)	Mount and unmount operations are slowed down	PERMANENT INTERMITTENT TRANSIENT
The Volume Daemon retrieves an incorrect volume configuration from vold.fstab (wrong mount point, label, partition number, sysfs path, ...)	output value	GET CONFIGURATION (FTAB)	The Volume Daemon retrieves incorrect information about the volumes, which become not accessible	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.13: Volume Daemon Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The Volume Daemon is unable to load the correct storage driver for mounting the volume	availability	MOUNT STORAGE	The Volume Daemon tries to load a storage driver in order to mount a volume, but the driver is not found or cannot be loaded by the kernel (e.g., due to a version mismatch or missing dependency)	PERMANENT INTERMITTENT TRANSIENT
The Volume Daemon takes a long time to mount a device	timeliness	MOUNT STORAGE	The Volume Daemon takes too much time to mount a volume	PERMANENT INTERMITTENT TRANSIENT
The Volume Daemon incorrectly mount a volume	output value	MOUNT STORAGE	The Volume Daemon incorrectly mount a volume, making it inaccessible or mounted with the wrong permission	PERMANENT INTERMITTENT TRANSIENT
The user submits an invalid or corrupted OBB	output value	MOUNT STORAGE	The OBB provided by the user is incorrect (e.g., the user lacks permissions)	PERMANENT INTERMITTENT TRANSIENT
The system submits an invalid or corrupted ASEC	output value	MOUNT STORAGE	The ASEC provided by the system is incorrect (e.g., the contents are corrupted)	PERMANENT INTERMITTENT TRANSIENT

TABLE B.14: Storage Drivers and Hardware Fault Model

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
The virtual device file (for example, /dev/block/mmcblk*) cannot be accessed due to the lack of permissions	resource management	DEVICE FILE	A concurrency issue causes a deadlock of kernel threads. The driver is unable to perform any operation.	PERMANENT TRANSIENT
	resource management	MEMORY	The device driver is unable to allocate resources (memory, I/O regions) due to a software bug in the kernel, an overload, or an error raised by the hardware.	PERMANENT TRANSIENT
The storage driver is unable to allocate memory (dynamic memory, I/O regions)	resource management	MEMORY	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The hardware device status is reset or is hung	PERMANENT INTERMITTENT TRANSIENT
	availability	READ/ WRITE	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations are aborted	PERMANENT INTERMITTENT TRANSIENT
Protocol error in the device driver, causing the abort of an I/O operation	output value	READ/ WRITE	A software bug in the device driver, or an incorrect event timing from the hardware, causes an error in the I/O communication protocol with the device. The current I/O operations are stalled	PERMANENT INTERMITTENT TRANSIENT
	timeliness	READ/ WRITE		
Protocol error in the device driver, causing the stall of an I/O operation	timeliness	READ/ WRITE		PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.14: Storage Drivers and Hardware Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
Superblock corruption	output value	READ/ WRITE SUPERBLOCK	The disk superblock becomes corrupted before the operation.	PERMANENT INTERMITTENT TRANSIENT
Transient I/O error when accessing superblock	output value	READ/ WRITE SUPERBLOCK	During the operation, the disk superblock becomes corrupted.	PERMANENT INTERMITTENT TRANSIENT
Corruption of individual inode	output value	READ/ WRITE INODE	An individual inode is randomly corrupted	PERMANENT INTERMITTENT TRANSIENT
Transient I/O error when accessing inode	output value	READ/ WRITE INODE	The access to an individual inode raises an I/O error	PERMANENT INTERMITTENT TRANSIENT
The inode access is slow	timeliness	READ/ WRITE INODE	The access to an inode takes too much time	PERMANENT INTERMITTENT TRANSIENT
Corruption of groups of inodes	output value	READ/ WRITE INODE	A group of inodes is randomly corrupted	PERMANENT INTERMITTENT TRANSIENT
Transient I/O error when accessing groups of inodes.	output value	READ/ WRITE INODE	The access to a group of inodes raises I/O errors	PERMANENT INTERMITTENT TRANSIENT
The access to a group of inodes is slow.	Timeliness	READ/ WRITE INODE	The access to a group of inodes takes too much time	PERMANENT INTERMITTENT TRANSIENT
Corruption of individual data block	output value	READ/ WRITE DATA BLOCK	An individual data block is randomly corrupted	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.14: Storage Drivers and Hardware Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
Transient I/O error when accessing data block	output value	READ/ WRITE DATA BLOCK	The access to an individual data block raises an I/O error	PERMANENT INTERMITTENT TRANSIENT
The data block access is slow	Timeliness	READ/ WRITE DATA BLOCK	The access to a data block takes too much time	PERMANENT INTERMITTENT TRANSIENT
Corruption of groups of data blocks	output value	READ/ WRITE DATA BLOCK	A group of data blocks is randomly corrupted	PERMANENT INTERMITTENT TRANSIENT
Transient I/O error when accessing groups of data blocks	output value	READ/ WRITE DATA BLOCK	The access to a group of data blocks raises I/O errors	PERMANENT INTERMITTENT TRANSIENT
The access to a group of data blocks is slow	Timeliness	READ/ WRITE DATA BLOCK	The access to a group of data blocks takes too much time	PERMANENT INTERMITTENT TRANSIENT
Corruption of individual dentry block	output value	READ/ WRITE DENTRY BLOCK	An individual dentry block is randomly corrupted	PERMANENT INTERMITTENT TRANSIENT
Transient I/O error when accessing dentry block	output value	READ/ WRITE DENTRY BLOCK	The access to an individual dentry block raises an I/O error	PERMANENT INTERMITTENT TRANSIENT
The dentry block access is slow	Timeliness	READ/ WRITE DENTRY BLOCK	The access to an individual dentry block takes too much time	PERMANENT INTERMITTENT TRANSIENT
Corruption of groups of dentry blocks	output value	READ/ WRITE DENTRY BLOCK	A group of dentry blocks is randomly corrupted	PERMANENT INTERMITTENT TRANSIENT

