Statistical Path Coverage for Non-Deterministic Complex Safety-Related Software Testing

DISSERTATION
to obtain the degree of Doctor
of Engineering

submitted by M.Sc. Imanol Allende

submitted to the School of Science and Technology
of the University of Siegen
Siegen 2022

Supervisor and first appraiser
Prof. Dr. Roman Obermaisser
University of Siegen

Second appraiser
Prof. Dr. Christof Fetzer
Dresden University of Technology

Date of the oral examination
16. May 2022

Acknowledgements

I would like to thank my supervisor, Professor Dr. Roman Obermaisser, as well as Dr. Jon Perez and Lisandro G. Monsalve, who gave me the opportunity to develop this thesis in collaboration with the University of Siegen and Ikerlan Research Centre. I would also like to express my deepest gratitude to Nicholas Mc Guire for his endless support. Mila esker nire Ikerlaneko familiari. Taldekide bikainak izan zarete, potentzial izugarriarekin. Plazer bat izan da zuekin lan egitea. 同样感谢来自兰州大学的同事唐先智、徐帅鑫和陆笛，很荣幸有机会与你们共事。

Eskerrik asko nire lagunei, azkeneko urte hauetan zuen adiskidetasuna eskaini didazuen guztioi. Bakoitzak bere aletxoa ipini du lan honetan. Atal berezi bat zor diet Aitor eta Josuri, urteak pasa ahala elkarrekin jarraitzen dugu batak besteari lagunduz. Ezin ahaztu Egoitz, inspirazio iturri bat izan zara. Baita izeko Ana eta amama ere, beti eskertuko dizuet nigatik egin duzuen guztia, asko izan baita. Nire eskerrik beroena Amaiari, emandako eta ematen didazun babes guztiagatik. Etapa honek eman dizkigun bizipenak erabil ditzagun elkarrekin ibiltzen jarraitzeko. Azkenik, bihotz bihotzez, mila esker nire arreba Edurne, aita eta amari. Erabaki guztiak beti batera hartu ditugu eta beti aurrera goaz elkarrekin. Beraz, hau ere, laurona da.

Eskerrik asko denoi!

Abstract

Emerging technologies in the embedded domain enable the development of innovative software-driven solutions. Autonomous systems are a clear example of this trend and have attracted considerable attention from different industrial sectors and research fields. In fact, they can be considered game-changers for several domains, even for the functional safety domain. These innovative safety-related systems are characterized by an increasing software complexity and high-performance requirements. Hence, a desirable requirement is to deploy an Operating System (OS), such as Linux, on these next-generation complex safety-related systems to fully take advantage of its benefits (e.g., security, reliability, software updates, performance). However, implementing a software layer, such as the Linux kernel, on a resource-sharing architecture hinders the verification process so that it is no longer feasible to base it on traditional approaches, most notably on testing. The potential of traditional testing lies in achieving exhaustive coverage, which is extremely difficult (if even feasible) in the systems with the complexity of those being developed today. Therefore, we believe that testing of software elements needs to be combined with analysis to pave the way towards safety assurance.

This thesis contributes with a novel statistical analysis technique to quantify the execution path coverage of the Linux kernel and for estimating the risk entailed by untested execution paths. In the first part, the main gaps in the field of test coverage are examined, specially focused on the Linux kernel. Afterward, different research activities are conducted to statistically estimate the test coverage by the analysis of the execution paths traversed during the testing campaign. The inherent non-determinism of the Linux kernel and the viability of estimating the coverage with different methods is further demonstrated. An additional statistical method to calculate the execution probability of untested paths and determine the risk they entail is proposed. Finally, a technique that combines all these contributions in order to quantify the testing process and the risk associated with the uncovered paths.

With the above contributions, this thesis proposes moving towards a statistical approach to quantify the coverage and the risk, and bridge the gap towards the certification of safety-related complex applications based on Linux or other complex OS that run on Commercial Off-The-Shelf (COTS) multi-core devices.

Zusammenfassung

Aufkommende Technologien im eingebetteten Bereich ermöglichen die Entwicklung innovativer softwaregesteuerter Lösungen. Autonome Systeme sind ein deutliches Beispiel für diesen Trend und haben in verschiedenen Industriezweigen und Forschungsbereichen große Aufmerksamkeit erregt. In der Tat können sie als bahnbrechend für mehrere Bereiche angesehen werden, auch für den Bereich der funktionalen Sicherheit. Diese innovativen sicherheitsrelevanten Systeme sind durch eine zunehmende Softwarekomplexität und hohe Leistungsanforderungen gekennzeichnet. Daher ist es wünschenswert, ein Betriebssystem wie Linux auf diesen komplexen sicherheitsrelevanten Systemen der nächsten Generation einzusetzen, um deren Vorteile (z. B. Sicherheit, Zuverlässigkeit, Software-Updates, Leistung) voll auszuschöpfen. Die Implementierung einer Softwareschicht wie des Linux-Kernels auf einer Architektur mit gemeinsamer Ressourcennutzung erschwert jedoch den Verifizierungsprozess, so dass es nicht mehr möglich ist, ihn auf herkömmliche Ansätze zu stützen, vor allem auf Tests. Das Potenzial traditioneller Tests besteht darin, eine vollständige Abdeckung zu erreichen, was bei Systemen mit der Komplexität der heute entwickelten Systeme äußerst schwierig ist (wenn überhaupt möglich). Daher sind wir der Meinung, dass das Testen von Softwareelementen mit der Analyse kombiniert werden muss, um den Weg zur Sicherheitsgewährleistung zu ebnen.

In dieser Arbeit wird ein neuartiges statistisches Analyseverfahren zur Quantifizierung der Ausführungspfadabdeckung des Linux-Kernels und zur Abschätzung des Risikos durch nicht getestete Ausführungspfade vorgestellt. Im ersten Teil werden die wichtigsten Lücken im Bereich der Testabdeckung untersucht, wobei der Schwerpunkt auf dem Linux-Kernel liegt. Danach werden verschiedene Forschungsaktivitäten durchgeführt, um die Testabdeckung durch die Analyse der während der Testkampagne durchlaufenen Ausführungspfade statistisch abzuschätzen. Der inhärente Nicht-Determinismus des Linux-Kernels und die Durchführbarkeit der Schätzung der Testabdeckung mit verschiedenen Methoden wird weiter demonstriert. Es wird eine zusätzliche statistische Methode zur Berechnung der Ausführungswahrscheinlichkeit von nicht getesteten Pfaden und zur Bestimmung des damit verbundenen Risikos vorgeschlagen. Schließlich wird eine Technik vorgeschlagen, die all diese Beiträge

kombiniert, um den Testprozess und das mit den unentdeckten Pfaden verbundene Risiko zu quantifizieren.

Mit den oben genannten Beiträgen wird in dieser Arbeit ein statistischer Ansatz zur Quantifizierung der Abdeckung und des Risikos vorgeschlagen, um die Lücke zur Zertifizierung sicherheitsrelevanter komplexer Anwendungen zu schließen, die auf Linux oder anderen komplexen Betriebssystemen basieren, die auf handelsüblichen Multicore-Geräten laufen.

Contents

List of Figures

List of Tables

List of Acronyms

ADAS Advanced Driver Assistance System

AEB Autonomous Emergency Braking

AI Artificial Intelligence

AIC Akaike Information Criterion

ALARP As Low As Reasonably Practicable

ASIL Automotive Safety Integrity Level

BIC Bayesian Information Criterion

CA Certification Authority

CDF Cumulative Distribution Function

CI Confidence Interval

CIP Civil Infrastructure Platform

CLT Central Limit Theorem

COTS Commercial Off-The-Shelf

CV Computer Vision

DAL Driving Automation Level

DB4SIL2 DataBase for Safety Integrity Level 2

DDT Dynamic Driving Task

DISA Defense Information Systems Agency

ECDF Empirical Cumulative Distribution Function

ECSS European Cooperation for Space Standardization

EU European Union

EVD Extreme Value Distribution

EVT Extreme Value Theory

FAA Federal Aviation Administration

FPGA Field Programmable Gate Array

GEV Generalized Extreme Value

GPOS General Purpose Operating System

GPU Graphics Processing Unit

HMI Human Machine Interface

HR Highly Recommended

i.i.d independent and identically distributed

IO Input/Output

IPC Inter-Process Communication

GPIO General Purpose Input/Output

LIDAR Laser Imaging Detection and Ranging

LOC Lines Of Code

LOPA Layers of Protection Analysis

LTP Linux Test Project

MC/DC Modified Condition / Decision Coverage

ML Machine Learning

MLE Maximum Likelihood Estimation

MMU Memory Management Unit

MSE	Mean Square Error
NA	Not Applicable
OS	Operating System
OSADL	Open Source Automation Development Lab
PFH	Probability of Failure per Hour
PID	Process Identifier
PLRU	Pseudo Least Recently Used
PMF	Probability Mass Function
PSA	Probabilistic Safety Assessment
pWCET	probabilistic Worst-Case Execution Time
RCU	Read-Copy Update
RTOS	Real-Time Operating System
ROS	Robot Operating System
RTOS	Real-Time Operating System
SIL	Safety Integrity Level
SLTS	Super Long Term Support
SOTIF	Safety Of The Intended Functionality
SPC	Statistical Path Coverage
SRT	Safe Response Time
UTS	UNIX Timesharing System
VFS	Virtual File System
V&V	Validation & Verification
WCS	Worst Case Scenario

Chapter 1

Introduction

Autonomous systems represent a significant leap forward in the ongoing technological evolution. The new technologies that compose these systems are leading to a breakthrough for different industrial sectors. Notwithstanding, they are also a major challenge for domains such as functional safety. Assuring the *"freedom of unacceptable risk"* (IEC 61508-4 Ed 2 Section 3.1.11) of the technologies involved in this type of systems is not a straightforward task since many of the measures that have been traditionally used in the safety domain are not sufficient [MGA20]. Therefore, it is necessary to investigate this type of systems from the safety perspective and study new techniques/measures to be able to assure the *"absence of catastrophic consequences on the user(s) and the environment"* [ALRL04].

This chapter provides an introduction to novel autonomous safety-related systems. It primarily focuses on the trends in next-generation safety-related systems complexity and the challenges of implementing them in use cases with functional safety requirements.

1.1 Trends in next-generation safety-related systems

Back in the 1980s, software became a disruptive technology for control systems due to its advantages compared to previous pure analog solutions [Ins11]. Since then, software-based technology has enabled the creation of a vast range of use cases and has provided a large number of benefits in distinct industrial domains. For instance, the introduction of software on control systems enabled increasing production speed and reducing costs. Now in the 2020s, these state-of-the-art systems incorporate groundbreaking technologies that allow creating novel use cases and incorporating a higher level of autonomy to the existing ones. As a result, a growing number of industries are investing in the development of these technologies, such as Artificial Intelligence (AI) algorithms and high-performance computing devices.

Just as software-based control systems, which at the time began to be deployed in industrial plants with potential threats to life or the environment, autonomous systems are currently even being deployed in use cases with functional safety requirements (e.g., autonomous vehicles) [FDW13]. Safety-related systems are defined as systems whose failure could cause human injury (even death) or property/environmental damage. Considering risk, the probability of occurrence and the severity of the potential harm, a safety-related autonomous system shall implement mechanisms to maintain its associated risks below a minimum tolerable level, as outlined in guiding functional safety standards [IEC61508].

Next-generation autonomous systems constitute a significant shift for dependable systems and safety-related systems. Thereby, these safety-related systems are characterized by (i) the integration of heterogeneous complex software applications of different safety criticality (i.e., mixed-criticality); and (ii) high-computational resource demands supported by multi-core devices, Graphics Processing Units (GPUs), and specialized accelerators (e.g., AI accelerator). Considering that safety and security are no longer independent properties in connected systems, as noted in IEC 61508-1 Ed 2 (7.4.3.3), safety-related systems face increasing demands regarding security and software update capabilities [Hob15]. However, the deployment of a system such as the Advanced Driver Assistance System (ADAS) entails several challenges for next-generation safety-related systems [Rei16]. For instance, the lack of high-performance properties of the hardware currently used in the automotive industry or more generally in the safety-related systems industry. ADASs consume Gigabytes of memory with dynamic memory allocation and have heavy use of CPU-based computing performance and hardware accelerators [BOW13]. Moreover, it is estimated that the computing performance of autonomous cars will increase significantly over the next few years [ARM]. Although widely available Commercial Off-The-Shelf (COTS) multi-core processors have been designed from the maximum average performance point of view rather than from a safety perspective, the safety industry aspires to fully employ the performance provided by these multi-core devices in their systems [PCOA+20, AAAL+15, AOP18, JGBF12, PGN+14, PGTT15].

The next-generation safety-related systems that we define in this Chapter involve the emergence of new tendencies. Among these trends, we find a leaning to develop increasingly complex systems and the need for an Operating System (OS) to fulfill the requirements of these innovative use cases.

1.1.1 Increasing complexity

The autonomous safety-related systems that we are attempting to develop today entail a significant shift from the safety systems that have been traditionally built. When we examine the use cases, we note that systems can generally be defined as highly complex systems. But, when we refer to complex systems, what do we mean?

Defining a complex system is certainly not easy. The literature provides different definitions. H. Kopetz differentiates two definitions depending on the point of view [Kop19]. On the one hand, the author identifies *object complexity*, defined as an object formed by several parts that interact with each other in different ways. On the other hand, the author cites D. Dvorak's defintion of *cognitive complexity* as the difficulty of understanding and verifying something [Dvo09]. M. Mitchel also defines complex systems in the *"Complexity - A Guided Tour"* book [Mit09].

"A system in which large networks of components with no central control and simple rules of operation give rise to complex collective behavior, sophisticated information processing, and adaptation via learning or evolution."

Whichever definition we choose, next-generation safety-related systems fall within the previous definitions. However, the definitions are far from what has been considered for traditional safety-related systems or safety standards. These systems are formed by numerous components that communicate with each other and even with external devices. Therefore, next-generation safety-related systems are no longer isolated from other devices and require external communication for operation and maintenance. This also means that security plays a key role. Although security is already taken into account by some safety standards and in traditional safety-related systems, it has not played such an important role as it does now. Security is becoming increasingly important to governments and companies. A large number of governments around the world are regulating security. For example, the European Union (EU) Cybersecurity Act was put in place to protect critical infrastructures and services (e.g., power plants, transportation systems) should be subject to security analisys and certification [EUC, eni]. Security is one of the main reasons why we need systems whose software can be remotely updated [AOP$^+$20]. After all, these systems are exposed to many threats. Therefore, it is necessary to provide mechanisms to update the system, enabling the adaptation of the system via updates. Furthermore, industry needs to take care of safety and security from the design phase of the system. Besides, these systems may execute Machine Learning (ML) algorithms that allow adapting the system too, but in this case via learning. Contrary to the traditionally static safety-related systems, these systems may deploy algorithms that are not precisely understood (e.g., ML algorithms [1]).

Finally, the deployment of different software modules or layers (e.g., OS, middleware, AI modules, mixed-criticality tasks) together with resource-sharing multi-core devices generate a highly sophisticated and interdependent information processing system. Consequently, we can state that next-generation safety-related systems are complex systems. Besides, there is also no reasonable expectation that they will revert back to low-complexity systems in the future. We can consider that we are at the beginning of complex safety-related systems and, moreover, we have no answer on how to make them safe. Thereby, there is a need to define novel techniques and methods with respect to functional safety compliance (e.g., testing strategies).

1.1.2 Why Linux?

Over the last decade, different industry domains have invested in the development of innovative and highly complex safety-related systems (e.g., autonomous vehicles) that might revolutionize multiple market sectors, including the functional safety industry. For all the several demands that involve these innovative safety-related systems, the need arises for an OS that supports the integration of these complex applications while providing functional safety capabilities.

GNU/Linux is broadly used in non-safety-critical systems. In fact, it is the leading General Purpose Operating System (GPOS) in different domains [CKH17]: 99% of the supercomputer market, 90% of the public cloud workload, 82% of the smartphone market, and 62% of the embedded market, some of which are mission-critical while not safety-related. Besides, it provides wide hardware support (including multi-core devices), security capabilities, and a large development ecosystem with a broad community of developers which supports the ever-increasing rate of code change. The latest kernel version has a rate of roughly 6 patches per hour (24/7 and 365 days/year). Additionally, the Real-Time Linux collaborative project aims to fully integrate real-time capabilities into the mainline Linux

[1]Note that many ML algorithms only satisfy a relaxed definition of algorithm

kernel [Rea20]. Safety-related systems shall exhibit well-defined behaviors and one aspect of well-behaved is the temporal domain. Most of the safety-related OSs have historically been Real-Time Operating Systems (RTOSs). Thereby, it is essential to note that the real-time patch for the Linux kernel is increasingly used in a growing number of use cases and the objective for the community is to include it in the mainline kernel [rea].

Companies and governments nowadays rely on this OS for crucial applications such as banking, telecommunications, and web servers. Governmental Agencies rely on Linux for their systems with cybersecurity requirements [lina, sus, dis]. The space domain also relies on Linux for a wide range of their systems [Lep17]. For instance, SpaceX relies on Linux for their primary flight control systems for Falcon 9 launch vehicle, Dragon spacecraft, and the ground stations [spa, Gru12, Lep17]. DLR-GSOC, the German Space Operations Center, also uses Linux for the ground station [SBG14]. Linux can even be found on Mars. NASA relies on Linux for Ingenuity, NASA's Mars helicopter landed with the Perseverance rover [GLB$^+$19]. Indeed, Ingenuity is based on a Qualcomm Snapdragon 801 COTS multi-core device. Therefore, considering the key role that GNU/Linux is successfully playing in these domains, we can perceive the potential that it could have for safety-related applications.

Bulwahn et al. analyzed the advantages of using GNU/Linux for an ADAS instead of a proprietary solution. The authors stated that it could reduce development and qualification costs while improving the quality and confidence [BOW13]. Besides, the open-source nature of the Linux kernel enables small and medium companies to build products that they would not be able to with either a proprietary or a homegrown OS. Its advantage is not only based on the fact that the OS is already developed and that the code can be reused. An interesting fact is the large number of professionals with at least a minimum knowledge on the Linux kernel. We can assume that nowadays the vast majority of computer science students in the world have at least some experience with Linux. So there is no need to train them on a proprietary or homegrown OS, since they already have the basic knowledge to write software, configure services, etc. This, after all, is translated into time to develop a final product and, therefore, economics. This causes a remarkable interest from companies of different industrial domains in achieving the certification of Linux for safety-critical systems.

SIL2LinuxMP is a project led by Open Source Automation Development Lab (OSADL) that pursues the certification and qualification of base components of an embedded GNU/Linux running on a COTS multi-core platform [PGB18, Gui19]. SIL2LinuxMP solution targets SIL2 complex systems and provides the required evidences to claim SIL2 systematic compliance with respect to IEC 61508 Ed 2 (route 3_S) [IEC61508]. Linux Foundation's ELISA project also works on the process definition to build and certify safety-related systems based on GNU/Linux [elia]. These two initiatives have significant partners from the automotive industry, which shows the interest of this sector in Linux for their next-generation safety-related systems (i.e. autonomous driving). However, it is essential to clarify that neither project is entirely focused on the automotive domain. Achieving a certifiable Linux is interesting for several domains and, thus, both projects have partners from different domains (e.g., robotics, railway).

It is essential to consider that GNU/Linux was not designed for use in safety-related systems and whenever such a GPOS is integrated on high-performance multi-core devices based on a resource-sharing architecture [PCOA$^+$20, AMGP$^+$19], a highly complex system emerges. As a result, many of the traditional methods that have been used for verification and validation in the safety domain would need to be reviewed (e.g., code test coverage). Although the Linux kernel development process is not compliant with safety standards, it has some rigorous processes that could be argued as potentially achieving the objectives stated in IEC 61508 Ed2. Thereby, it could be suitable for safety applications after a conscious in-depth analysis and documentation. It may also need some minor amendments of measures and techniques at the procedural level and minor improvements at the code level (e.g. defensive structures). As a side note, we point out that similar constraints pertain to complex hardware, which is also facing serious challenges with traditional safety-related measures and techniques when applying them to highly complex multi-core systems [PCOA$^+$20]. COTS multi-core devices are in general not designed with safety-related systems in mind.

Consequently, we can state that Linux is increasingly relied upon for mission-critical systems. In addition, if we examine the features of the next-generation safety-related systems (e.g., computing performance, concurrent computing, security, and updating capabilities), we see that GNU/Linux is a potential option. However, how can we achieve adequate safety assurance? It is plausible that the conclusion is that we cannot build safe systems at this complexity level. If that is the case for GNU/Linux then, it will likewise be the case for more or less all other OS/RTOS that could satisfy the needs that these systems state. Any alternative to Linux would face similar challenges and possibly similar solutions. In this sense, the solutions presented in this thesis are considered generalizable and not Linux-specific, even though the actual implementation is Linux-based.

1.2 Challenges with next-generation safety-related systems

The autonomous systems that we are attempting to develop today entail a significant shift from the safety systems that have been traditionally built. Whereas traditional approaches were successful and effective for the simpler systems for which they were designed, there have been considerable changes compared to these groundbreaking systems [Lev16]. Although companies are announcing the development of highly complex dependable systems, there is still a need to pave the way towards the assurance of such dependable and safety-related complex systems. Unfortunately, no standard exists that takes into account the specific features of such complex systems. The techniques and measures set recommended by current safety standards were not designed and intended for these types of systems, and the update of current safety standards and definition of new types of safety standards are in progress, though to date have not yet managed to cover this gap (e.g., ISO/PAS 21448:2019 Road vehicles - Safety Of The Intended Functionality (SOTIF) [ISO21448]).

One of the key issues on existing safety standards is that they were not developed considering these new-generation safety-related systems. Most notably, algorithmic uncertainty management had not even been conceived for diagnostics. Safety standards were created focused on simple single-core systems (IEC 61508-3 Annex F). Nevertheless, the scalability of performance is no longer feasible by clock frequency and, hence, higher-computing performance devices are required. Although widely available state-of-the-art COTS multi-core devices and GPOSs have been designed for maximum average performance and not from a safety perspective, the safety industry aspires to employ the performance provided by these multi-core systems and the advantages of using a GPOS in their safety-critical systems [PCOA$^+$20].

The ideal for a safety-related system still is to (i) know all failure modes, (ii) understand all failure conditions, and (iii) have adequate evidence to justify the assumptions (IEC 61508-2 7.4.4.1.2/7.4.4.1.3)). Nevertheless, this is generally considered not technically feasible for next-generation autonomous safety systems [MGA20]. For instance, imagine that we execute an ML algorithm, are we able to identify all failure modes, understand them and provide adequate evidence about our assumptions? Unfortunately, no. Or, at least, not yet. Consequently, for complex software elements, an alternative approach is necessary to achieve adequate assurance.

We should note here that even for traditional safety-related systems, this ideal was never attained. No human process generating these systems was ever fault-free. What we do have are systems in which we feel that the ability of comprehension is sufficiently complete and consistent to assume that significant deviations from intent would be anticipated. In many ways, the emerging issues were always there, albeit at a level that we perceived as tolerable. However, we now realize that for the previously described novel system examples this no longer holds.

The safety domain has traditionally considered testing as one of the major methods to support the safety system assurance evidence. IEC 61508-4 Ed 2 defines dynamic testing as running software and/or hardware in a controlled mode to demonstrate the presence of the required behavior and the absence of unwanted one [IEC61508]. The results obtained from these methods have been considered adequate assurance for testing traditional safety-related applications. The key argument being that since all potential failure modes are known, the absence of the same can be confirmed by a (theoretically) exhaustive test of all expected (or permitted) use of the provided functionality. However, with the increasing complexity of the software in applications such as next-generation autonomous systems, it seems that relevant test coverage is hardly achievable (if feasible) by dynamic testing and, thus, may not be the major assurance method anymore [AMGP+21a].

For example, imagine attempting to achieve Modified Condition / Decision Coverage (MC/DC) for an AI middleware or a full-featured multi-user/multi-core OS. Accordingly, attempting to achieve adequate test coverage on an OS such as Linux is not reasonably achievable (if feasible at all). It is increasingly difficult (and economically questionable) to test GNU/Linux-based complex software systems with an adequate understanding of their behavior, by the exercise of partial inputs into a virtually infinite input space, even with a huge amount of resources. As a result, the ideal of complete coverage by testing might not be feasible for this type of software. Distinct factors constrain the complete coverage achievement in the Linux kernel:

1. **Total existing paths:** The Linux kernel can be considered quite large due to the breadth of supported hardware and the number of resources it offers. The latest kernel versions have approximately 27 million LOC [linb]. Although all these lines do not form a Linux-based system (e.g., it does not use all the drivers available in the kernel), a Linux-based safety-related system is still estimated at around 1 million LOC [OSA].

   Therefore, it is considered technically infeasible to get the 100% execution paths tested as some of them are not even exercised by the given application in the target system. Even achieving a reduced while justifiable coverage [2] seems questionable.

2. **Execution path variability:** Different studies show how the Linux kernel execution is non-deterministic [MGOS09, OMG10, OMGFOO13, OMGF14, OGOO15]. This means that the kernel does not follow the same execution path, given the same input parameters. In other words, for an identical system-call with the same input parameters, several possible execution paths may exist. Consequently, it is no longer feasible to assess the correctness of this type of systems with a unique iteration of a test-case for each combination of input parameters [GLC+15].

3. **Existent static analysis tools:** have traditionally been used to identify all possible execution paths. Although these types of tools are widely used for direct call examination,

[2]IEC 61508-3 Ed 2 Annex B.2 permits $\leq 100\%$ if justified

their applicability for kernel execution path analysis is limited. For instance, these tools have major difficulties solving indirect calls [LH19], which are commonly used in the kernel, and also provide traces formed by dead code that is never executed [Sim20]. There also is the issue of dynamic allocation and management of resources that may lead to almost infinitely deep paths that are theoretically possible but increasingly unlikely and notably impossible to deterministically trigger. Therefore, these types of tools also provide traces with a de facto zero or negligible probability of execution.

It should be mentioned that throughout this document the term *path* is used as the sequence of invoked function calls. We also consider the term *trace* a synonym of *path*.

1.3 Contributions

The multiple technical challenges of safety-related complex software, specifically of the verification phase of OSs, are the main motivation of this thesis. Subsequently, this thesis studies an alternative technique that examines the test coverage of a GPOS for a safety-related software application, as traditional testing techniques hinder the achievement of the objective. The goal of this research is to move from full path coverage towards a statistical approach. In other words, the aim is to cover the paths relevant to the specific system rather than all possible paths and, thus, quantify the assurance along with the uncertainty. In addition, we examine the cases where the 100% test coverage is not achievable in order to provide appropriate and rigorous justification. We analyze the proposed technique with a Pedestrian Detection Autonomous Emergency Braking (AEB) case study. The technique that we propose is based on three main phases: data collection, test coverage, and residual risk estimation. In order to be able to provide this technique, we contribute with different novel methods that form the SPC technique:

- **Dynamic Data Collection and Validation:** We describe a process to obtain the execution traces exercised during the testing process, specifically based on the Linux kernel. The process is based on (i) recording the execution traces during the testing process, (ii) post-processing the recorded traces to obtain a data set to perform further analyses, and (iii) validating the quantity and quality of this data set in order to perform the analyses.

- **Parametric Statistical Method for Software Execution Test Coverage:** We present a parametric statistical method to provide a quantification of test coverage for testing complex safety-related software based on (a subset of) GNU/Linux. In other words, the described method statistically estimates the maximum number of kernel execution paths that an application can exercise in order to calculate the number of not-covered execution paths. The contribution is based on modeling the execution traces' appearance behavior. For this purpose, we try to understand the mechanisms that generate the appearance of untested traces.

- **Non-Parametric Statistical Method for Software Execution Test Coverage:** We describe a statistical analysis method that allows estimating the test coverage with the remaining uncertainty. This method has the same goal as the previous one, proposing a technical solution to move towards the quantification of uncertainty based on known incomplete expectations and analysis of the variance of these expectations. The analysis is based on different non-parametric estimators that are widely used in other scientific disciplines (e.g., biology). Contrary to parametric statistics, non-parametric statistics make no assumptions about the underlying distribution of the data.
- **Execution Path Uncertainty for Safety Software Test Coverage:** As full test coverage is hardly achievable (if feasible), we propose a method for estimating the risk associated with the execution path uncertainty. The method provides a probability value for the untested (unseen) execution paths and, afterward, estimates the risk that they entail. The proposed method can be used with the As Low As Reasonably Practicable (ALARP) principle to provide the required code test coverage justification or identify the need to increase the testing depth and scope.

With these methods, the results obtained from them, and the conclusions drawn, we define in detail the novel technique that we have named Statistical Path Coverage (SPC). The thesis explains in detail the SPC technique and how the different methods complement each other to form it.

1.3.1 Additional contributions

Although this document does not directly reflect them, this thesis has made several contributions closely related to this topic.

- We provide a review [AMGP$^+$19] of the safety architecture defined in the SIL2LinuxMP project (based on Layers of Protection Analysis (LOPA)) [OSA, PGB18], using as a reference the requirements identified by multiple studies for a certifiable hypervisor [AAAL$^+$15, PGTT15, PGN$^+$14]. The contribution identifies the potential of specific mechanisms provided by the Linux kernel to isolate tasks in container-based technology, achieving a similar architecture to the one obtained with hypervisors.

- Besides, the SIL2LinuxMP architecture is also examined in the SELENE H2020 European project context [HFP$^+$20]. This research project targets the development of a RISC-V, GNU/Linux, and Jailhouse hypervisor-based safety-related computing platform to be evaluated in two case studies: an autonomous train operation system based on Computer Vision (CV) and AI, and an autonomous robot.

- Participation in a survey of COTS multi-core devices for safety-critical systems [PCOA$^+$20].

- Performance impact and achievable diagnostic coverage analysis of a catalogue of widely used checksums algorithms implemented in the Matrix-Matrix Multiplication (MMM), one of the backbone algebraic operations of ML-based autonomous systems [FPA$^+$21].

- Finally, the main constraints involved in the certification of next-generation safety-related systems have been identified. Besides, a certification assessment approach is analyzed for these systems [MGA20].

It is important to note that although this thesis is fully focused on the Linux kernel, we infer that the contributions provided by this thesis may be valuable and adjustable for other GPOSs. Nonetheless, we cannot confirm it as the necessary analyses have not been conducted.

1.4 Thesis Organization

The dissertation document is structured in nine main chapters, graphically represented in Figure 1.1 and summarized as follows:

- Chapter 2 presents a critical review of the state of the art. The analysis is performed taking into account current safety standards and main research contributions found in the literature. The chapter also presents the main gaps identified in the literature and provides a brief problem statement.

- Chapter 3 briefly describes and presents the proposed Statistical Path Coverage (SPC) technique. We present the data collection and validation process. Besides, we briefly introduce three novel methods that form this technique, which aim to pave the way towards the test coverage of next-generation safety-related autonomous systems, specifically based on the Linux kernel.

- Chapter 4 describes the case study and the data set acquisition used as a guiding example in the description of the proposed methods for the analysis of test coverage with the remaining uncertainty (Chapter 5 and Chapter 6) and estimated the execution probability of untested paths (Chapter 7). This data is obtained from the AEB case study software application running on a defined platform and system-context. Hence, the correctness of the recorded data is also validated in order to perform the analysis. The case study is a guiding example for the contributions carried during the research activities of the thesis.

- Chapter 5 describes a novel statistics-based test coverage method that aims to overcome the limitations for the testing of GNU/Linux-based safety systems. The objective is to statistically estimate the number of kernel paths to be able to quantify the uncertainty provided by the unknown paths by exercising the target-system in its system-context.

- Chapter 6 analyzes different non-parametric estimators with the objective of estimating the total number of traces that have a relevant probability of appearing in a specific use case. This method allows quantifying the test coverage.

- Chapter 7 describes the proposed method to estimate the Linux kernel execution path uncertainty during the testing phase of a Linux-based safety-critical system, leading to the probability (estimation) of unseen execution traces which, following the ALARP principle, shall be minimized. In addition, if safety standards such as IEC 61508 Ed 2 are taken into account, in the case that coverage is $\leq 100\%$, an appropriate justification is required. Consequently, this chapter describes a method that may be of significant value for next-generation safety-related systems that need to provide a justification of test coverage incompleteness.

- Chapter 8 examines the results obtained from the different proposed methods and explains how they complement. It describes the SPC technique in detail based on the proposed methods and the obtained results. Moreover, it compares the result ranges obtained from Chapters 5 and 6 with the intention of validating and verifying their concordance and, hence, the adequacy of the proposed methods.

- Chapter 9 presents the main conclusions and potential future work lines.

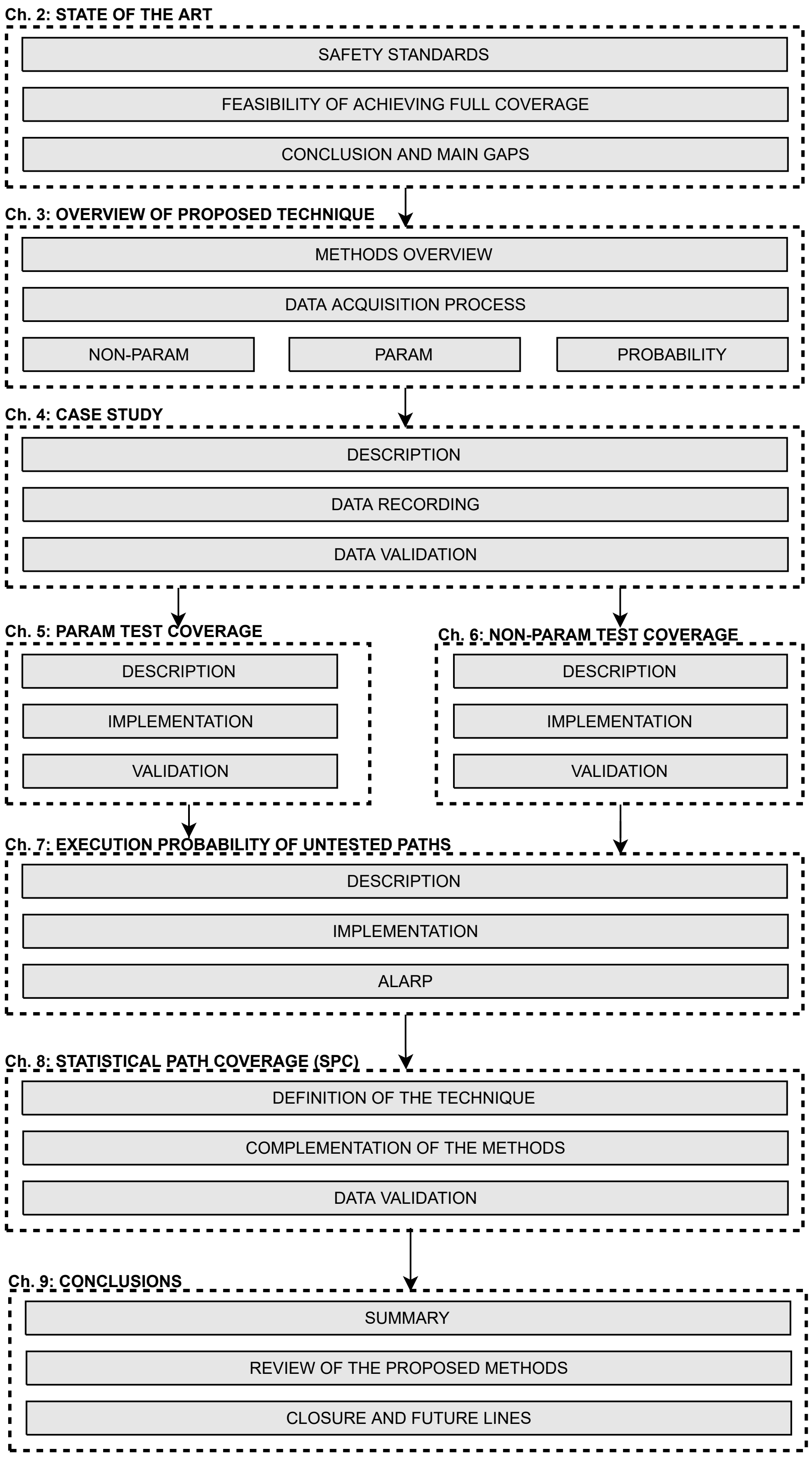


Figure 1.1 Structure of the dissertation document.

Chapter 2

State of the Art

During the last decades, existing safety standards have allowed ensuring the safety of socio-technical systems or, equivalently, guarantee a tolerable residual risk of the system. IEC 61508 is widely considered a generic international safety standard used as a reference for multiple domain-specific standards such as automotive, machinery, industry, railway, and medical domains. Notwithstanding, when these standards were developed, current autonomous systems did not even exist yet and, therefore, the standards were not created for systems with the level of complexity that characterize next-generation safety-related systems. As a result, it is essential to examine these standards to identify suitability or limitations with current safety-related autonomous systems. In this chapter, we try to answer some key questions to perform a critical state of the art review.

- Which systems do safety standards describe?
- Which measures/techniques do these standards describe?
- Are these standards completely valid for next-generation safety-related systems?
- Which are the limitations of these measures/techniques for next-generation safety-related systems?

2.1 Functional safety standards

Safety is one of the five attributes that comprise dependability along with availability, reliability, integrity, and maintainability [ALRL04]. Functional safety is generally defined as the *"absence of catastrophic consequences on the user(s) and the environment"* [ALRL04], *"the state in which risk is acceptable"* [ARP4754A] or within a functional standard such as IEC 61508 as *"freedom from unacceptable risk"* (IEC 61508-4 Ed2 Clause 3.1.11) [IEC61508]. Furthermore, system safety is commonly defined as *"the application of engineering and management principles, criteria, and techniques to achieve acceptable risk"* (MIL-STD-882E Clause 3.2.13) [oD12].

Safety tasks are categorized in different criticality levels, which are determined depending on the risk associated with the system, to implement appropriate mechanisms to maintain these associated risks below a minimum tolerable level. Each safety standard has its own criticality level categorization nomenclature: (i) IEC 61508 establishes it with a Safety Integrity Level (SIL) ranging from 1 (lowest criticality) to 4 (highest criticality) for (ii) ISO 26262 uses from Automotive Safety Integrity Level (ASIL) A (lowest criticality) to D (highest criticality) for automotive safety standards. The higher the criticality, the lower the permissible probability of dangerous failures, e.g., a SIL 4 has associated a probability of dangerous failure in the range of 10^{-8} to 10^{-9} hours of operation (approximately once every 14.155 to 114.155 years). Generally, a higher criticality level entails a higher cost effort for certification [AOP18, PCOA^{+}20].