Continued on next page

Table B.14: Storage Drivers and Hardware Fault Model – continued from previous page

NAME	FAILURE MODE	SERVICE/ RESOURCE	DESCRIPTION	PERSISTENCE
Transient I/O error when accessing groups of dentry blocks	output value	READ/ WRITE DENTRY BLOCK	The access to a group of dentry blocks raises I/O errors	PERMANENT INTERMITTENT TRANSIENT
The access to a group of dentry blocks is slow	Timeliness	READ/ WRITE DENTRY BLOCK	The access to a group of dentry blocks takes too much time	PERMANENT INTERMITTENT TRANSIENT
Single corrupted block read/write	output value	READ/ WRITE ANY	A block to read/write is corrupted during an operation	PERMANENT INTERMITTENT TRANSIENT
Transient I/O error on block read/write	output value	READ/ WRITE ANY	The controller raises an I/O exception	PERMANENT INTERMITTENT TRANSIENT
Transient I/O error on multiple block read/write	output value	READ/ WRITE ANY	The controller raises multiple I/O exceptions	PERMANENT INTERMITTENT TRANSIENT
Multiple corrupted block read/write	output value	READ/ WRITE ANY	Multiple blocks to read/write are corrupted during an operation	PERMANENT INTERMITTENT TRANSIENT
Controller hang on block read/write	timeliness	READ/ WRITE ANY	The controller is not responsive and does not perform any I/O	PERMANENT INTERMITTENT TRANSIENT

References

Life is too short to be living somebody else's dream.