IEC 61508 is broadly accepted as the generic international safety standard [IEC61508] and it is also the reference standard for specific domain standards, such as nuclear (IEC 61513 [IEC61513]), automotive (ISO26262 [ISO26262]), railway (EN 5012X [EN50126, EN50128, EN50129]), process industry (EN 61511 [IEC61511]) and elevator (ISO 22201 [ISO22201]). Other domains, including space and avionics, are based on their own independent standard (DO 178, European Cooperation for Space Standardization (ECSS), etc). Nevertheless, the widespread consensus established in the literature is that there is no fully appropriate standard for highly complex safety-related systems [KFFW19, PCOA^{+}20, AOP18, PGTT15]. Generally, safety standards lag behind state-of-the-art technologies and, thus, they are not still prepared for next-generation safety-related systems [AAAAC17]. Although IEC 61508 is considered the generic international safety standard, it is mainly defined for single-core architectures and, thus, it provides limited guidance for multi-core systems [AAAAC17, PCOA^{+}20, AOP18, PGTT15]. Besides, domain standards were designed targeting specific use cases. The standard describes systems without the need for software updates. Traditional safety systems are rarely modified and if a modification is necessary, re-certification entails a high effort and cost [AOP^{+}20]. There is neither a standard available that provides an explicit guide for AI safety development and certification [OY17]. Thereby, there is no standard available that takes into account the specific properties of next-generation complex systems. But, which standard should we take as a reference for this study?

2.1.1 Scope of the standards

With the growing interest of the automotive domain in these types of systems, it may seem reasonable to use ISO 26262 as the reference standard for autonomous systems. However, basing our study on a domain-specific standard limits the applicability of the use cases and entails major issues as these domain standards were developed with very specific use cases in mind (e.g., airbag, brake light). For instance, ISO 26262 was developed for vehicles with *"a human driver responsible for safe operation"* [Koo19]. If we use a domain standard, we are dealing with the challenges that entail a highly autonomous system and a disruptive change of the domain itself. Roughly speaking, the autonomous vehicles that the automotive industry is developing have little to do with the vehicles we have known so far. Note that we do not mean that the domain standards are inadequate, but that they are not intended for autonomous systems and offer little flexibility as they are focused on certain systems. Due to the need for a safety standard for a highly autonomous system, committees initiated by companies and governments are working on updating current safety standards and on the definition of new ones to cover the assurance of these systems. Recent standards such as ISO/PAS 21448:2019 Road vehicles – SOTIF [ISO21448] and UL 4600 [KFFW19] pave the way towards the safety assurance of ADAS, though without currently achieving a comprehensive coverage.

Even though IEC 61508 does not target highly autonomous systems either, literature and different initiatives consider that the flexibility provided by this standard makes it a suitable candidate to focus on [MGA20, Gui18a, OSA, elia]. As they are generic guides, we believe they are more versatile for different use cases. It is also essential to note that safety standards provide guidance and that they cannot be considered guidelines.

Besides, IEC 61508 standard takes into account from a general point of view the complexity of the system and the degree of novelty of design in order to achieve safety. The IEC 61508-2 Ed2 safety standard differentiates between two types of systems [IEC61508]: Type-A and Type-B. On the one hand, a system is classified as Type-A if (i) all failure modes are identified, (ii) failure behavior is known, and (iii) suitable failure data is available to report the claimed ratios. On the other hand, a system is classified as Type-B when any previous points are not satisfied. Hence, it is possible to appreciate that the standard makes a distinction between systems of different complexity. Type-A systems are defined as low-complex systems, whereas Type-B systems are mid-complex or high-complex systems.

Other standards, notably those derived from IEC 61508, follow a very similar definition approach except for minor variations. The differences focus on industry-specific knowledge that allows mitigating some of the complexity issues. In particular, the machine tools standard (IEC 62061) notes that a Type-B subsystem that is certified according to IEC 61508 may be treated as a Type-A system in the context of an aggregated system. This establishes a way for complex elements that have well-defined interfaces to be aggregated into larger systems. Even though the ideal for a safety-related system still is to fulfill the three attributes of Type-A, this is generally considered not technically feasible for next-generation autonomous safety systems [MGA20].

Consequently, the research activities conducted in this thesis consider IEC 61508 the reference standard with the aim of extending the contribution to other safety domains. Besides, IEC 61508 and ISO 26262 (automotive) define equivalent testing techniques [ISO26262, IEC61508].

2.1.2 Testing process

Analyzing the techniques and measurements proposed by the standards also allows identifying the types of systems they are intended for and their limitations with state-of-the-art systems. According to the IEC 61508 Ed2 safety standard, software module testing, integration testing and system testing are phases that form the safety life cycle. Testing allows identifying existent faults and, hence, it is possible to fix or mitigate them. Besides, the testing process normally enhances the understanding of the system's behavior.

IEC 61508-3 lists a number of techniques for software testing and integration in Table A.5 [IEC61508]. Among them, the standard highly recommends the following techniques for SIL 2 [IEC61508]:

- **Dynamic analysis and testing:** the execution of hardware and/or software in a systematic or controlled way, with the objective of proving that the intended behavior is present and unintended behavior is absent (IEC 61508-7 B.6.5).
- **Functional Testing:** aims to identify failures throughout the specification and design phases so that they do not occur during software/hardware implementation and integration (IEC 61508-7 B.5.1).
- **Black box testing:** checking the dynamic response under a real environment with the aim of identifying failures in the fulfillment of functional specifications and evaluating utility and robustness (IEC 61508-7 B.5.2).

- **Data recording and analysis:** the documentation of all data, decisions, and rationale in the software project with the aim of facilitating verification, validation, evaluation, and maintenance (IEC 61508-7 C.5.2).
- **Test management and automation tools:** the implementation of an exhaustive and systematic approach for software and system testing by the use of appropriate tools (IEC 61508-7 C.4.7)

IEC 61508-3 Ed2 standard lists specific techniques for dynamic and black-box testing in Table B.2 and Table B.3 [IEC61508]. The first technique Highly Recommended (HR) in Table B.2 for a SIL 2 is *"Test case execution from boundary value analysis"*, which is also recommended by Table B.3 *"Equivalence classes and input partition testing, including boundary value analysis"*. In other words, the test-cases are formed from the data of permitted ranges, of not allowed ranges, of the boundary area, extreme values, and the combination of these classes. The objective of these techniques is to exercise the application with all possible value types it can be exercised. But, when is testing enough? The testing process is measured by test coverage, referenced in IEC 61508-3 Table B.2. The standard differentiates four types of coverage:

1. **Structural test coverage (entry points):** quantifies the ratio of functions that are called at least ones.
2. **Structural test coverage (statements):** quantifies the ratio of statements that are called.
3. **Structural test coverage (branches):** quantifies the ratio of branches that are executed.
4. **Structural test coverage (conditions, MC/DC):** measures the proportion of operated conditions of a composite conditional branch.

For a SIL 2 system the standard highly recommends (i) structural test coverage (entry points) and (ii) structural test coverage (statements), which is also known as *execution path or trace coverage*. In other words, quantifies the execution traces (i.e., the sequence of functions) that have been followed. Moreover, the standard includes a statement for the four items: *"Where 100 % coverage cannot be achieved (e.g., statement coverage of defensive code), an appropriate explanation should be given"* [IEC61508].

Testing of safety-related systems is a normatively required process by functional safety standards. In the case of those relating to software, it has gained great importance in safety assurance due to the impossibility of formally verifying the absence of bugs. Although the majority of the systematic failures can be excluded when a system is thoroughly tested, it is not possible to assert that an element is *free of faults* unless, at least, exhaustive coverage is achievable. Software testing faces its challenges, especially with complex software elements. Exhaustively testing a system involves exercising the whole input range vector of the system, something that is significantly difficult (if possible) in highly complex systems. Consequently, is it possible to achieve full coverage?

2.2 Feasibility of achieving 100 % test coverage

Test coverage analysis is one of the major measures required for safety-related systems as it quantifies the amount of code that is executed during the testing process. Achieving 100% test coverage on traditional safety-related systems is considered a state-of-the-art requirement for most (current) safety-critical systems (SIL1 or above). However, achieving full coverage usually involves a tremendous effort. The ninety-ninety rule is a humorous aphorism by Tom Cargill, which defines the effort that implies attempting to achieve 100% test coverage: *"The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time"* [Ben85].

Prause et al. examined the effort-efficiency balance of attempting 100% code coverage in the space domain [PWH+17]. The authors concluded that the break-even point of effort-efficiency is between 80% and 95%. Achieving coverage above this point is considered costly and could even lead to new risks. The research also concludes that *"100% should not become a 'formal' goal only"* because otherwise, we only focus on enhancing the value. The objective must be to perform adequate testing and to consider code coverage a result of the testing.

Compared to traditional safety systems, autonomous safety-related systems that we are trying to develop currently generally consist of a larger number of code lines. Besides, the code they execute (e.g., application, OS) is usually specifically developed for safety (IEC 61508 route 1_S). However, we now have much more extensive coding that may have not followed a safety development process. Thereby, some questions arise with the highly complex safety-related systems:

- Is it viable to achieve full coverage in a next-generation safety-related system?
- Is it possible to achieve full coverage on a system based on the Linux kernel?
- Are we able to call all functions and force the execution of all code statements contained in the kernel?

In this chapter, we analyze the viability of achieving full coverage in complex systems, specifically focused on systems based on the Linux kernel. We critically review the literature to identify the main gaps by examining the features that may limit covering the code during the verification phase.

2.2.1 Total existing execution paths

The newest released Linux kernel version has over 27 million LOC and approximately a patch rate of 6 per hour. Besides, as a result of having a worldwide community, Linux kernel development is 24/7 and 365 days/year [kerb, kera]. If we analyze the evolution of the Linux kernel source code (see Figure 2.1) [kerb], we observe that the LOC that form the kernel increase continuously over the released versions. Note that around half of the lines correspond to drivers [kera] since the Linux kernel provides wide hardware support. In addition, the Linux kernel is highly configurable, being able to compile a kernel only with the desired facilities. Thus, it is possible to reduce the kernel size only to the essential core components for the safety-related system. It has been estimated that a safety-related system based on Linux would contain roughly 1 million of those LOC [OSA]. However, we are still talking about an extended source code that includes a huge number of different execution paths.

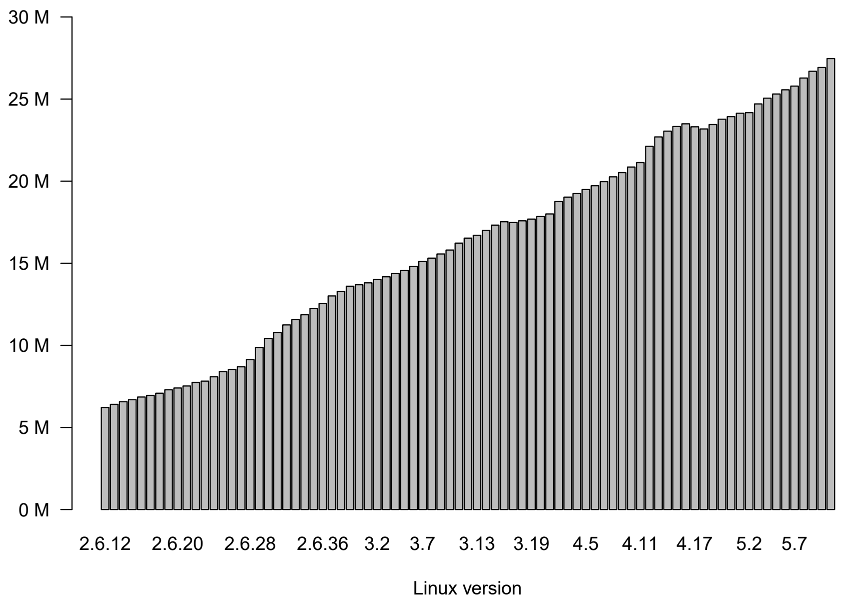


Figure 2.1 LOC of the Linux kernel in each released version [kerb].

Figure 2.2 illustrates a rough example of the kernel architecture with the possible kernel execution paths of a specific application. Dashed lines represent the paths that the application can exercise, and the straight lines are those paths that exist in the kernel but which are not executable from the application under test. Figure 2.2 considers each circle a distinct function and they are categorized in different colors depending on the subsystem they own (e.g., init, fs, block, mm, kernel).

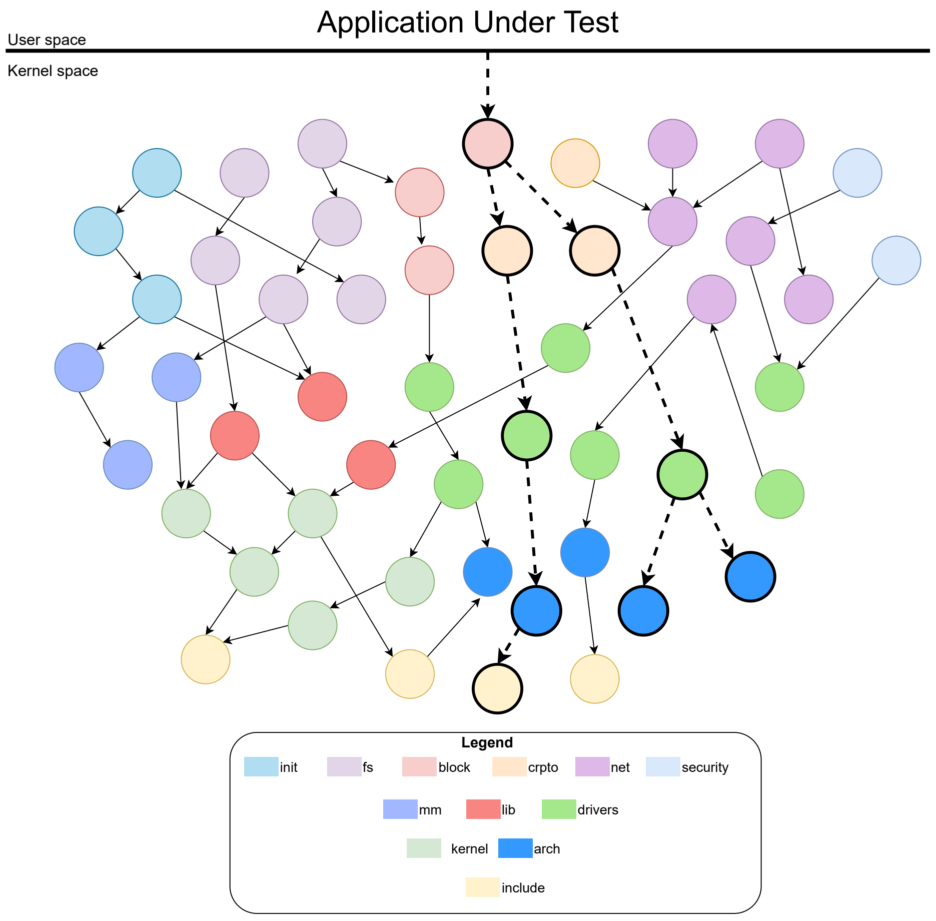


Figure 2.2 Rough illustration of the kernel architecture and the possible execution paths.

Even if the kernel is reduced to what is strictly necessary, it seems unrealistic to achieve full coverage. The application under test only exercises a number of kernel functions and, thus, there are a series of paths that can not be tested with the exercise of this application. After all, there are execution paths that are only possible to be executed if a specific call is performed with specific inputs. For instance, the code of system-call *read()* cannot be exercised if the application does not call *read()*. Furthermore, depending on the input values of the *read()*, the path can also be distinct. Thus, the input parameters of the system-call also determine which paths can be exercised inside the system-call. In other words, there are a number of paths within a system-call that are only executable with a specific combination of input parameters.

2.2.2 Execution path variability

In order to be able to identify the main issues of testing techniques with the state-of-the-art use cases, let us put ourselves in the simplest case having as reference the black-box of Figure 2.3. If we assume that we have an ideal world of pure stateless functions, we can state that there is no impact from the environment in the internal black-box. Consequently, in simple applications running on a single-core, it is potentially feasible to check if this sequential code is correct or incorrect without uncertainty. By exercising all the combinations of its inputs, we will be able to check if the software is correct or incorrect. If we exercise the application with an input vector that causes a failure, this failure will always occur with that input. Therefore, we can describe it using a deterministic model, where functional software failures are also deterministic as it is stipulated that software faults are due to systematic faults. However, most techniques are valid under one condition: the execution path is fully determined by the input-set, meaning that this type of application has no uncertainty.

If we consider a simple system that consists of a single thread application, we can say that the software itself does not present any inherent non-determinism of software execution. Note that the term *"determinism"* is used throughout the thesis in the context of determining the sequence of functions' executions, and not in the real-time temporal sense. Traditional safety-related applications are ideally stateless, which means that there is no impact of execution history and the environment on the internal state of the software.

Figure 2.3 Black-box testing [IEC61508].

But for the next-generation applications, the techniques based on deterministic systems do not produce valid/complete results. These conditions fail with actual GPOS with a large number of possible paths per function and input. Next-generation safety-related systems can include a ML or AI algorithm-based application, which runs above a middleware (e.g., Robot Operating System (ROS)), on a GPOS (e.g., GNU/Linux) and, finally, on COTS multi-core devices with a shared resources architecture and non-deterministic mechanisms (e.g., branch prediction, Pseudo Least Recently Used (PLRU) cache replacement). Consequently, every level of this architecture can introduce non-determinism and with concurrency in place, non-determinism can arise even if the hardware-induced non-determinism is neglected. As it can be seen in Figure 2.4, for the same input, there is not only one possible execution path. Actually, the same input function call can follow different available paths, which, in addition, provide the same result. Therefore, even with a fixed input, it is not possible to predict the internal path that is taken. Besides, the complexity is increased as some of these possible execution paths are unknown, making it infeasible to have test-cases for them. We also have to take into account that there is no control of the system state to force the system to take a particular path. Systems are no longer static. Consequently, observable paths do not involve being all the relevant paths.

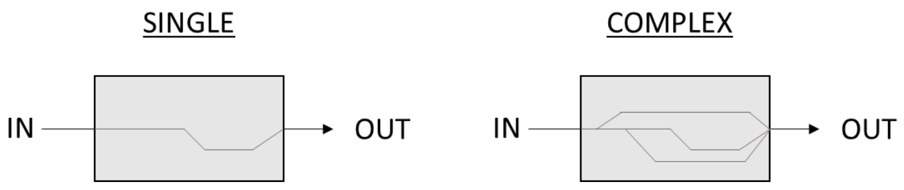


Figure 2.4 Path comparison between single and complex systems.

N. Mc Guire provided in a workshop an idyllic example of a simple source of non-determinism: *"imagine that we have a dual-core executing identical instances of an application that write in the same General Purpose Input/Output (GPIO) and that both instances are perfectly synchronized by the same clock. Which core will write first in the GPIO?"*. If we forget about concurrency, synchronization, or locking mechanisms of the GPIO, the answer is *random*. We cannot know it, as both processors might have a 50% chance. So this is an example of how a strictly deterministic application forms a non-deterministic system when it is executed in a complex hardware.

Weaver et al. identified the existence of sources of non-determinism by examining performance counters [WM08, WTM13]. One of the conclusions provided by the publications is that I/O inputs, pointer values, and time-dependent applications can take unpredictable execution code paths.

SIL2LinuxMP project, led by OSADL, shows the non-determinism of the Linux kernel execution in different publications [MGOS09, OGOO15, OMGF14, OMGFOO13, OMG10]. Mc Guire et al. published the first results related to the inherent non-determinism of the Linux kernel execution while they were searching for jitter sources [MGOS09]. This first study concludes that a complex GPOS such as GNU/Linux amplifies the intrinsic randomness of modern processors. Taking this first analysis as a basis, Okech et al. studied in different articles the execution path non-determinism of the Linux kernel [OGOO15, OMGF14, OMGFOO13, OMG10]. The authors provided the results of their analyses showing the

Linux kernel's execution variability both in timing and in execution path length. For example, an identical system-call duration time can vary from 12 μs to 23253 μs, while the execution path length can vary from 77 to 692 functions [OMGF14]. Therefore, the same call with the same inputs and the same final output can lead to different paths. Their research analyzes the execution of different system calls by dynamic inspection (*Ftrace* tool), identifying a high number of different execution paths for each call. The authors heuristically categorized the most frequently executed paths as common paths and the rest as non-common paths. These studies reveal the differences in the execution paths of a replicated application execution on two cores. The same application executed at the same time in different cores of the same multi-core device may have different kernel execution paths as any of the rare-paths can be executed. In such a way, Okech et al. concluded that the execution paths at the kernel-level are not totally determined by the inputs of the applications [OGOO15].