— Hugh M. Hefner

- [1] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, 1997.
- [2] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No pain, no gain?: the utility of parallel fault injections," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 494–505.
- [3] D. Ferraretto and G. Pravadelli, "Efficient fault injection in qemu," in *Test Symposium (LATS), 2015 16th Latin-American*. IEEE, 2015, pp. 1–6.
- [4] R. Sasnauskas and J. Regehr, "Intent fuzzer: crafting intents of death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*. ACM, 2014, pp. 1–5.
- [5] A. Mukherjee and D. P. Siewiorek, "Measuring software dependability by robustness benchmarking," *IEEE Transactions on Software Engineering*, vol. 23, no. 6, pp. 366–378, 1997.
- [6] Capgemini, M. Focus, and Sogeti, "World quality report 2017–2018," 2017.
- [7] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, 2004.
- [8] IDC - Analyze the Future, "IDC: Smartphone OS Market Share 2016, 2015," June 2017. [Online]. Available: <http://www.idc.com/promo/smartphone-market-share/os>
- [9] Android, "Android Open Source Project," August 2017. [Online]. Available: <https://source.android.com/>
- [10] E. Weyuker, "Testing component-based software: A cautionary tale," *IEEE Software*, vol. 15, no. 5, 1998.
- [11] T. O. Vuori and Q. N. Huy, "Distributed attention and shared emotions in the innovation process: How nokia lost the smartphone battle," *Administrative Science Quarterly*, vol. 61, no. 1, pp. 9–51, 2016.
- [12] Common Vulnerability and Exposures, "CVE-2016-2060," May 2017. [Online]. Available: <https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2060>

- [13] E. Martins, C. M. Rubira, and N. G. Leme, "Jaca: A reflective fault injection tool based on patterns," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*. IEEE, 2002, pp. 483–487.
- [14] Antonio Ken Iannillo, Roberto Natella, Domenico Cotroneo, Santonu Sarkar, "A Fault Injection Tool For Java Software Application," January 2013. [Online]. Available: https://akiannillo.github.io/master_degree/Iannillo_masterthesis.pdf
- [15] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "A Survey of Software Aging and Rejuvenation Studies," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, p. 8, 2014.
- [16] S. Garg, A. V. Moorsel, K. Vaidyanathan, and K. S. Trivedi, "A Methodology for Detection and Estimation of Software Aging," in *Proc. of the 9th Intl. Symp. on Software Reliability Engineering (ISSRE)*, 1998.
- [17] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of Software Aging in a Web Server," *IEEE Trans. Reliability*, vol. 55, no. 3, pp. 480–491, 2006.
- [18] L. Silva, H. Madeira, and J. Silva, "Software Aging and Rejuvenation in a SOAP-based Server," in *Proc. of the 5th IEEE Intl. Symp. on Network Computing and Applications (NCA)*, 2006, pp. 56–65.
- [19] R. Matias and J. Paulo Filho, "An experimental study on software aging and rejuvenation in web servers," in *Computer Software and Applications Conference, 2006. COMPSAC'06. 30th Annual International*, vol. 1. IEEE, 2006, pp. 189–196.
- [20] D. Cotroneo, S. Orlando, R. Pietrantuono, and S. Russo, "A measurement-based ageing analysis of the jvm," *Software Testing, Verification and Reliability*, vol. 23, no. 3, pp. 199–239, 2013.
- [21] D. Cotroneo, R. Natella, R. Pietrantuono, and S. Russo, "Software Aging Analysis of the Linux Operating System," in *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*. IEEE, 2010, pp. 71–80.
- [22] J. Araujo, R. Matos, V. Alves, P. Maciel, F. Souza, K. S. Trivedi *et al.*, "Software Aging in the Eucalyptus Cloud Computing Infrastructure: Characterization and Rejuvenation," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 10, no. 1, p. 11, 2014.
- [23] D. Cotroneo, R. Natella, and R. Pietrantuono, "Predicting aging-related bugs using software complexity metrics," *Performance Evaluation*, vol. 70, no. 3, pp. 163–178, 2013.
- [24] D. Cotroneo, M. Grottke, R. Natella, R. Pietrantuono, and K. S. Trivedi, "Fault triggers in open-source software: An experience report," in *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 2013, pp. 178–187.
- [25] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, "Aging-related bugs in cloud computing software," in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 287–292.
- [26] M. Grottke, A. Nikora, and K. Trivedi, "An empirical investigation of fault types in space mission system software," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2010.