SIL2LinuxMP project also analyzed the non-determinism of a simple application on a multi-core platform by recording the race conditions [OSA]. The experiment is based on the increment of the same counter from different processes and comparing the difference of the counter value and with the times that the processes are executed. The comparison results in the number of race occurrences. Figure 2.5 shows the race occurrences (vertical axis) while the execution loop length increases (horizontal axis) in an Intel i7 multi-core processor.

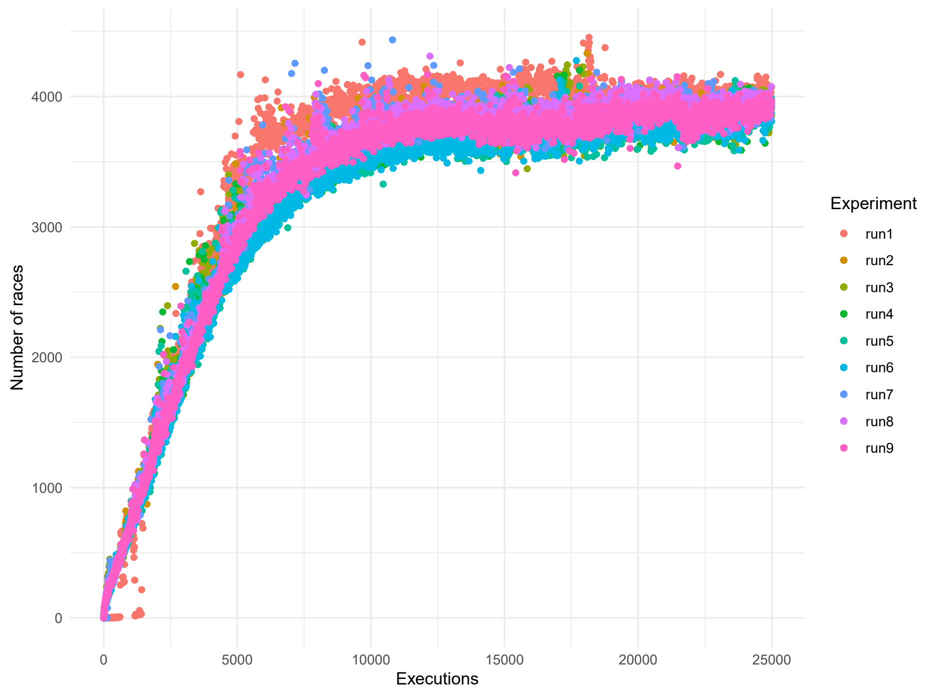


Figure 2.5 Race condition occurrence in an Intel i7 [OSA].

Inherent non-determinism of the Linux kernel means that the same application may not follow the same execution path at kernel-level with the same inputs due to the internal state of the OS and therefore, it makes it much more difficult, if feasible at all, to demonstrate the execution behavior correctness during testing phases. Simplifying, we can imagine the Linux kernel as a composition of several interdependent state machines, which are asynchronous concerning each other, and thus, a given software execution path can exhibit stochastic behavior due to this asynchronicity. The execution path is dependent on an uncontrolled system state and, hence, the system state can be considered an input to the function. It is possible to state that complex systems operate under conditions that are not completely known. As a result, there are two sources that cause the variability in the execution paths:

- **Source 1:** Variability of the input parameters of the functions (e.g., open(libc.so), open(/dev/urandom)).
- **Source 2:** Variability of other environmental inputs or system conditions not directly related to the functions under study (e.g., Read-Copy Update (RCU), spinlocks). Thus, the non-determinism is caused by conditions that depend on shared states.

So an example of *'Source 2'* would be to have a branch in the code that depends on a shared condition.

```
void function(param) {
  ...
  if (shared_condition) {
      conditional_code(param | other)
  }
}
```

Similar examples of the code above can be found in the Linux kernel (v5.12). For instance, the memory subsystem (mm/vmalloc.c) shows an execution path that depends on an asynchronous event. There is an external condition (i.e., *in_interrupt()*) decides if the *vfree()* call releases memory immediately or postpones it (schedules it for later execution). Function *in_interrupt()*) checks if there is a positive interrupt/preemption level count. Therefore, this state depends on unrelated concurrent events. It is not feasible for a test-case to deliberately execute a specific trace.

```
static void __vfree(const void *addr)
{
    if (unlikely(in_interrupt()))
        __vfree_deferred(addr);
    else
        __vunmap(addr, 1);
}
```

Luo et al. performed extensive research on tests that have non-deterministic outcomes, also known as flaky tests [LHEM14]. According to the authors, flaky tests are relatively frequent in large code bases. The majority of the time flakiness is caused by the test itself, but there are also cases where this is caused by the non-determinism of the code under test. Having this research as a reference, Palomba and Zaidman examined the causes of flaky tests, determining that the most common causes are asynchronous wait, Input/Output (IO) operation, and concurrency [PZ17]. These causes can be either from the test code or from the code under test. Geo et al. demonstrated non-determinism of code under test, pointing out how worthless it is to run the test only once [GLC$^+$15]. Therefore, test cases must be run repeatedly. But how many times? How many iterations are necessary to achieve adequate coverage?

2.2.3 Current coverage analysis tools

The available literature shows that test coverage of the Linux kernel has been an interesting topic of research for years [LHRF03, Iye02]. *Gcov* code coverage tool is included in the mainline Linux kernel and can be used at both user and kernel-level [gco]. At the kernel-level, this tool provides coverage results (line coverage, functions, and branches) based on each source code file of the kernel. *Kcov* is another code coverage tester suitable for coverage-guided randomized testing (fuzzing) [kco]. It also provides the results by source code file. Several test suites take into account the kernel coverage while testing the Linux kernel. Linux Test Project (LTP) is commonly used with Gcov test coverage tool LTP [LHRF03, Iye02]. *Syzkaller* is an unsupervised kernel fuzzer that is guided depending on the coverage [syz].

As these code coverage tools provide a result based on the entire available source code, they are suitable for this type of test suites that aim to test as much of the kernel as possible in order to find as many bugs as possible. However, when we want to know the kernel code coverage of a specific application, they do not provide useful results since the results are based on the entire kernel and not all the traces that the application can exercise.

Static source code analysis tools have been used to identify all the execution paths that an application can exercise. These tools examine statically all the possible function sequence combinations of a specific source code. The Linux community also provides tools that provide the call graphs of the kernel. One of these tools is known as *NCC* (no longer maintained) that reports the functions call sequences, and thus, provides execution paths of the Linux kernel [nccb, ncca]. *Call-graph* is another static tool that is intended to provide the function call sequences that an application can exercise in the Linux kernel [ELIb]. Similarly, *Egypt* tool creates call-graphs with gcc compiler [Gus].

Notwithstanding, static source code analysis tools have certain limitations with codes like those of the Linux kernel. Simunovic enumerates the major issues for static tools that examine the call sequences of the Linux kernel [Sim20]: detecting indirect calls, aliases, dead code, and assembly code. Documentation from Egypt tool states that it *"does not display indirect function calls. Doing that is impossible in the general case: determining which functions will call each other indirectly at runtime would require solving the halting problem"*. Linux kernel implements a large number of indirect calls (above fifty thousand) with the purpose of supporting dynamic behavior [LH19]. Indirect calls are function pointers that select the invocation of a specific implementation (from the different existent ones) at runtime. Indirect calls are widely used for interface functions and callback. Besides, they are considered essential for different software. Nevertheless, indirect calls hindrance the construction of precise call-graphs [LH19]. Thereby, these tools cannot only identify certain execution paths (e.g., dependent on indirect calls and aliases) but may also offer paths that are impossible to execute (e.g., dead code).

Note that safety standards, including IEC 61508 Ed 2, states that the use of pointers should be limited. The objective of this measure is to *"avoid the problems caused by accessing data without first checking range and type of the pointer. To support modular testing and verification of software. To limit the consequence of failures"*. Next-generation safety-related systems require higher use of pointers compared to traditionally built safety systems, especially for dynamic memory allocation. However, they need to be used correctly. SIL2LinuxMP project examines the assessment of the suitability of pointer uses [OSA]. The assessment of correct use of pointer type is covered by the warnings provided by the compilers, while the correct address space is achieved with Memory Management Unit (MMU) and static code checkers (e.g., Sparse semantic checker tool). Moreover, defensive structures also need to be considered, checking the pointers before accessing them.

2.3 Conclusions and main gaps

The safety domain has traditionally considered testing as one of the major methods to support the safety system correctness evidence. IEC 61508-4 Ed2 defines dynamic testing as running software and/or hardware in a controlled mode to demonstrate the presence of the required behavior and the absence of unwanted one [IEC61508]. The results obtained from this method have been appropriate for the safety-related applications that we have known until today (i.e., lower-complexity systems). However, with the software's increasing complexity, it seems that full coverage by dynamic testing cannot be the main method anymore.

Consequently, considering the inherent non-determinism that characterizes the described next-generation complex systems, it is no longer feasible to perform a beforehand detailed analysis of execution paths based on the kernel source code (static analysis) and then try to test them all (traditional approach). On the one hand, it is not feasible to perform a detailed analysis of all the execution paths, especially because of path variability due to *'Source 2'*. Although static analysis tools provide a number of different execution paths, only a small size of the provided paths is observable [OSA]. For instance, if we do a static code analysis of the Linux kernel, we get a large number of execution paths. Nonetheless, some of these paths will never be executed due to the architecture that is being used, the application that is running, or the configuration used (e.g., Virtual File System (VFS)). Furthermore, it would be extremely difficult (if feasible), to reproduce some of the scenarios that would provoke some of these predicted/statically analyzed paths coming from *'Source 2'* to be executed. Generally speaking, it is not feasible to predict which path will be executed in a complex system even with a non-variable input.

Despite the fact that there are different test coverage tools alternatives, one of the main issues of these types of tools is the lack of information about the execution probability of the paths. Note that risk is calculated by the multiplication of probability of occurrence of harm and the severity of consequences ($risk = probability * severity$). However, since these tools show the possibility and do not provide a probability, it is not possible to determine the risk. Some of these paths could be considered negligible as their execution probability is so low that they can be considered within the tolerable risk even without testing them.

Our research is focused on the study and definition of a statistical method for the analysis, on one hand, of testable paths and, on the other hand, of the unseen paths. Note that both groups are related to each other, as a higher number of testable paths involves fewer unseen paths. The goal is to estimate statistically the number of kernel execution paths for a given software application by testing and quantifying the uncertainty taking based on the recorded execution traces. In addition, the idea is to be able to estimate the probability of execution of the not recorded paths during the study and, by doing so, be able to estimate the residual risk and the number of failures per hour as it is referenced in the IEC 61508 standard as Probability of Failure per Hour (PFH). The approach of this research group is to define statistical methods taking as reference a set of assumptions based on a preliminary analysis of the execution traces and of the kernel development process.

The use of statistical analysis is not new in safety engineering. IEC 61508 standard provides initial guidelines of *"a probabilistic approach to determining software safety integrity for pre-developed software"* on Part 7 Annex D. IEC 61508-2 Annex F references the IEC 61164 standard titled *"Reliability growth – Statistical test and estimation methods"*, which provides models and numerical methods to determine the reliability growth based on failure data. Out of IEC 61508 standard context, we can find the Probabilistic Safety Assessment (PSA) safety analysis method. PSA is a probabilistic method that estimates the risk of nuclear power plants and allows enhancing their safety [Ald13, Ful99, PNAC92].

Within the context of research projects, it is possible to find different approaches based on statistical studies to deal with different challenges that can also be identified in the next-generation of safety-related systems. Extreme Value Theory (EVT)-based probabilistic analysis is successfully being applied for the probabilistic Worst-Case Execution Time (pWCET) analysis of complex software systems of different criticality deployed in COTS multi-core processors [CKM$^+$19, AAAL$^+$15, CGSH$^+$12, CAA$^+$16], even in Linux based systems [SAdOdO18]. Bayesian methods are also examined and proposed to quantify uncertainty in Deep Learning for AI safety applications [Gal16, KBC15, KC16, FNT$^+$20].

Chapter 3

Overview of the Proposed Technique

Aiming to contribute to the main gaps identified in the literature, several research activities have been conducted. The performed analyses result with the definition of a technique that aims to statistically quantify the test coverage of the Linux kernel in the context of next-generation autonomous systems. The proposed technique is based on different methods that are studied and described in the present document.

In this chapter, we present a brief description of the technique with which we intend to contribute. For this purpose, we analyze three novel methods, which aim to pave the way towards the test coverage of next-generation safety-related autonomous systems, specifically based on the Linux kernel.

First, an overview of the proposed technique is reported. Then, we describe two different approaches to assess test coverage based on execution paths with a relevant probability of being executed. Both methods share the same objective but employ different approaches to achieve it. Their results are reported in Chapters 5 and 6. The estimation of the risk entailed to the untested execution paths is described later with the results reported in Chapter 7.

3.1 Overview of the technique and methods

The challenges in the context of test coverage of next-generation safety-related systems are reported in Chapters 1 and 2. Several characteristics that define these systems hinder the assessment of the testing phase (e.g., inherent non-determinism). The state of the art review identifies the need for alternative measures and techniques to assess and quantify the testing coverage. Therefore, we analyze different approaches in order to quantify the coverage, specifically focusing on the Linux kernel. Moreover, as the literature states, the achievement of full coverage is extremely complex (if even feasible). Consequently, we propose an additional method to estimate the risk associated with the untested code. Due to the limitations identified in Chapter 2, there is a need for research into different methods that can perform assurance activities. As IEC 61508 functional safety standard points that assurance activities are based on analysis [IEC61508], we conduct several analyses to evaluate the safety assurance of autonomous systems with functional safety requirements.

Chapter 2 identifies the limitations of static analysis tools for next-generation safety-related systems. Thereby, dynamic analysis tools are considered a possible substitute for static analysis tools [Sim20]. Dynamic analysis is performed during system-running and, hence, allows examining the execution of the system. Thereby, an interesting option is to analyze the execution traces of the system by recording the traces during execution. This allows examining the execution traces exercised by a task. As a result, it also allows identifying and examining the path variability of the Linux kernel by executing test-case(s) repeatedly. Besides, the dynamic analysis does not share the limitations of static tools (e.g., indirect calls). In the case of the Linux kernel, dynamic analysis tools record the execution traces that an application exercises at the kernel-level on behalf of a user-space task. This task can be reliably performed with FTrace, a tracing tool that allows recording the function call sequences. However, as the main limitations due to non-determinism remain, certain questions need to be answered even for dynamic tracing.

- If we identify the existent traces while the kernel is exercised repeatedly by an application or test-case, how do we know the total number of traces that have a relevant probability of appearing?
- Thus, how do we quantify the test coverage ?
- How do we know the risk related to the untested (unknown) paths?

In this thesis, we try to answer all these questions by presenting and analyzing complementary approaches with the aim of paving the way towards acceptable safety assurance, specifically targeting the test coverage of Linux-based safety-related systems. Although dynamic analysis seems to be a suitable alternative, recording execution data alone does not provide much information. The obtained data needs further analysis in order to be able to make certain estimations. So, how do we predict what remains to be executed having as reference previous executions? This is where statistics come into play. Using the data we already know, those seen in previous runs, and various statistical techniques, we can make certain estimations. Thus, taking into account the characteristics and the limitations of these systems, several statistical techniques are selected in order to perform the necessary analyses.

The contributions made in this thesis are based on statistical approaches that examine data obtained by the dynamic analysis of the system under test. The main objectives of the proposed methods and, therefore, of the research activities of this thesis are:

- **Data collection:** Record the execution traces of the Linux kernel while the system is under test. Therefore, a number of test cases are executed in a defined system-context while the execution kernel traces that are exercised are recorded. The collection needs to manage and identify the distinct traces that are executed.
- **Test coverage:** Analyze different statistical approaches that estimate the test coverage. Estimate the number of paths that have a relevant probability of being executed by examining the traces that have been executed during the testing process.
- **Risk quantification:** Examine a statistical method that quantifies the risk entailed to the untested code as full coverage is hardly achievable.

In this thesis, we have performed different research activities in order to contribute with a technique that meets these three objectives: Statistical Path Coverage (SPC). The technique is for systems that exhibit stochastic behavior of function call sequences or, in other words, for systems whose execution path depends on external (global) state that is asynchronously modified by unrelated processes. The technique examines actual (observed) execution path and evaluates the path distribution (recurrence) properties. Figure 3.1 describes briefly the SPC technique that we aim to contribute with. The idea is to collect the execution traces dynamically and, then, use this data to statistically estimate the test coverage and risk entailed to the untested paths. Therefore, we can identify the three phases that form the SPC technique.

With these objectives in mind and with the intention of addressing the identified main gaps, this thesis analyzes three methods: two for test coverage and one for residual risk estimation. Furthermore, we also describe the data collection and validation process.

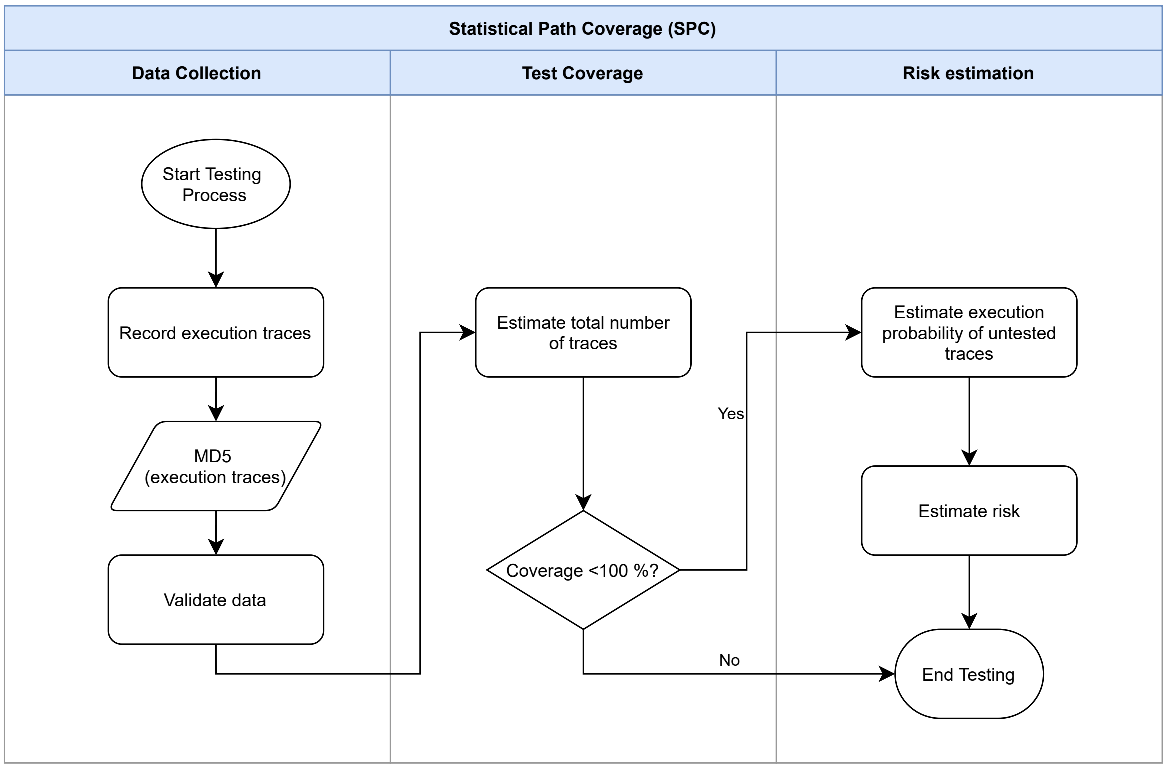


Figure 3.1 Statistical Path Coverage (SPC) technique overview.