- [27] J. Alonso, J. Torres, J. L. Berral, and R. Gavalda, "Adaptive On-Line Software Aging Prediction based on Machine Learning," in *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE, 2010, pp. 507–516.
- [28] P. Zheng, Y. Qi, Y. Zhou, P. Chen, J. Zhan, and M. R. Lyu, "An Automatic Framework for Detecting and Characterizing Performance Degradation of Software Systems," *Reliability, IEEE Transactions on*, vol. 63, no. 4, pp. 927–943, 2014.
- [29] R. Matias, A. Andrzejak, F. Machida, D. Elias, and K. Trivedi, "A systematic differential analysis for fast and robust detection of software aging," in *Reliable Distributed Systems (SRDS), 2014 IEEE 33rd International Symposium on*. IEEE, 2014, pp. 311–320.
- [30] J. Araujo, V. Alves, D. Oliveira, P. Dias, B. Silva, and P. Maciel, "An Investigative Approach to Software Aging in Android Applications," in *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*. IEEE, 2013, pp. 1229–1234.
- [31] H. Wu and K. Wolter, "Software aging in mobile devices: Partial computation offloading as a solution," in *Software Reliability Engineering Workshops (ISSREW), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 125–131.
- [32] Q. Wang and K. Wolter, "Reducing task completion time in mobile offloading systems through online adaptive local restart," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 3–13.
- [33] Y. Kang, Y. Zhou, H. Xu, and M. R. Lyu, "PersisDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions," *arXiv preprint arXiv:1512.07950*, 2015.
- [34] S. Marcek and M. Drozda, "Predicting system failures on mobile devices," in *Proceedings of the Mediterranean Conference on Information & Communication Technologies 2015*. Springer, 2016, pp. 499–508.
- [35] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, 1990.
- [36] P. Koopman and J. DeVale, "The exception handling effectiveness of posix operating systems," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 837–848, 2000.
- [37] K. Kanoun, Y. Crouzet, A. Kalakech, A. Rugina, and P. Rumeau, "Benchmarking the Dependability of Windows and Linux Using PostMark Workloads," in *Proc. FTCS*, 2005.
- [38] Michal Zalewski, "American Fuzzy Lop (AFL)," December 2016. [Online]. Available: <http://lcamtuf.coredump.cx/afl/>
- [39] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." in *OSDI*, vol. 8, 2008, pp. 209–224.
- [40] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing," in *NDSS*, vol. 8, 2008.
- [41] C. Mulliner and C. Miller, "Fuzzing the Phone in your Phone," *Black Hat USA*, June, 2009.

- [42] H. Ye, S. Cheng, L. Zhang, and F. Jiang, "Droidfuzzer: Fuzzing the android apps with intent-filter tag," in *Proc. Intl. Conference on Advances in Mobile Computing & Multimedia*, 2013.
- [43] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of android applications on the cloud," in *Proc. 7th Intl. Wksp. Automation of Software Test (AST)*. IEEE, 2012.
- [44] A. K. Maji, F. A. Arshad, S. Bagchi, and J. S. Rellermeier, "An Empirical Study of the Robustness of Inter-Component Communication in Android," in *Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, 2012.
- [45] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: Analyzing the Android Permission Specification," in *Proc. ACM Conf. on Computer and Communications Security*, 2012.
- [46] Y. Hu and I. Neamtiu, "Fuzzy and cross-app replay for smartphone apps," in *Proc. 11th Intl. Wksp. Automation of Software Test*. ACM, 2016.
- [47] C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R.-P. Weinmann, *iOS Hacker's Handbook*. John Wiley & Sons, 2012.
- [48] W. H. Lee, M. Srirangam Ramanujam, and S. Krishnan, "On designing an efficient distributed black-box fuzzing system for mobile devices," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 31–42.
- [49] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra *et al.*, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking*. ACM, 2014, pp. 519–530.
- [50] C. Cao, N. Gao, P. Liu, and J. Xiang, "Towards Analyzing the Input Validation Vulnerabilities associated with Android System Services," in *Proc. 31st Annual Computer Security Applications Conf.* ACM, 2015.
- [51] H. Feng and K. G. Shin, "BinderCracker: Assessing the Robustness of Android System Services," *arXiv preprint arXiv:1604.06964*, 2016.
- [52] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, p. 44, 2016.
- [53] Android. (2017, Sep.) Start the emulator from the command line | android studio. [Online]. Available: <https://developer.android.com/studio/run/emulator-commandline.html>
- [54] R. Natella, D. Cotroneo, J. Duraes, H. S. Madeira *et al.*, "On fault representativeness of software fault injection," *Software Engineering, IEEE Transactions on*, vol. 39, no. 1, pp. 80–96, 2013.
- [55] Roy Longbottom, "Roy Longbottom's Android Benchmarks," June 2017. [Online]. Available: <http://www.roylongbottom.org.uk/android%20benchmarks.htm>
- [56] Nelson Guilherme M. Leme, Eliane Martins, "JACA Software Fault Injection Tool," June 2017. [Online]. Available: <http://www.ic.unicamp.br/~eliane/JACA.html>