1. **Data collection based on dynamic analysis:** We use the DB4SIL2 tool developed in the SIL2LinuxMP project to perform the acquisition of the data set. Besides, we propose a data validation process in order to know if the obtained data set is large enough and fulfills the quality requirements.

2. **Test coverage modeling the tested data:** Modeling the recorded data set to describe the behavior of the system and, thus, estimate the total number of traces (see Chapter 5). It shares the objective with the following approach, but in this case, the method is based on parametric statistics.

3. **Test coverage with non-parametric estimators:** Use non-parametric species richness estimators with the objective of estimating the test coverage (see Chapter 6). We inspire in a biological issue or challenge to contribute with a method that examines the traces that are executed during the testing process in order to calculate the number of traces that have a relevant probability of being executed.

4. **Residual risk quantification by estimating execution probabilities:** Estimate the execution probability of untested execution paths and, hence, be able to quantify the risk that entails (see Chapter 7).

As it can be observed, the contributions are in line with the phases defined in Figure 3.1. Firstly, data collection and validation based on the tool provided by the SIL2LinuxMP project. Secondly, two complementary methods for test coverage estimation based on the collected data. Finally, a last method for software risk estimation.

Some of the contributions of this thesis have the same objective but are performed with different statistical methods. This is the case with the test coverage methods. Having two independent methods allows comparing the obtained results and, hence, having a certain level of assurance that the methods are adequate if the results are consistent. Moreover, this thesis also offers a method to analyze the risk that entails the execution of untested traces. Test coverage methods show that it is extremely difficult to obtain 100% coverage. Therefore, the proposed method aims to estimate the risk associated with the non-covered execution traces.

3.2 Data acquisition based on dynamic analysis

Static analysis tools seem no longer adequate for identifying the execution paths (see Chapter 2). Consequently, an alternative approach is to base the study on dynamic analysis. Test coverage measures the ratio of tested code. Thereby, if we want to research and advance in this field, we need to identify the code that is executed by the application under test. As in this particular case, the research is focused on the Linux kernel, it is essential to know the traces that the application exercises at the kernel-level.

As aforementioned, the study is fully focused on the execution path (or trace) coverage, which IEC 61508 standard refers to as *"structural test coverage (entry points + statements)"*. Therefore, this means that there is no need to control the inline functions or the branches that occur during execution. To carry out the research, we need to obtain a data set consisting of the execution paths exercised by the application during testing. In this section, we define how the acquisition of the execution paths is performed and its later post-processing to have an adequate data set to perform the study correctly.

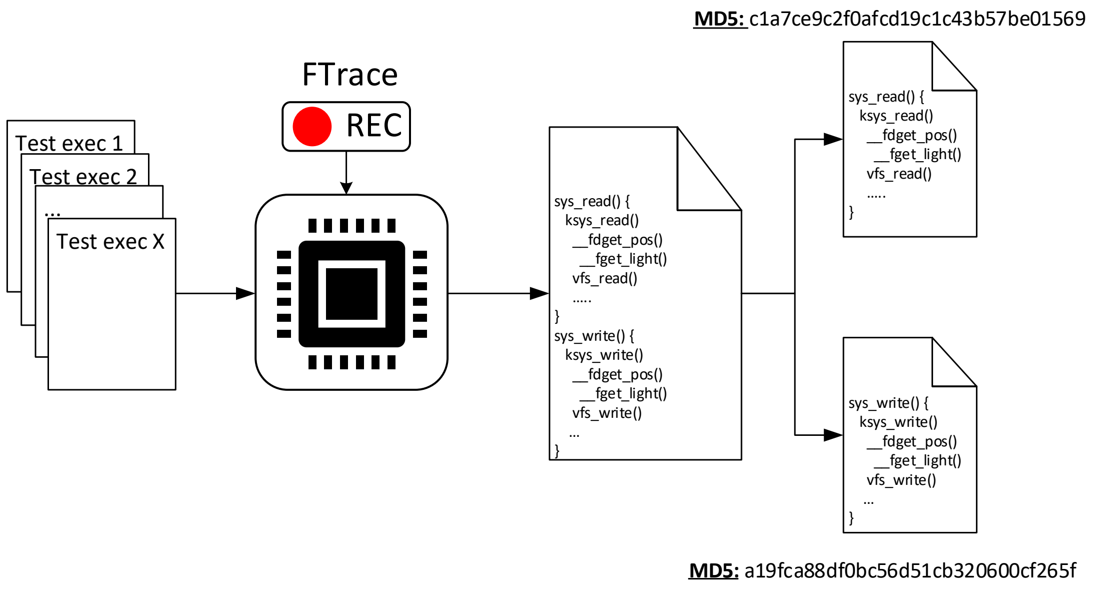
Prior to performing the acquisition, we need to define the context in which the system is exercised. The idea is to record the execution traces of the application by exercising the target-system in its system-context. Thus, we need to define these three parameters before starting testing the system.

- **Target system:** refers to the platform that is used for the system under test. This involves both the hardware and the software that is used. For example, ZynqMP Ultrascale++ consisting of a quad-core ARM Cortex-A53 running the v4.19.75-cip11 Civil Infrastructure Platform (CIP) Super Long Term Support (SLTS) Linux version.

- **Application:** is the safety application that is tested.
- **System-context:** refers to the context in which the system will operate and, therefore, the cases to be tested. Therefore, the system-context is defined as a series of executions of the test-case of interest in the target system along with a possible overload of a large number of subsystems.

3.2.1 Recording and post-processing

For the collection of kernel-level execution traces for a given Linux-based application software, we employ DataBase for Safety Integrity Level 2 (DB4SIL2) tool [Pla16]. DB4SIL2 is an open-source tool, mainly developed in Python, that supports the collection of kernel execution traces and post-processing of them. The tool was developed for the SIL2LinuxMP project at OSADL [OSA] and some minor modifications have been done for this analysis. The kernel trace collection is done using FTrace, a Linux kernel built-in tool that allows tracing the execution path and key state information of the kernel [ftr17]. It is highly configurable and supports tracing kernel functions, interrupts, CPU changes, low-level locking, etc. FTrace is a dynamic analysis tool that provides the function call sequence (i.e., execution path or trace) executed at the kernel-level.


MD5: c1a7ce9c2f0afcd19c1c43b57be01569
FTrace
REC
Test exec 1
Test exec 2
...
Test exec X
MD5: a19fca88df0bc56d51cb320600cf265f

```
sys_read() {
  ksys_read()
    __fdget_pos()
      __fget_light()
    vfs_read()
    .....
}
sys_write() {
  ksys_write()
    __fdget_pos()
      __fget_light()
    vfs_write()
    ...
}
```

```
sys_read() {
  ksys_read()
    __fdget_pos()
      __fget_light()
    vfs_read()
    .....
}
```

```
sys_write() {
  ksys_write()
    __fdget_pos()
      __fget_light()
    vfs_write()
    ...
}
```

Figure 3.2 DB4SIL2 overview diagram.

Figure 3.2 shows an overview diagram of the DB4SIL2 tool. The tool is formed by two main parts. On the one hand, it has a client that is responsible for collecting the traces of the platform that is being analyzed and, on the other hand, the server-side is in charge of post-processing the collected data. DB4SIL2 client forks the main process and sets the child Process Identifier (PID) in the FTrace setup. Then, it starts FTrace and executes the application that is going to be analyzed. Once the execution of the application has finished, it collects the obtained trace and sends it to the server.

Finally, the DB4SIL2 server-side post-processes the received trace. Post-processing consists of identifying the different system calls present in the collected trace. System-calls are considered the kernel entrance function, as the application requests the utilization of the kernel (or the resource it provides) through different system-calls. Thus, the recorded execution is mainly formed by the sequential execution of distinct system-calls. Furthermore, the identification and classification of the execution traces are performed through a hash function (i.e., MD5). MD5 message-digest algorithm is a traditional hash that can be used to determine uniqueness. Therefore, each system-call trace results in a hash value. Designating a hash value to each specific execution trace sequence facilitates data analysis. If a system-call path is repeated, it will result in the same MD5 value and, therefore, it simplifies the computing processing required to obtain the number of different system-call paths and frequency of occurrence of each of them.

The system-call traces are not the complete records of exercising the kernel per se. The whole execution trace is formed by a series of system-calls plus a few more functions. Therefore, just analyzing system-calls cannot be considered a completely proper verification of the kernel but rather a contribution to the verification of the kernel interface. Although a large set of kernel functions (notably the generic functions like locking primitives, memory management, or reporting functions) are recorded quite extensively, one cannot conclude the unobserved paths in the kernel itself. However, if the method shows effectiveness for the kernel interface, it could be expanded to the overall kernel.

3.2.2 Data set formation

The data set that we obtain is composed of a large number of execution traces. The traces are organized in different test-campaigns, as each test-campaign is formed of repeated execution of a test or a series of tests. As a result, we get a series of hashes for each test-campaign that represent the execution of different system-calls. Two concepts that need to be clear:

- **Unique trace:** a specific sequence of functions that corresponds to one or more executions. In other words, if several executions result in the same execution trace, we consider that specific function call sequence a *unique trace*. Paths with the same MD5 refer to the same *unique trace*, as having the same hash value means that the same function call sequence has been exercised. On the contrary, if the execution follows a different path, it results in a different hash value and, thus, in an additional *unique trace*.

- **Test-campaign:** grouping of repeated runs of a specific test-case or several test-cases.

Consequently, for each test-campaign we obtain the following information: (i) the system-calls that have been executed (e.g., sys_open, sys_read), (ii) the different *unique traces* that each of these system-calls has followed, and (iii) the frequency of execution of these traces. We include below an example of a trace and the obtained MD5 hash value. Note that in this case, the system-call has a prefix of the architecture (i.e., arm64), but not all architectures have this specification or use a prefix.

MD5: c6f287e6c6a3e6228a8ee046ebacca0d

```
__arm64_sys_write()
  ksys_write ()
    __fdget_pos ()
       __fget_light ()
    vfs_write ()
      rw_verify_area ()
         security_file_permission ()
      __vfs_write ()
        write_null ()
      __fsnotify_parent ()
      fsnotify ()
```

The execution trace corresponds to system-call *write()*. Briefly, the execution trace of *sys_write()* calls *vfs_write()*. Virtual File System (VFS) is an abstract-switching layer that assigns the correct device or type of file system (e.g., ext4, ext3, msdos) to the incoming call. This trace is accessing */dev/null* as the call to *write_null()* shows. During the trace, we can identify other functions that check access, security permissions, and notifications.

3.2.3 Understanding inherent non-determinism on the Linux kernel

Before introducing the methods that we propose in this thesis, it is necessary to have a proper understanding of the non-determinism of the systems that are being developed today, specifically focused on the Linux kernel. Chapter 2 provides a theoretical explanation of the inherent non-determinism. Instead, in this section, we offer a practical example based on the example of *__arm64_sys_write()* discussed above and described in Annex A.

The state of the art review showed that the cause of the non-determinism are external dependencies. Multi-core processors execute program instructions concurrently on several cores. As they follow a shared resource architecture, there are several serialization control mechanisms to avoid concurrent accesses to resources. These measures cause non-determinism as the execution paths vary depending on the state of shared resources. However, the non-determinism does not only lie in the software but in the hardware as well. There are different causes for non-determinism [HFP$^+$20, OSA]:

- Unknown system state.
- Dynamic resources.
- Physical concurrences.
- Cache line replacements.
- Non-deterministic optimizations.
- Out of order execution.
- Dynamic wait states.
- Speculative executions.

Therefore, the Linux kernel has a number of constructs that are non-deterministic by design. For instance, all the sleeping locks such as mutex, rt_mutex, semaphores are non-deterministic. Additionally, external asynchronous events (e.g., interrupts) occur with the integration and use of interconnected systems. An increasing number of autonomous systems use AI accelerators that are asynchronous to the CPUs and non-cache coherent. For instance, the Linux kernel widely uses the RCU synchronization mechanism to improve concurrency. RCU allows reading data structures concurrently, even while these data structures are being updated by other threads [mck20]. The majority of these serialization and synchronization mechanisms have the potential to result in different execution paths and, hence, provoke the execution of a branch. In other words, the execution can deviate from the performance-optimal trace (without any serialization mechanism) to execute additional functions related to serialization. One branch that has been widely identified during the research activities conducted in this thesis is *rcu_note_context_switch()*.

```
 rcu_note_context_switch ()
   rcu_sched_qs()
   rcu_preempt_qs()
_raw_spin_lock()
 pick_next_task_fair ()
    put_prev_task_fair ()
       update_curr ()
       __enqueue_entity ()
       __update_load_avg_se()
       __update_load_avg_cfs_rq()
wakeup_preempt_entity. isra .8()
 clear_buddies ()
 set_next_entity ()
       __update_load_avg_se()
       __update_load_avg_cfs_rq()
 finish_task_switch ()
       _raw_spin_unlock_irq()
```

Since Linux is developed from a performance-optimization perspective, the most common execution path usually does not have the execution of these serialization mechanisms. This is observed in the *__arm64_sys_write()* example. Approximately 90% of the time, the executed trace is the example we have presented above. However, the rest of the times the same trace has been executed but with a series of branches at different points of its execution. For instance, if the system-call *write()* is exercised repeatedly, *rcu_note_context_switch()* branch can be identified at different points of the execution path.

```
__arm64_sys_write()
                <- rcu branch
   ksys_write ()
                <- rcu branch
```

```
  __fdget_pos()
           <- rcu branch
     __fget_light ()
           <- rcu branch
  vfs_write ()
           <- rcu branch
    rw_verify_area ()
       security_file_permission ()
           <- rcu branch
    __vfs_write ()
           <- rcu branch
       write_null ()
           <- rcu branch
    __fsnotify_parent ()
           <- rcu branch
    fsnotify ()
           <- rcu branch
```

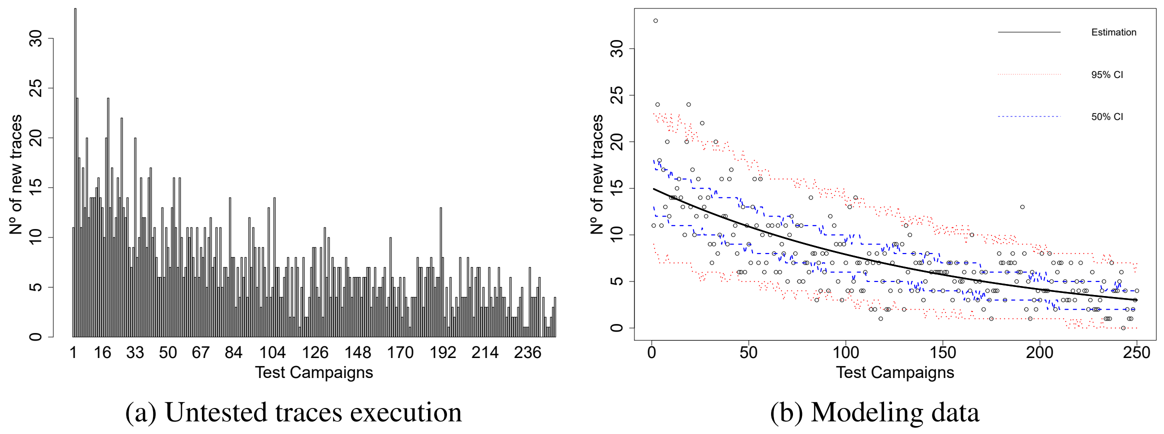
Just the RCU branch has been identified almost after every function of the common execution trace. However, this is not the only branch that can be executed. Throughout this thesis different branches have been identified in the distinct system-calls (e.g., *__do_page_fault, raw_spin_lock, __slab_alloc, dput, finish_task_switch*). Besides, SIL2LinuxMP project also identified a number of asynchronous events during the DB4SIL2 tool development. Some of all these asynchronous events are collect in Table 3.1. The table categorizes the events with the same subsystems presented in Figure 2.2. In addition, the kernel subsystem shows additional categorization (e.g., sched, time, power). An interesting fact that emerges from this categorization is that asynchronous events do not come from a specific subsystem, but are distributed throughout the entire kernel architecture. Note that Table 3.1 is a preliminary list of the research conducted with the branching model. To be accurate, it needs further investigation. Therefore, we caution the reader that this is not a complete/accurate list of asynchronous Linux kernel events. It is essential to take into account that in a preemptive OS, scheduling can occur at almost any point of the execution. This is depicted in the example above, where the execution of system-call *write()* shows interruptions almost in all function calls. Note that this could be ideally avoided by disabling preemption. However, the number of use cases where a non-preemptive OS is effective or useful is extremely limited.

Additionally, a branch can be branched too. In other words, while the trace is executing a deviation of the most common execution path (e.g., rcu), the deviation may be deviated too. Thus, it results in a different execution path with a branched branch and, hence, in an additional *unique trace*. With all of the above, it makes sense that each trace has a different probability of execution. Depending on the branch, it will be more likely than another and, therefore, generates traces that are executed more frequently.

3.3 Test coverage estimation with a parametric approach

The aim of this method is to quantify the coverage while testing complex safety-related software based on (a subset of) GNU/Linux. Even though it shares the objective with the following method, the present method proposes a parametric approach to perform it. Parametric statistical methods describe the underlying population of a data set with a model, which has a fixed set of parameters. Therefore, the study selects an appropriate statistical model to describe the execution of untested traces. The idea is to obtain a model that describes the behavior of the system and, thus, be able to predict how the system would behave if the system would be tested during an infinite time.

The goal of this research is to move from full path coverage towards a statistical approach and, thus, be able to quantify the assurance along with the uncertainty. The idea behind the method is to comprehend the process of appearance of new traces during testing and find a suitable model to describe the behavior. Fitting the model provides the equation that describes the appearance of previously untested traces and, hence, we can estimate the execution process of the ones that have not been observed yet. Consequently, we can estimate the uncertainty of the system.



(a) Untested traces execution (b) Modeling data

Figure 3.3 Analysis process of the proposed parametric method.

Table 3.1 Asynchronous events identified during the thesis and by the SIL2LinuxMP team.

fs	kernfs_notify_workfn pstore_dowork __d_lookup	dget_parent dput
block	blk_done_softirq	bio_dirty_fn
net	net_rx_actionnet_tx_action cleanup_net iucv_work_fn	p9_poll_workfn switchdev_deferred_process_work vmci_transport_cleanup
mm	kmemleak_do_cleanup pcpu_balance_workfn	__slab_alloc
lib	irq_poll_softirq	free_obj_work
drivers	acpi_device_del_work_fn sb_notify_work deferred_probe_work_func crng_reseed fcopy_send_data vss_send_op perform_shutdown kvp_send_key kgdboc_restore_input_helper moom_callback sysrq_showregs_othercpus	iio_dma_buffer_cleanup_worker do_deferred_remove ser_release aer_recover_work_func maple_dma_handler maple_vblank_handler console_callback tty_schedule_flip con_driver_unregister_callback event_handler evm_deferred_drvvbu

arch	__do_page_fault
	ia64_mca_cmc_vector_disable_keventd()
	ia64_mca_cmc_vector_enable_keventd()
		mce_do_trigger
		process_sci_queue_work
		pvclock_gtod_update_fn

include	__raw_spin_lock __raw_spin_trylock	__raw_spin_lock_irqsave
kernel	close_work poweroff_work_func tasklet_action	deferred_cad reboot_work_func tasklet_hi_action
	time: run_timer_softirq clocksource_watchdog_work	clock_was_set_work
	rcu: rcu_note_context_switch rcu_process_callbacks	__rcu_read_unlock rcu_read_unlock_special
	sched: update_curr preempt_schedule_irq pick_next_task_fair	__clear_sched_clock_stable run_rebalance_domains finish_task_switch
	power: do_poweroff try_to_suspend	__wakelocks_gc
	cgroup: cpuset_hotplug_workfn	

- **The appearance of untested traces:** Analyze the appearance of traces that have not been executed previously while the system is being tested. As the system testing effort increases, untested paths emerge. The aim is to understand the *"mechanism"* that results in the execution of previously untested paths.

- **Review statistical models:** Examine which statistical model fits and is reasonable to model the appearance of untested paths. For this purpose, it is necessary to understand adequately how these traces are executed. Behind the model that is used, adequate reasoning of the model is necessary.

- **Modeling Data:** Perform regression analysis with the obtained data set. Check if the model fits correctly and fulfills the requirements of the model.

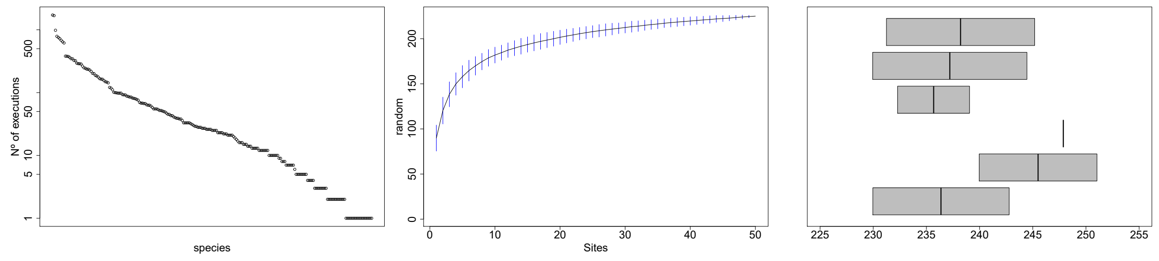
- **Examine the results:** Estimate the coverage with the obtained results.

3.4 Test coverage estimation with non-parametric estimators

Non-parametric species richness estimators are widely used in biology and ecology to estimate the total number of species having as reference previously observed samples. The literature collects different estimators based on distinct principles. Some of the estimators are based on abundance data (i.e., species appeared once, twice, etc.) and others are based on incidence data (i.e., species appeared in one site, in two sites, etc.). Nonetheless, all estimators focus on the existence of *rare-species* in order to perform the estimate of the total number of traces. In other words, the estimate of the total number of species is obtained mainly from the information provided by species that are rarely observed.

Although execution paths are not species, we believe that non-parametric species richness estimators may be used to estimate the total number of traces that an application can exercise at the kernel-level and, hence, quantify the test coverage. The data collection process identifies each *unique trace* with a hash value. Some of them appear several times, and others, are executed rarely. Consequently, if we consider each hash value a specie, we can evaluate the use of non-parametric estimators to estimate the total number of execution traces.

Figure 3.4 shows a brief description of the employed method. The plotted data belongs to a data set of Barro Colorado Island Tree Counts [CPL$^+$02], which is widely used in the evaluation of species richness estimators. The method is formed from the following main points:



(a) Frequency of species (b) Accumulation curve (c) Results comparison

Figure 3.4 Analysis process of the proposed non-parametric method.

- **Examine the existence of rare-traces:** Non-parametric species richness estimators are based on the existence of events that occur rarely (e.g., once or twice). Therefore, in order to perform the analysis, we need to examine the composition of the data set and see if there are traces with low execution frequencies.
- **Accumulation curve:** The principle between all the estimators is an extrapolation of the accumulation curve of the species richness. The assumption is that species richness is asymptotic if we would have an infinite number of samples. Therefore, we need to assure the asymptotic trend of the data, even though it is yet not achieved.
- **Identification and implementation of different estimators:** The literature collects different non-parametric species richness estimators. In this thesis, we select the most widely used ones and evaluate the results obtained with each of them.
- **Compare the results:** Check if the results have certain concordance between them in order to have certain reliability of them.
- **Examine the results with different data sets:** Evaluate the estimators with different size data sets.

3.5 Estimation of residual risk associated with untested code

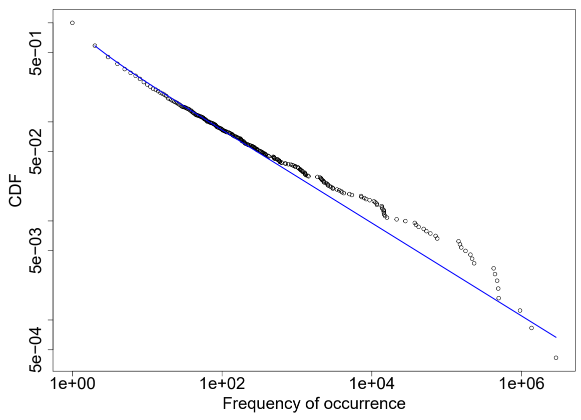
Classical safety-related applications are ideally based on deterministic systems, and they should follow the same execution path (function call sequence) when they are exercised with the same input vector. As a result, it is feasible to test functional correctness by testing the code with all possible input values. Besides, IEC 61508 safety standard states that appropriate justification needs to be provided when 100% test coverage is not achievable [IEC61508].

Safety standards list several testing techniques and measures that are valid for deterministic systems [IEC61508]. However, these techniques fail, at least partially, with non-deterministic systems since several execution paths may exist for an identical input vector. As a result, it is not possible to ensure the correctness of the software code by a unique run of the test-case for each vector.

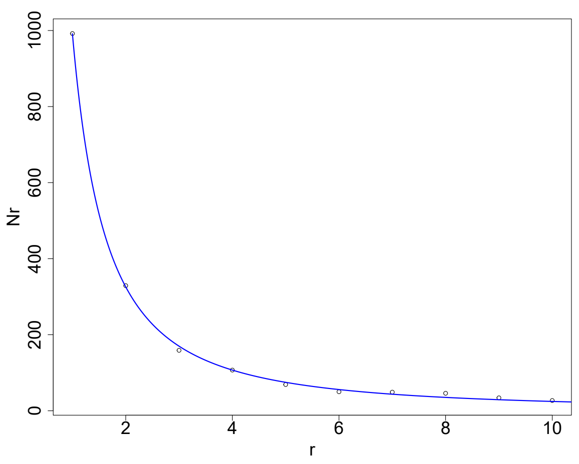
The non-determinism of complex systems makes it extremely difficult to ensure that all possible execution paths have been tested. Qualification by only testing is hardly an option, as the overall complexity of the environments is too high to allow any relevant completeness to be achieved with a defined level of assurance. In other words, it is possible to run the algorithm for 10^x hours but, in the end, we simply do not know how to translate this into a failure rate. Consequently, there is an uncertainty in the execution of these systems that is difficult to quantify with traditional measures, if feasible. Furthermore, this uncertainty poses a risk to functional safety as the execution of an untested path may have catastrophic consequences.

Due to this uncertainty, there is a need to estimate the residual risk that involves deploying these next-generation autonomous safety-related systems. Risk is calculated by multiplying the occurrence probability of an event by its severity. Thus, in order to know the risk, it is required to estimate these two factors. Therefore, this kernel execution path uncertainty, caused by untested paths, jeopardizes the testing phase of the safety-critical application(s). We propose a method for the estimation of execution path uncertainty that provides a probability value for the untested (unseen) execution paths that can be used with the ALARP principle to provide the required code test coverage justification or identify the need to increase the testing depth and scope. For this purpose, we analyze the execution frequencies of the traces, examine a model that defines these frequencies (Figure 3.5), and, finally, estimate the execution probabilities of the untested traces.

- **Examine execution frequencies:** Check the differences in the execution frequency of the distinct execution traces. Examine the proportion of traces depending on their execution rates.
- **Model data:** Model the data with an adequate model to perform further study (Figure 3.5).
- **Simple Good-Turing equations:** Employ Good-Turing equations to estimate the execution probability of untested/unknown traces.
- **Estimate risk:** Calculate the risk that entails the execution of untested traces.



(a) CDF frequency of occurrence



(b) Modeling data

Figure 3.5 Analysis process of the proposed probability estimation method.

3.6 Summary

The present chapter describes briefly the proposed SPC technique and the methods that constitute it. The intent of the methods is to pave the way in the field of test coverage of complex systems, specifically focusing on Linux-based safety-related systems. Three methods are briefly presented to cover this topic and, hence, address some of the gaps identified in the literature.

We believe it is important to emphasize that this technique or similar could not be found in the literature, or at least as far as we are concerned. Firstly, the literature mostly focuses on deterministic systems, leaving an important gap for this type of systems. Afterward, we have not found any approach that calculates the test coverage by taking as 100% the paths that have a relevant probability of being executed. Traditionally, test coverage has been focused on all possible paths and not on not on paths with a relevant probability of being executed. Finally, offering a technique that takes into account the risk associated with untested software constitutes a novel contribution. Moreover, contributing with an approach based on dynamically collected data and the use of statistics is innovative in the field of test coverage.

Knowing that the testing process is one of the most important gaps in the literature related to the safety assurance of the next-generation autonomous systems, the proposed methods were designed to estimate the uncertainty of the systems and the risk that entails this uncertainty. Both test coverage methods pursue the quantification of the uncertainty. Moreover, the risk estimation method provides a justification necessary for a system without full coverage. Therefore, we have three complementary methods, along with the data collection process, that are used to define the SPC technique. The methods are described in more detail and analyzed in Chapters 5, 6, and 7. In addition, the methods use the data obtained in the AEB case study described in Chapter 4 as a guiding example. Note that the methods are not dependent on the use case. In fact, the proposed methods are also examined with an additional use case in Annex A. Finally, taking into account the obtained results, we define in detail how the methods complement to form the SPC technique.

Chapter 4

Case Study - Dynamic Data Collection and Validation

Advanced Driver Assistance Systems (ADASs) are paving the way towards the deployment of self-driving vehicles on roads. Vehicles with a certain degree of autonomy can already be found on the market and prototypes with a higher automation level are being tested in public roads [WH16]. Reduced visibility, driver distraction, or a pedestrian crossing without attention are some of the main risk factors in traffic accidents [Org]. A large number of these accidents are attributable to car drivers who have braked late or not hard enough. As a result, the automotive industry has introduced the Autonomous Emergency Braking (AEB) system intending to prevent this type of accident by braking the vehicle in case of emergency regardless of the driver's actions. This system reduces the probability of accidents, as it allows the driver to be warned in advance of a critical situation and can also brake independently if necessary. Besides, even if the critical situation is detected too late to brake the vehicle completely, the autonomous speed reduction achieves a significant decrease in the accident's severity.

4.1 AEB Systems

The possibility of detecting pedestrians by means of camera image processing has significantly improved AEB systems. Advances in neural networks and in the dedicated accelerators for performing inference of the trained networks allow detecting pedestrians and obstacles with ever-increasing accuracy. The obstacle detection algorithms that were developed over the last few years have the capability to detect the trajectory of a pedestrian. This provides a higher precision in critical scenarios identification, since detecting a pedestrian in the proximity does not mean that the brakes have to be activated. For example, in a city, we can find pedestrians on the sidewalk near the road without any intention of crossing.

Driving Automation Levels (DALs) classify the level of driving automation, from level 0 (no automation) to 5 (fully autonomous), depending on the Dynamic Driving Tasks (DDTs) that the car is capable of performing [CSL^{+}19]. The pedestrian detection AEB system allows achieving a level of 1 or 2 as the vehicle can brake autonomously if a collision is imminent. The AEB system has three main objectives:

1. It shall send the braking vehicle command, within the Safe Response Time (SRT), when there is a pedestrian vulnerable to a collision with the vehicle.
2. It shall warn the driver if there is a pedestrian on the road or intending to enter the road.
3. It shall not send the command of braking when it is not necessary.

Sahin et al. summarized the overall functional architecture of existent Fully Automated Driving systems [TKZS16]. The authors categorized the modules that form the system in four groups: Sensors, Perception Understanding, Motion Planning and Vehicle Control & Actuation. Taking this publication as a reference, we propose a basic block diagram for the AEB system in Figure 4.1. The block diagram is summarized as follows:

- The **Sensors** are used to obtain information about the vehicle's environment. Pedestrian detection AEB systems may use different types of sensors. Even though Figure 4.1 only depicts Cameras, these systems may also employ Radars or Laser Imaging Detection and Ranging (LIDAR) sensors.
- The **Perception Understanding** module is in charge of analyzing data for *pedestrian* and *obstacle* detection. It is based on object detection ML algorithms and runs on any type of neural network accelerator, such as GPUs.

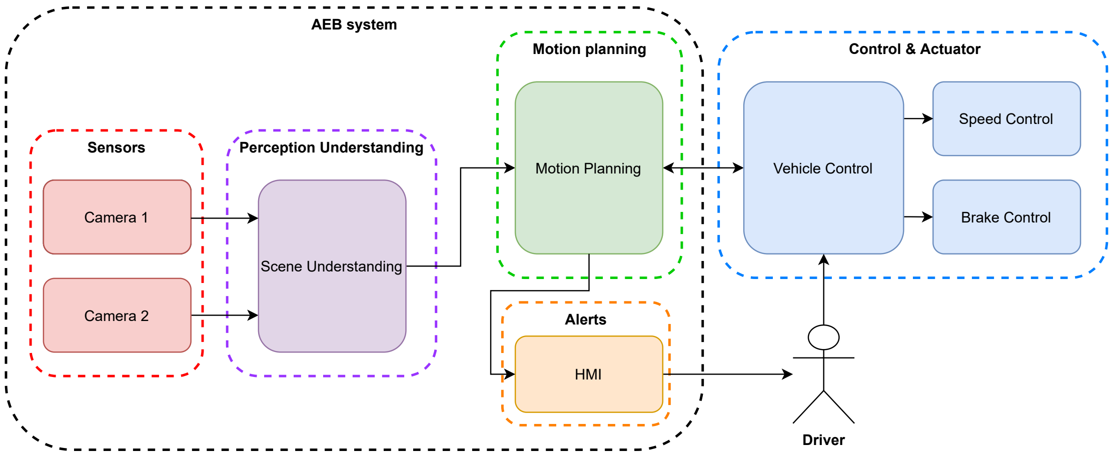


Figure 4.1 Basic block diagrams that constitute the AEB system.

- The **Motion Planning** module is considered the control unit and is responsible for deciding whether the vehicle is in a critical situation and if the emergency braking has to be commanded.

  Depending on the values obtained from the Perception Understanding, it classifies the detection between critical and non-critical events and sends appropriate commands accordingly (ego vehicle brake, warn the driver).

- The **Vehicle control** module controls the speed of the vehicle and activates the brakes if the Motion Planning commands it.

- The **Alerts** warn the driver of a pedestrian detection to act before the situation becomes critical. It also alerts the driver while the AEB system is acting in a critical scenario.

4.2 AEB System Case Study

For the research activities presented throughout this thesis, we have employed an AEB system case study. This case study has features of the next-generation autonomous systems, with ML algorithms for obstacle detection and the Linux kernel as the OS for the control unit (i.e., Motion Planning software partition). Obstacle and pedestrian detection algorithms are executed in a dedicated GPU, while the Linux kernel runs in a COTS multi-core device.

The study presented in this thesis is mainly focused on the Motion Planning software partition, especially in the testing of the safety function, as it is the partition that runs the Linux kernel. With the purpose of fully focusing on the research activities conducted on this thesis, a simplified AEB system is employed instead of the complete version. Consequently, this research-grade AEB system case study allows us to focus on the assessment of the proposed methods, in order to be able to extend the study appropriately in the future to a commercial use case.

As stated in Chapter 3, in order to perform the data collection and further analyses, it it necessary to define three parameters:

- **Target system:** The case study is implemented on an NVidia Jetson Nano formed by a quad-core ARM Cortex-A57 processor and a Maxwell GPU architecture with 128 cores. The Motion Planning software partition is located on the multi-core processor and runs a Linux kernel (v4.9) with the safety-architecture presented in SIL2LinuxMP project (see Section 4.2.1) [OSA]. This architecture is based on Linux containers that allow isolating tasks of different criticality (i.e., different SIL) and has been argued to be equivalent to the one offered by safety-oriented hypervisors [AMGP+19].

- **Application:** The Motion Planning module of an AEB case study. The module's functionality is explained in Section 4.1. In this case, the application exercises ten different system-calls (i.e., clone, futex(wait), futex(wake), ppoll, read, recvmsg, sendto(brake), sendto(hmi), write, writev).

- **System-context:** To simplify the analysis, we have decided to replace the cameras with an ego vehicle point of view driving video that collects different critical situations that may occur in real scenarios and enables the reproducibility of evaluation results. Although in this case we have decided to use a video as test-case input, the acquisition and analysis can be extended to a larger number of test-cases. In the analyses presented in this thesis, the system context has also been represented by simulating its Worst Case Scenario (WCS). For this purpose, we have relied on several articles that simulate the WCS for a real-time Linux system [Alt09, Alt16] and on the first articles of inherent non-determinism of the kernel [OMGFOO13]. Accordingly, to simulate the WCS, one of the partitions is heavily loaded to stress the system, specifically Partition 0 in this case (i.e., Stress function). A high CPU load is generated with *hackbench* and a high internal IRQ load is generated with a recursive search of patterns in the disk. Note that the LOPA architecture reduces the interference between partitions significantly [AMGP+19]. *Hackbench* is a widely used tool in the kernel development community to exercise the kernel in the corner cases so as to trigger worst case behavior as well as covering the usual type of requests (see Figure 4.3).

4.2.1 Layered isolation architecture

The Linux kernel provides several isolation mechanisms that have been widely used especially in the security domain and, increasingly, in other application domains like virtual servers or virtualized networks. These isolation mechanisms are used to build containers, and by these techniques, isolate them from the rest of the system. Therefore, in the event of a failure in one of these containers, this failure will have no consequences for the rest of the system. The techniques are also in use for sandboxing of not trusted applications as well as isolation of exposed services (e.g., web servers). The implementation of several isolation mechanisms allows the construction of a layered isolation architecture [PGB18]. SIL2LinuxMP has proposed multiple layers of protection that are semi-independent among them and collectively have two objectives: (1) achieve isolation between independent applications and (2) assure API and resource constraints (specified by a hazard analysis) compliance. The architecture is based on LOPA [PGB18], assigning multiple layers of protection to each hazard class.

In this use case, we use the mechanisms listed in SIL2LinuxMP project [PGB18] to achieve isolation of the Motion Planning software partition. In this way, we ensure that each partition has a dedicated CPU and memory range.