- [57] J. Andrews, L. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. Intl. Conf. on Software Engineering*, 2005.
- [58] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software Rejuvenation: Analysis, Module and Applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on.* IEEE, 1995, pp. 381–390.
- [59] G. Carrozza, D. Cotroneo, R. Natella, A. Pecchia, and S. Russo, "Memory leak analysis of mission-critical middleware," *Journal of Systems and Software*, vol. 83, no. 9, pp. 1556–1567, 2010.
- [60] D. Cotroneo, F. Fucci, A. K. Iannillo, R. Natella, and R. Pietrantuono, "Software aging analysis of the android mobile os," in *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on.* IEEE, 2016, pp. 478–489.
- [61] J.-C. Fabre, F. Salles, M. R. Moreno, and J. Arlat, "Assessment of COTS microkernels by fault injection," in *Proc. Dependable Computing for Critical Applications 7*, 1999.
- [62] S. Winter, C. Sârbu, N. Suri, and B. Murphy, "The impact of fault models on software robustness evaluations," in *Proc. Intl. Conf. on Software Engineering*, 2011.
- [63] D. Cotroneo, D. Di Leo, F. Fucci, and R. Natella, "SABRINE: State-Based Robustness Testing of Operating Systems," in *Proc. IEEE/ACM Intl. Conf. on Automated Software Engineering*, 2013.
- [64] N. Kropp, P. Koopman, and D. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *Proc. Intl. Symp. on Fault-Tolerant Comp.*, 1998.
- [65] Google, "syzkaller - linux syscall fuzzer," May 2017. [Online]. Available: <https://github.com/google/syzkaller>
- [66] S. Bhansali, W.-K. Chen, S. De Jong, A. Edwards, R. Murray, M. Drinić, D. Mihočka, and J. Chau, "Framework for instruction-level tracing and analysis of program executions," in *Proc. 2nd Intl. conference on Virtual Execution Environments.* ACM, 2006.
- [67] E. Bounimova, P. Godefroid, and D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production," in *Proc. 2013 Intl. Conference on Software Engineering*, 2013.
- [68] Android, "Intent | Android Developer," May 2017. [Online]. Available: <https://developer.android.com/reference/android/content/Intent.html>
- [69] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *Acm sigplan notices*, vol. 46, no. 4, pp. 53–64, 2011.
- [70] L. Luo, Q. Zeng, C. Cao, K. Chen, J. Liu, L. Liu, N. Gao, M. Yang, X. Xing, and P. Liu, "Context-aware System Service Call-oriented Symbolic Execution of Android Framework with Application to Exploit Generation," *arXiv preprint arXiv:1611.00837*, 2016.
- [71] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, J. Laprie, E. Martins, and D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications," *IEEE Trans. on Software Engineering*, vol. 16, no. 2, 1990.
- [72] J. Durães and H. Madeira, "Emulation of Software faults: A Field Data Study and a Practical Approach," *IEEE Trans. on Software Engineering*, vol. 32, no. 11, 2006.