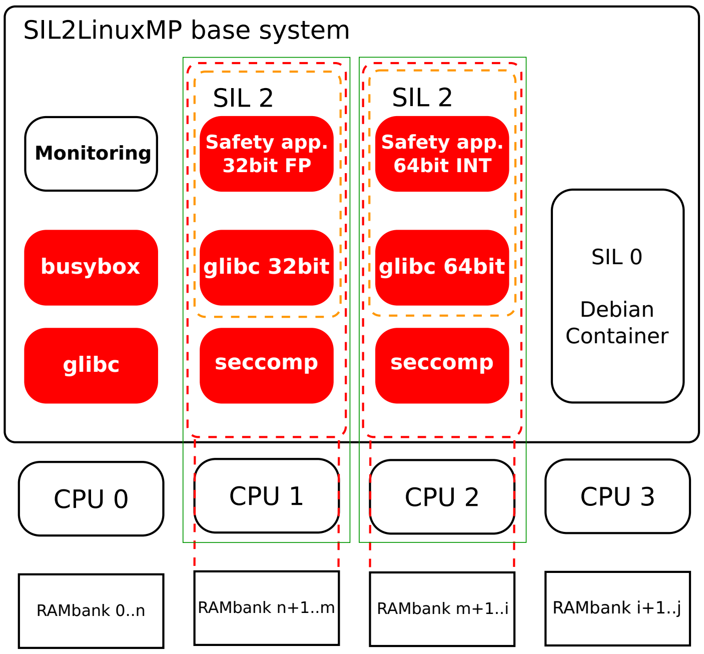


Figure 4.2 SIL2LinuxMP architecture [PGB18].

1. **Separate Namespaces:** It allows partitioning kernel resources such that one or various processes identify different resources from other sets of processes (PID, Inter-Process

Communication (IPC), cgroup, mount, UNIX Timesharing System (UTS), network and user). Thus, processors of an exact namespace have their own isolated instance of the global resources.

2. **System-call filtering (seccomp):** This kernel feature limits system calls from the container to the kernel.

3. **Control Groups (cgroups):** supports the assignment and limitation of the hardware resources to the processors (e.g., memory, CPU, I/O).

4. **CPU shielding:** supports the reservation of CPU resources to specific tasks, being able to assign one or several CPUs only to certain tasks.

5. **PALLOC allocator:** PALLOC is a Linux patch that minimizes the memory performance unpredictability in COTS multi-core processors by RAM-bank aware partitioning. It provides protection against applications accessing a wrong memory location and supports the partition of the cache memories.

Figure 4.2 illustrates the generic architecture proposed by SIL2LinuxMP based on the described layers. Namespace layer (1) corresponds to the smallest dashed square, which includes the safety application and the library. Diversity layer (2) is shown by the 32/64-bit nomenclature. The third layer, seccomp (3), is represented below the application and the library. Next, cgroups (4), is shown by the middle size dashed square with seccomp inside. CPU shielding (5) is identified by the continuous line square, which includes the CPU selection. Finally, PALLOC allocator (6) is presented with the biggest dashed square, which sets the RAMbank.

Considering the generic architecture proposed by the SIL2LinuxMP project, the architecture for the Motion Planning software partition is proposed in Figure 4.3. The architecture is based on four partitions, which included the alert functions. Nonetheless, Figure 4.3 depicts also functions that have not been implemented in this thesis to highlight opportunities for future works. We define the experimental setup (Figure 4.3) described below:

- **Partition 0:** We replace the content of the original partition 0 with 'stress', a high CPU load benchmark that enables the execution/injection of accelerated worst-case stress interferences [OGOO15].

- **Partition 1:** It executes the software partition and integrates safety-related tasks under analysis.

- **Partition 2:** The focus of the current research is the analysis of a single partition, so the specific aspects of a redundant partition are not considered in this scenario. However, a future extended analysis could consider the redundant architecture.
- **Partition 3**: It provides warnings to the driver through the Human Machine Interface (HMI). Some systems consider HMI as a SIL 1 or 2 task. However, in this case, to simplify the case study, we consider it to be SIL 0. In a future analysis, it would also be possible to execute additional non-critical tasks.

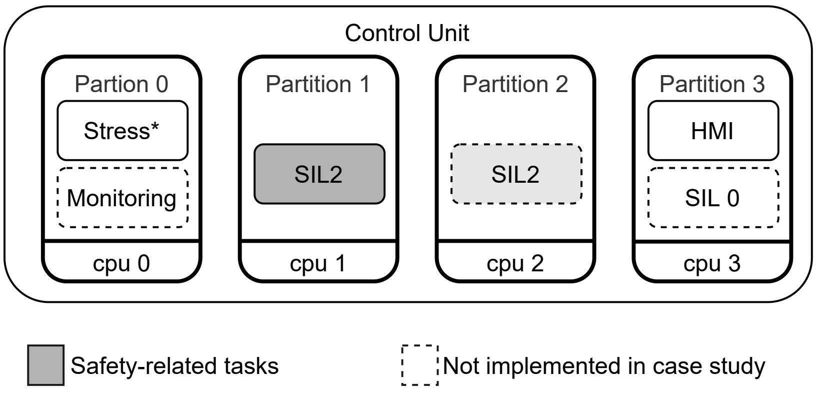


Figure 4.3 Architecture of the proposed case study (AEB system).

More detailed information can be found in the following publications [OSA, AMGP+19, PGB18].

4.3 Data set collection

The data acquisition process explained in Section 3.2 is employed for the AEB case study. Concerning the inherent non-determinism of the Linux kernel's execution paths, the test-case of interest is executed repeatedly during different test-campaigns. As a result, we get a certain number of test-campaigns each formed by a number of test-case executions.

With the purpose of executing test-cases with critical situations that may occur in real scenarios and enable the reproducibility of evaluation results, we replace the cameras with an ego vehicle point of view driving video. Therefore, the test-case that is repeatedly executed is a video with several critical situations where the AEB has to perform different actions.

As it is explained in Chapter 3, the methods are based on system-call traces. Thus, we record the kernel execution trace that the safety application executes and, afterward, we identify the invoked system-calls. An example of the system-calls sequence that is exercised in the AEB case study is provided below. The sequence also provides the MD5 hash value that identifies the trace of each system-call. Consequently, the same hash values mean that the system-call has exercised exactly the same function sequence.

Example of trace sequence: System-call sequences and the hash value for each trace.

```
...
S:sil2app:SyS_recvmsg()
E:sil2app:9fcb39ee2a9fd0785a4c8a301e5dea2d
S:sil2app:SyS_recvmsg()
E:sil2app:d36002618570643ae2f4171f5285d9ef
S:sil2app:SyS_ppoll()
E:sil2app:f040d8706f203c6159b04c0f24423c1e
S:sil2app:SyS_recvmsg()
E:sil2app:d36002618570643ae2f4171f5285d9ef
S:sil2app:SyS_recvmsg()
E:sil2app:d36002618570643ae2f4171f5285d9ef
S:sil2app:SyS_ppoll()
E:sil2app:f040d8706f203c6159b04c0f24423c1e
S:sil2app:SyS_recvmsg()
E:sil2app:d36002618570643ae2f4171f5285d9ef
S:sil2app:SyS_ppoll()
E:sil2app:29267707eadfb78aa36fff64b73bd324
S:sil2app:SyS_writev()
E:sil2app:9b6a2d29381d2a97da6ccb8296357207
S:sil2app:SyS_recvmsg()
E:sil2app:d36002618570643ae2f4171f5285d9ef
S:sil2app:SyS_recvmsg()
E:sil2app:d36002618570643ae2f4171f5285d9ef
...
```

S: prefix designates the identified system-call, while *E:* prefix is used to provide the MD5 hash. The hash value represents the identification number of the function sequence that occurs in the system-call. In the case of the AEB case study, the SIL 2 application exercises the following system-calls:

a) clone b) futex(wait) c) futex(wake) d) ppoll e) read

f) recvmsg g) sendto(brake) h) sendto(hmi) i) write j) writev

There are repeated system-calls but with different inputs (e.g., futex(wait), futex(wake)). This is because they are called with different input parameters. For example, the same application can open a file or a device, but the traces may have nothing to do with each other. An example of a system-call trace is provided below. The trace corresponds to the system-call *ppoll()*, specifically to the most frequently executed trace.

```
SyS_ppoll()
   poll_select_set_timeout  ()
  do_sys_poll ()
    __check_object_size ()
      is_vmalloc_or_module_addr()
      pfn_valid ()
        memblock_is_map_memory()
      check_stack_object ()
    __fdget ()
      __fget_light  ()
        __fget ()
          __rcu_read_lock ()
          __rcu_read_unlock ()
    sock_poll ()
      unix_poll ()
    fput ()
    __fdget ()
      __fget_light  ()
        __fget ()
          __rcu_read_lock ()
          __rcu_read_unlock ()
    eventfd_poll  ()
    fput ()
     poll_freewait  ()
  poll_select_copy_remaining  ()
```

Figure 4.4 shows the number of *unique traces* that have been recorded during 250 test-campaigns. The graph shows that the number of unique paths varies depending on the test-campaign. The horizontal label represents each of the test-campaigns formed by three thousand executions, and the vertical label collects the number of *unique traces* per test-campaign in this system-call.

As Figure 4.4 shows, about 200 different system-call traces occur per campaign. Table 4.1 collects the results of the recorded data set. It lists the system-calls that are exercised

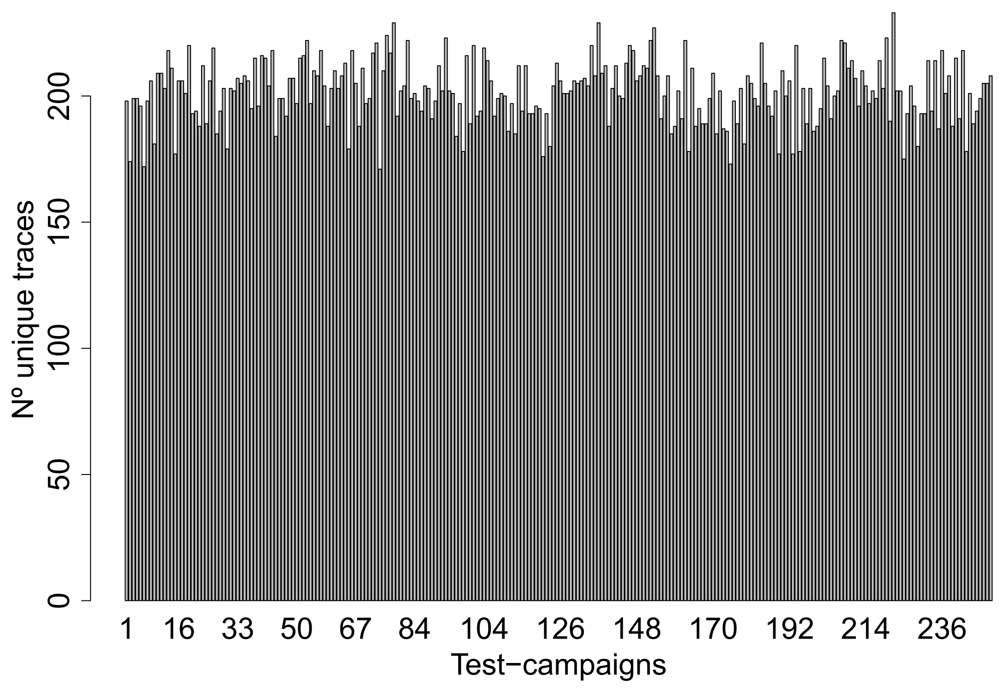


Figure 4.4 Number of *unique traces* in 250 test campaigns.

by the application, the number of times the system-calls have been called, the number of different traces that have been executed in each case, and, finally, the proportion of times that the most common paths have been executed.

It is observed that the application makes considerably different use of each system-call, executing certain system-calls more frequently than others. The inherent non-determinism displayed by each system-call also shows a significant difference. We have system-calls with high variability (e.g., clone) and others with low variability (e.g., futex(wake)). Besides, if we inspect the execution frequency of the distinct traces, it is possible to observe that there is a reduced number of traces that are commonly executed in each system-call.

Note that the total number of different traces is not the sum of the second column's results. In this case study, we identified equivalent traces between *sendto(brake)* and *sendto(hmi)*. Besides, the total value of *'Most common-trace (%)'* is Not Applicable (NA) as the value is only representative with respect to each system-call.

Table 4.1 Results of the recorded data set divided by system-calls.

System-calls	Number of executions	Number of different traces	Most common-trace (%)
clone	439923	1629	34.17
futex(wait)	495675	13	89.12
futex(wake)	557887	6	88.77
ppoll	1842298	27	73.85
read	556426	42	41.82
recvmsg	3327092	40	86.76
sendto(brake)	493491	161	30.36
sendto(hmi)	490502	206	94.19
write	966482	121	98.74
writev	476686	264	43.55
TOTAL	9646462	2409	NA

4.4 Data set validation

In order to do accurate statistical analysis, it is necessary to validate the data set. As described above, the analysis performed in this work is based on a data set divided into a number of test campaigns, each of them consisting of a number of runs. Thereby, the correctness of the data set needs to be validated from a quantitative and qualitative point of view. On the one hand, it is necessary to validate that the number of test campaigns is sufficient for statistical analysis. On the other hand, it is also necessary to validate that data set qualitatively.

4.4.1 Quantitative

To validate the size of the data set, we propose a method based on Extreme Value Theory (EVT). EVT is a statistical field that focuses its study on the events associated with the tail of the distributions and, consequently, can be used to determine the probability of extreme deviations that are characterized by their stochastic behavior. Specifically, EVT allows a quantitative classification of outliers as well as a judgment if the observed set is complete with respect to the expected range. Therefore, it allows estimating if we should expect to see a higher range of non-determinism if we would collect more test-campaigns. Note that it does not state the total number of traces that we should expect, but the extreme number of traces to expect in a test-campaign. EVT is also commonly defined as the counterpart of the Central Limit Theorem (CLT). While CLT examines a complete distribution behavior, EVT is focused on the tail of the distributions.

There are three types of Extreme Value Distributions (EVDs) (Gumbel, Fréchet and Weibull) that can model the limits for any set of data [Faw13]. Generalized Extreme Value

(GEV) combines Type I - Gumbel ($\xi = 0$ - or close to 0), Type II - Fréchet ($\xi > 0$) and Type III - Weibull ($\xi < 0$) EVDs. The distribution type of a model depends on the behavior of the distribution tail. GEV combines the previously described three distribution types, allowing the use of EVT without the need to know the detailed behavior of the distribution tail. The cumulative distribution function of the GEV distribution is:

$$F(\chi;\mu,\sigma,\xi = exp\{-[1+\xi(\frac{\chi-\mu}{\sigma})]^{-1/\xi}\} \tag{4.1}$$

μ: location parameter; σ: scale parameter; ξ: shape parameter

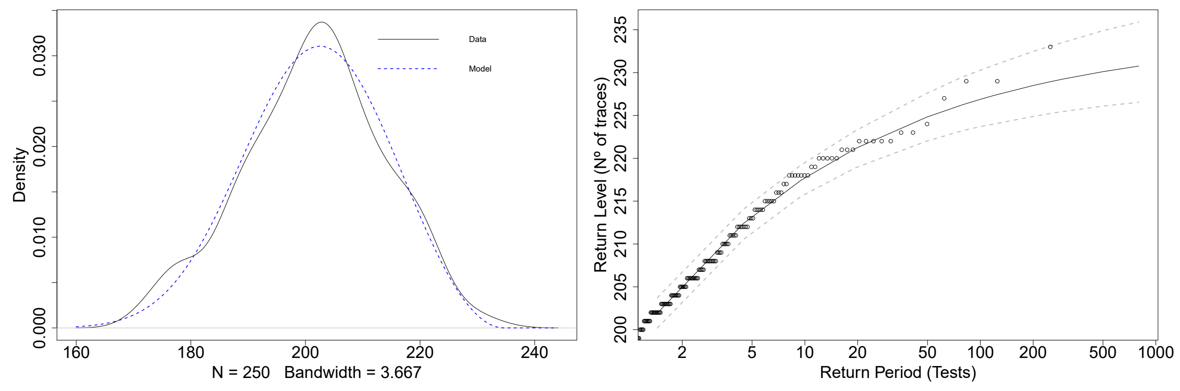
The three parameters shown in Equation 4.1 need to be estimated in order to use GEV. L-moments is a common and flexible method used for the estimation of these parameters. The aim of this method is to estimate a suitable probability distribution by certain linear combinations of order statistics [Hos90].

The idea is to calculate the extreme value (and its Confidence Interval (CI)) of the number of *unique traces* to expect in a single test campaign, but for numbers of test-campaigns higher than the number of effectively executed (e.g., 500, 1000, and 10000), by applying parameter estimation techniques to the collected data set. If the calculated extreme values and CIs show stability and convergence, one can conclude that the number of test campaigns of the data set is sufficient. It states that the recorded data set is complete with respect to the expected range.

The parameters estimation for the recorded system-calls of the AEB application result in a Gumbel distribution (location: 197.56, scale: 12.66 and shape: -0.34). Figure 4.5 illustrates two different plots related to the GEV model result. The left graph is the density plot of the empirical and GEV fitted model, and the right plot explains the return levels with 95% CIs.

With the collected data set and applying parameter estimation techniques, it is possible to calculate the return values of the extreme values for a higher number of test-campaigns. In this case, it is calculated for 500, 1000, and 10000 test-campaigns. Additionally, the CI of the return values is calculated by bootstrapping. Bootstrap is a widely used statistical tool that allows estimating accuracy [JWHT13]. It is a test based on random sampling with replacement. Therefore, it allows generating many *'new'* data sets by randomly sampling with replacement from the recorded data set.

Table 4.2 shows the return values with their corresponding CI for 500, 1000 and 10000 test-campaigns. Note that CIs are calculated by bootstrapping with 1000 simulations. Table 4.2 shows that the maximum number estimated by EVT is almost 26, with near values of CIs. Furthermore, the CIs and return value results show that the outcome is stable and convergent enough to justify that the size of the data is valid. In other words, the recorded data set is



(a) Density (b) Return level

Figure 4.5 Density and return value plots from the GEV distribution.

large enough and it can be concluded that recording more data will not provide significant information gain.

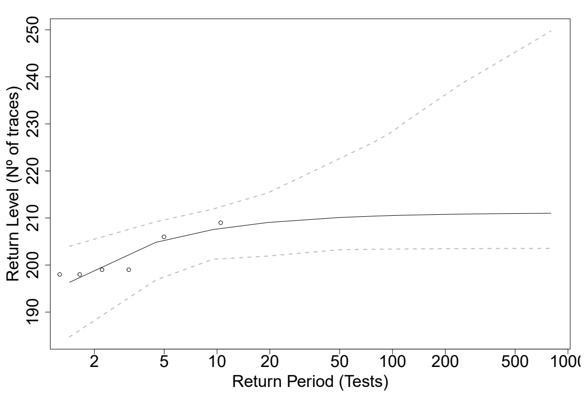
Table 4.2 Obtained extreme values.

Campaigns	95% lower CI	Estimate	95% upper CI
500-Test return level	225.7	230.1	235.5
1000-Test return level	226.0	230.6	236.6
10000-Test return level	226.4	231.5	238.8

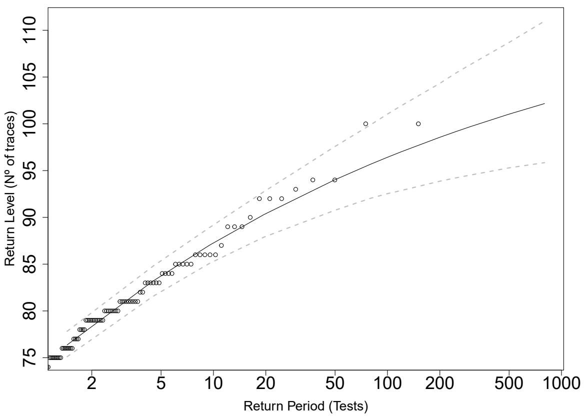
In order to properly understand the function of EVT, if instead of collecting the 250 test-campaigns we collect only ten, we would get the results shown in Figure 4.6a. As it can be observed, the result is not stable since the number of campaigns increases and the CI value also increases considerably. This means that the data set formed by ten campaigns is not enough for the analyses and, as a result, a larger number of test-campaigns needs to be recorded. Apart from that, if the size of each test-campaign is decreased from 250 executions to ten, we also can visualize the instability of the results.