- [73] J. Christmansson and R. Chillarege, "Generation of an Error Set that Emulates Software Faults based on Field Data," in *Proc. Intl. Symp. on Fault-Tolerant Comp.*, 1996.
- [74] J. Voas, F. Charron, G. McGraw, K. Miller, and M. Friedman, "Predicting How Badly 'Good' Software Can Behave," *IEEE Software*, vol. 14, no. 4, 1997.
- [75] P. Marinescu and G. Candea, "LFI: A practical and general library-level fault injector," in *Proc. Intl. Conf. on Dependable Systems and Networks*, 2009.
- [76] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "FATE and DESTINI: A Framework for Cloud Recovery Testing," in *Proc. USENIX Symposium on Networked Systems Design and Implementation*, 2011.
- [77] A. Lanzaro, R. Natella, S. Winter, D. Cotroneo, and N. Suri, "An Empirical Study of Injected versus Actual Interface Errors," in *Proc. ACM Intl. Symp. Soft. Testing and Analysis (ISSTA)*, 2014, pp. 397–408.
- [78] K. Yaghmour, *Embedded Android: Porting, Extending, and Customizing*. "O'Reilly Media, Inc.", 2013.
- [79] D. Cotroneo, A. Lanzaro, R. Natella, and R. Barbosa, "Experimental analysis of binary-level software fault injection in complex software," in *Proc. European Dependable Computing Conf.*, 2012.
- [80] D. Cotroneo, A. Lanzaro, and R. Natella, "Faultprog: Testing the Accuracy of Binary-Level Software Fault Injection," *IEEE Transactions on Dependable and Secure Computing*, 2016.
- [81] D. Powell, "Failure mode assumptions and assumption coverage," in *Predictably Dependable Computing Systems*. Springer, 1995, pp. 123–140.
- [82] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using fiat," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, 1990.
- [83] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.
- [84] D. P. Siewiorek, J. J. Hudak, B.-H. Suh, and Z. Segal, "Development of a benchmark to measure system robustness," in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*. IEEE, 1993, pp. 88–97.
- [85] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*. Prentice Hall Press, 2014, ch. 10.8.
- [86] AndroidXRef, "Androidxref," August 2017. [Online]. Available: <http://androidxref.com/>
- [87] A. ETSI, "command set for gsm mobile equipment (me)," *ETSI*, vol. 300, p. 642.
- [88] Android, "Configure on-device developer options | android studio," May 2016. [Online]. Available: <https://developer.android.com/studio/debug/dev-options.html>
- [89] —, "Ui/application exerciser monkey | android studio," May 2016. [Online]. Available: <https://developer.android.com/studio/test/monkey.html>

- [90] Android Developers, "Keeping Your App Responsive," May 2017. [Online]. Available: <https://developer.android.com/training/articles/perf-anr.html>
- [91] —, "Write and View Logs with Logcat," May 2017. [Online]. Available: <https://developer.android.com/studio/debug/am-logcat.html>
- [92] —, "Logcat Command-line Tool," May 2017. [Online]. Available: <https://developer.android.com/studio/command-line/logcat.html>
- [93] D. C. Montgomery, *Design and analysis of experiments*. John Wiley & Sons, 2008.
- [94] M. Grottke and K. Trivedi, "Fighting Bugs: Remove, Retry, Replicate, and Rejuvenate," *IEEE Computer*, vol. 40, no. 2, pp. 107–109, 2007.
- [95] J. Alonso, R. Matias, E. Vicente, A. Maria, and K. S. Trivedi, "A comparative experimental study of software rejuvenation overhead," *Performance Evaluation*, vol. 70, no. 3, pp. 231–250, 2013.
- [96] M. Grottke, D. S. Kim, R. Mansharamani, M. Nambiar, R. Natella, and K. S. Trivedi, "Recovery From Software Failures Caused by Mandelbugs," *IEEE Transactions on Reliability*, vol. 65, no. 1, pp. 70–87, 2016.
- [97] R. Matias Jr and P. Freitas, "An Experimental Study on Software Aging and Rejuvenation in Web Servers," in *Proc. of the 30th Intl. Computer Software and Applications Conference (COMPSAC)*, vol. 01, 2006, pp. 189–196.
- [98] P. K. Sen, "Estimates of the regression coefficient based on kendall's tau," *Journal of the American Statistical Association*, vol. 63, no. 324, pp. 1379–1389, 1968.
- [99] H. Theil, "A rank-invariant method of linear and polynomial regression analysis," in *Henri Theil's Contributions to Economics and Econometrics*. Springer, 1992, pp. 345–381.
- [100] W. Pirie, "Spearman rank correlation coefficient," *Encyclopedia of statistical sciences*, 1988.
- [101] F. J. Anscombe, "The validity of comparative experiments," *Journal of the royal statistical society. series A (General)*, vol. 111, no. 3, pp. 181–211, 1948.
- [102] W. W. Daniel, "Kruskal-wallis one-way analysis of variance by ranks," *Applied Non-parametric Statistics*, pp. 226–234, 1990.
- [103] Android, "Developers - keeping your app responsive," May 2016. [Online]. Available: <https://developer.android.com/training/articles/perf-anr.html#Reinforcing>
- [104] M. Grottke, R. Matias, and K. S. Trivedi, "The fundamentals of software aging," in *IEEE Proceedings of Workshop on Software Aging and Rejuvenation, in conjunction with ISSRE*. Seattle, WA, 2008.
- [105] Android, "Developers - investigating your ram usage," May 2016. [Online]. Available: <https://developer.android.com/studio/profile/investigate-ram.html#LogMessages>
- [106] Android, "Android Security Bulletin," May 2017. [Online]. Available: <https://source.android.com/security/bulletin/>
- [107] LG, "LG Security Bulletins," March 2017. [Online]. Available: https://lgsecurity.lge.com/security_updates.html

- [108] Motorola, “Moto Security Updates,” March 2017. [Online]. Available: https://motorola-global-portal.custhelp.com/app/software-upgrade-security/g_id/5593
- [109] Samsung, “Samsung Android Security Updates,” March 2017. [Online]. Available: <http://security.samsungmobile.com/smrupdate.html>
- [110] Android Studio, “Android Debug Bridge,” April 2017. [Online]. Available: <https://developer.android.com/studio/command-line/adb.html>
- [111] ARM, “CoreSight on-chip trace and debug,” May 2017. [Online]. Available: <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.coresight/index.html>
- [112] Ole André V. Ravnås, “Frida,” February 2017. [Online]. Available: <https://www.frida.re>
- [113] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in *Proc. 2016 ACM SIGSAC Conference on Computer and Communications Security*.
- [114] AndroidXRef, “Cross Reference: Intent.java,” May 2017. [Online]. Available: http://androidxref.com/7.0.0_r1/xref/frameworks/base/core/java/android/content/Intent.java
- [115] AndroidXRef, “Cross Reference: IBinder.java - linkToDeath,” May 2017. [Online]. Available: http://androidxref.com/7.0.0_r1/xref/frameworks/base/core/java/android/os/IBinder.java#257
- [116] Samsung, “WE VoIP Application for Business,” May 2017. [Online]. Available: <http://www.samsung.com/us/business/business-communication-systems/unified-communication-solutions/IPX-LSMP/STD>
- [117] N. Nethercote and J. Seward, “Valgrind: A framework for heavyweight dynamic binary instrumentation,” in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007.
- [118] Android, “Android - 8.0 oreo,” September 2017. [Online]. Available: <https://www.android.com/versions/oreo-8-0/>
- [119] Things. (2017, Sep.) Android things | android things. [Online]. Available: <https://developer.android.com/things/index.html>
- [120] Android, “Android,” August 2017. [Online]. Available: <https://www.android.com/>

```
arm64_writer_put_label (&cw, the_end)
```