4.4.2 Qualitative

The application of EVT requires that the data set is independent and identically distributed (i.i.d). This means that the execution frequency distribution of each test-campaign is identically distributed and that are independent between them. Hence, in this phase of the analysis, it is necessary to analyze if the data set obtained from the test-campaigns fulfills this require-



(a) EVT results with CI obtained from 10 test-campaign.



(b) EVT results with CI obtained from smaller test-campaigns.

Figure 4.6 Return values of smaller data sets.

ment. While the independence of the data set can be calculated with Ljung-Box [LB78], the analysis of the identical distribution can be done by Kolmogorov–Smirnov test [Jr.51].

Besides, with the purpose of determining if there is any dependency between the execution paths, an autocorrelation analysis is performed. Autocorrelation represents the level of similarity between the data set of a particular time series and a delayed version of this data set. Hence, by examining the autocorrelation of a data set, it is possible to know if there is a probable dependency between execution paths. In other words, the aim is to identify if one execution path is caused due to the execution of a previous one. All the execution traces must be independent of each other, meaning that the occurrence of a particular trace does not cause another trace to be more or less likely.

It shall be noted that although Ljung-Box and autocorrelation analyses seem similar; we propose to apply them to different types of data. In the first case, it studies the independence of the resulting paths between test-campaigns. Whereas in the second case, the analysis is focused on the correlation of every single trace throughout the same test-campaign. As a result, we verify the independence between test-campaigns and between execution paths.

independent and identically distributed (i.i.d)

The use of EVT can be validated by the study of i.i.d as it is necessary to analyze if the data set obtained from the test-campaigns fulfills this requirement. While the independence of the data set is calculated with Ljung-Box [LB78], the analysis of the identical distribution is done by Kolmogorov–Smirnov test [Jr.51]. Basically, with Ljung-Box it is possible to test the lack of serial autocorrelation and the Kolmogorov-Smirnov test determines if two different samples belong to the same distribution.

If each test campaign is analyzed individually, it is possible to see that in each system-call there is a common path executed the vast majority of the times and that there are a number of traces executed only a few times or even just once. For instance, 98.36% of traces of system-call *write()* belong to the same execution path. Therefore, in order to assess i.i.d more efficiently, we exclude the most *common-traces* from the analysis since keeping the *common-traces* would be detected as a dependency.

The classification between *common-traces* and *rare-traces* is performed employing Shannon Entropy. Shannon Entropy quantifies the amount of information of a data set, thus, we select a dividing ratio where the two classifications provide the most equivalent amount of information [Sha48]. Shannon Entropy is measured with Equation 4.2, where p_i represents the fraction population of the event i and n the number of different events.

$$H(x) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{4.2}$$

If we order the traces from the most common to the least, we can estimate the point (x) where two groups offer the most similar amount of information. H_1 represents the *common-traces* and H_2 to the *rare-traces*. Note that n represents the maximum observed execution frequency.

$$H_1(x) = -\sum_{i=1}^{x} p_i \log_2 p_i \tag{4.3}$$

$$H_2(x) = -\sum_{i=x+1}^{n} p_i \log_2 p_i \tag{4.4}$$

We use the sub-set formed by the *rare-traces* to analyze the i.i.d, obtaining positive i.i.d results, thus allowing the use of EVT. Both, Ljung-Box and Kolmorogorov-Smirnov, provide results with probabilities higher than 5%, which is a widely accepted threshold in different domains for the use of EVT [QLG16, GT09, TLW$^+$16].

Figure 4.7 illustrates the Empirical Cumulative Distribution Function (ECDF) of two randomly selected test cases showing that they are identically distributed.

Autocorrelation

Besides, with the purpose of determining if there is any dependency between the execution paths, an autocorrelation is performed to validate the quality of the data set for further analysis. Autocorrelation provides the level of similarity between the data set of a particular

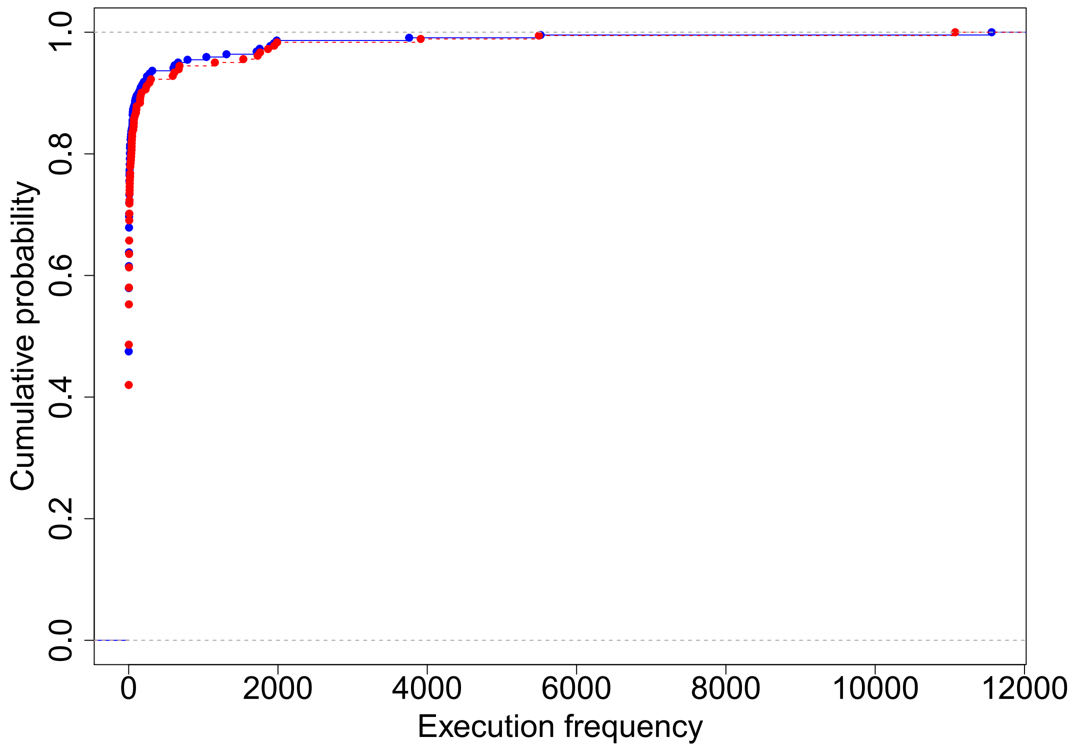


Figure 4.7 ECDF of a randomly selected test campaign of system-call clone

time series and a delayed version of this data set. Hence, by examining the autocorrelation of a data set, it is possible to know if there is any correlation between execution paths. In other words, the aim is to identify if one execution path is caused due to the execution of a previous one. All the execution traces must be independent of each other, meaning that a particular trace's occurrence does not make another trace to be more or less likely.

Figure 4.8 shows the autocorrelation result of a randomly selected test campaign. As expected, in lag 0, the correlation is 1, but the value is negligible in the rest of the lag values. The autocorrelation study has also been performed with the data obtained from all the tests campaign, obtaining equivalent results.

The blue lines in Figure 4.8 show a maximum value (excluding lag 0 - the trace with itself) and a minimum value. Autocorrelation maximum value obtained in all test campaigns is 0.03584, the minimum is -0.06348 and the mean is 0.03472.

It shall be noted that although Ljung-Box and autocorrelation analyses seem similar, we propose to apply them to different types of data. In the first case, it studies the independence of the resulting number of *unique traces* per test campaign. Whereas in the second case, the analysis is focused on the correlation of every single trace throughout the same test campaign.

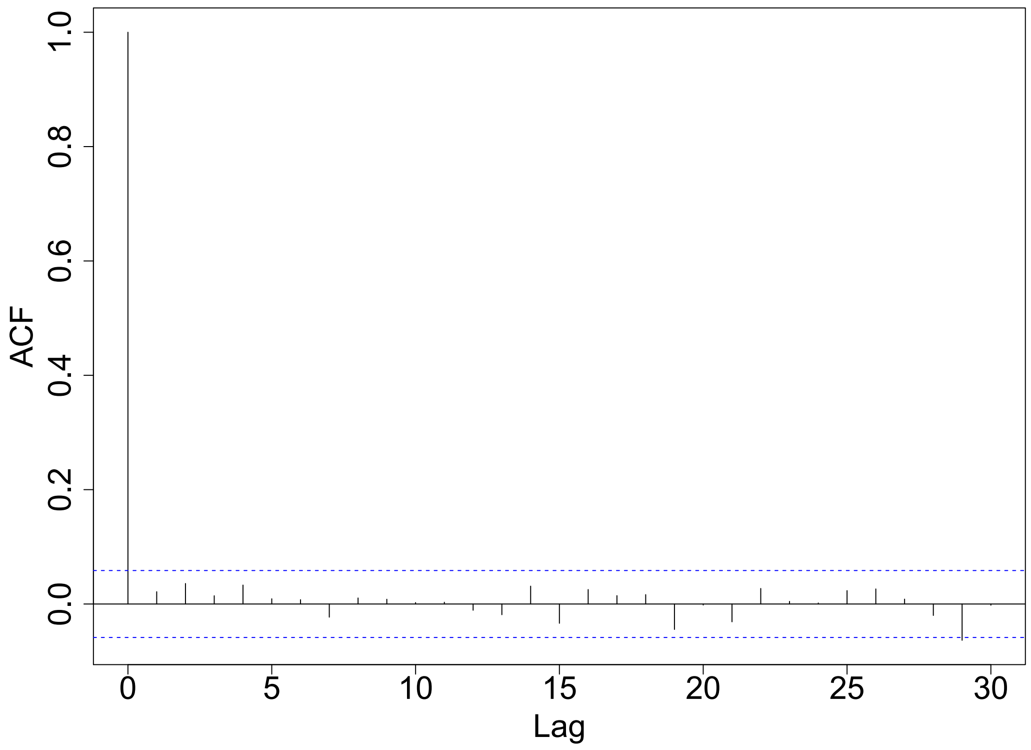


Figure 4.8 Autocorrelation of a randomly selected test campaign of system-call clone.

4.5 Summary

The chapter presents the case study that is used as a guiding example for the contributions carried out during the research activities of the thesis. The case study is a research-grade AEB system based on Linux kernel and ML algorithms and representative of next-generation safety-related systems. Research activities, and therefore the data collection, are focused on the AEB system's control unit, where the Linux kernel runs on a defined system-context.

The chapter also presents the data that has been recorded with the case study. The data and the conducted analyses provide key information to continue the research. Firstly, the data demonstrates the inherent non-determinism of the Linux kernel, as it shows the existence of several execution paths for each system-call. Secondly, the chapter shows how the data is analyzed to conclude that the data set is large enough to continue the study. Finally, it shows the independence between the execution traces.

Chapter 5

Parametric Statistical Method for Software Execution Test Coverage

The review of the state of the art in Chapter 2 allows identifying the main gaps and limitations in terms of test coverage for software such as the Linux kernel. This Chapter presents a novel method that aims to cope with the identified constraints. The presented method brings a paradigm shift in test coverage as the objective is no longer attempting to achieve full coverage of the complete source code, but rather we estimate the number of traces that have a relevant probability of being exercised in a specific use case. In other words, the goal is to estimate the number of execution paths that the safety function under test can exercise in the Linux kernel by employing statistics to the data obtained from previously tested execution paths. In this manner, it is possible to quantify the number of traces that have not yet been observed during testing but are to be expected and, thus, quantify the uncertainty. Note that this is very much distinct from the paths that could theoretically be reached (e.g. think of theoretically possible infinite retries in some cases).

For this purpose, the conducted research examines the inherent diversity of the Linux kernel execution. As Chapter 4 presents, the same call with identical inputs can exercise different possible paths to be executed. But do all traces have the same probability of execution or do they show different execution frequencies?

This chapter describes a novel technique for parametric statistical analysis that pursues two main objectives: (i) estimate the number of traces that an application can exercise at the kernel-level on a specific target platform and system-context, and (ii) calculate the number of traces that have not yet been exercised and, thus, quantify the uncertainty. The study is based on the assumption that the system state is stationary.

5.1 Description of the approach

The research described in this chapter is based on modeling the appearance of new *unique traces* (i.e., execution of untested traces) in the Linux kernel based on the data of a system under test. Parametric statistical methods assume that the underlying population of a data set can be described with a model based on a fixed set of parameters. Statistical models allow making generalizations or predictions of larger populations [Sal10]. Consequently, the model obtained from the recorded data set allows predicting the behavior of the system if it would be infinitely tested and, as a result, estimating the total number of *unique traces* that the system can exercise. Parametric statistics involve understanding in high detail the *"mechanism"* that generates the data since the model to be selected has to be consistent with a theoretical explanation. Each existing model has a set of requirements that have to be respected by the application under review. Therefore, it is not only that the data has to fit the model mathematically, but that the application under study is in accordance with the theory of the model of the generative process. It is true that there are data analysis applications that simply look for the model that mathematically best fits the data. However, this would hardly be acceptable in functional safety. A rigorous explanation of the model being implemented is needed, and not just a *"goodness of fit"* result. Besides, with the *"best goodness of fit"* approach, the resulting model for one use case would not necessarily fit another use case. Therefore, if the theoretical explanation of the presented approach matches the data obtained from the described use case (see Chapter 4), surely the method will be suitable for other use cases. For example, the method presented in this chapter is also analyzed with an additional case study (see Annex A).

The statistical analysis that we propose is based on count data modeling, specifically modeling the appearance of new *unique traces*. In other words, modeling the execution of traces that were not executed previously during the testing phase. Therefore, if we take an application and run it repeatedly we can see how new traces appear as the testing effort increases. So, once these appearances are recorded, it is possible to model the count of them and, later, estimate the appearance behavior of these traces if the system is exercised infinitely. Algorithm 1 explains the count values of the appearance of new *unique traces*. *UniqueTraces* is a vector that collects all these counts and, thus, the counts that we assume that decrease while the test-campaign number increases.

As an aid to the reader, a rough intuitive model might help here. If we assume that there is a process that allows different paths to be invoked based on dependency on an uncontrolled system state, then we would expect that the rate of new paths observed over time steadily decreases. This count cannot be negative and would (ideally) have a trend towards zero. Thus, it seems reasonable to initially expect that a member of the exponential distributions

Algorithm 1 Count of unique-traces to be modeled.

```
1: for $testcampaign = 1, 2, \ldots$ do
2:     $Count \leftarrow 0$
3:     for $execution = 1, 2, \ldots, N$ do
4:         if execution's MD5 not previously seen then
5:             Count++
6:         end if
7:     end for
8:     $UniqueTraces[testcampaign] \leftarrow Count$
9: end for
```

may fit the observed counts of new unique paths being emitted by the system. Hence, the analysis is based on the following initial assumption:

- **Assumption:** the number of new traces appearing per test-campaign will decrease while the number of test-campaigns increases as the number of traces is finite.

If we fit a model to the obtained data during testing, we can predict the behavior of the system if it is tested for an infinite amount of time. The model describes the appearance of all the existing *unique traces* and, consequently, we can calculate the total number of *unique traces* and the execution test-coverage.

5.2 Different probabilities of execution

In order to develop the research, it is first necessary to analyze the data set that we have recorded. Table 4.1 (Chapter 4) shows that in each system-call there is a common path. This is also shown in Figure 5.1. But what does this mean exactly? Examining the data set we identify a shared pattern between the system-calls. The data set shows that in all of the system-calls there are a small number of paths that constitute the vast majority of executions (i.e., common-paths) and that there are several paths rarely executed (i.e., rare-paths). This has already been reported in previous studies by Okech et al. [OGOO15, OMGF14, OMGFOO13, OMG10]. Figure 5.1 illustrates on a logarithmic scale how there are a small number of *unique traces* that are executed with a significantly higher frequency. The shown data corresponds to the entire data set, with all test-campaigns and system-calls.

From the results, we can conclude that the execution of each *unique trace* is not equiprobable. Each execution trace has a distinct execution probability. In this case, if we knew all the existent possible traces and their probabilities the sum would be equal to one ($\sum_{i=1}^{S} p_i = 1$).

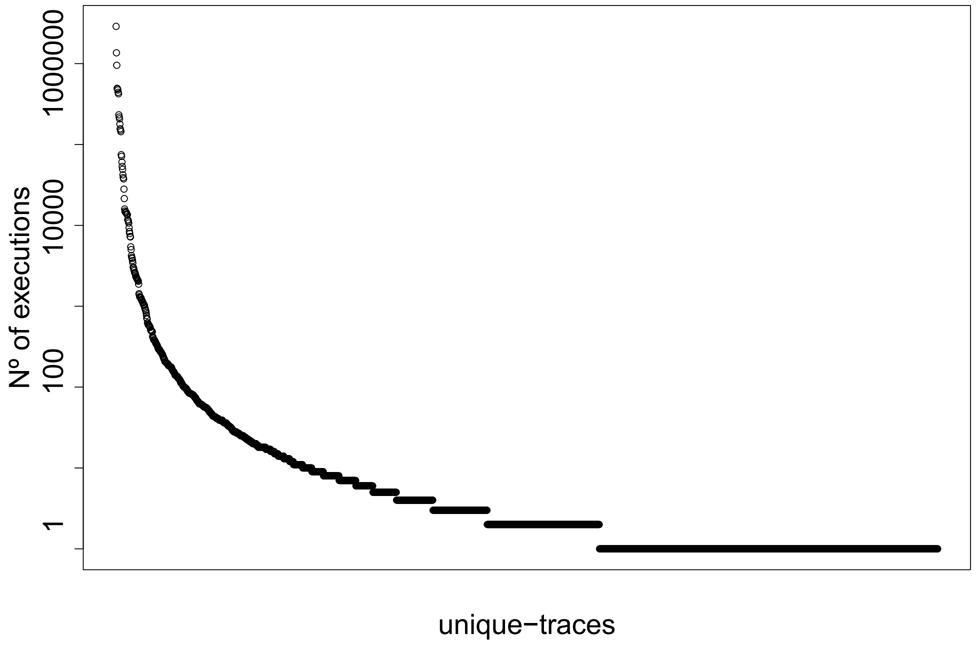


Figure 5.1 Execution frequency of the different *unique traces*.

5.3 Modeling count data

For the purpose of achieving the goal of estimating the number of traces that a binary code can exercise in the Linux kernel, this section focuses on modeling the appearance of *unique traces* during testing over time. As a result, it is necessary to model the recorded data set obtained in order to conduct the statistical study. The statistical model describes the recorded data set with a mathematical expression, which standardizes the way data elements relate to each other [Hil14].

Poisson regression is a generalized linear model used for modeling count data. Regression analysis is a statistical method employed to estimate the relationship between the dependent variable and one or various independent variables [CT13, JKK05]. Poisson regression assumes that the count data follows the Poisson distribution, a discrete distribution described with a single parameter (λ). This parameter is the mean number of occurrences over a given interval. Equation 5.1 describes Poisson's probability mass function, where x is the number of occurrences.

$$P(x) = \frac{\lambda^x e^{-\lambda}}{x!} \tag{5.1}$$

Poisson regression is an adequate generalized linear model for the analysis if the following model assumptions are met [Hil14]:

- The response variable is a non-negative count per a given interval.
- The response variable follows a Poisson distribution. Poisson distribution describes the probability that a certain number of events will occur during a certain period.
- The mean and the variance are equal. Therefore, the dispersion has an approximate value of one.

$$Var(X) = \lambda \tag{5.2}$$

- Observations are independent.
- The log of the mean rate can be modeled as a linear function.

$$log(\lambda_i) = \beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip} \tag{5.3}$$

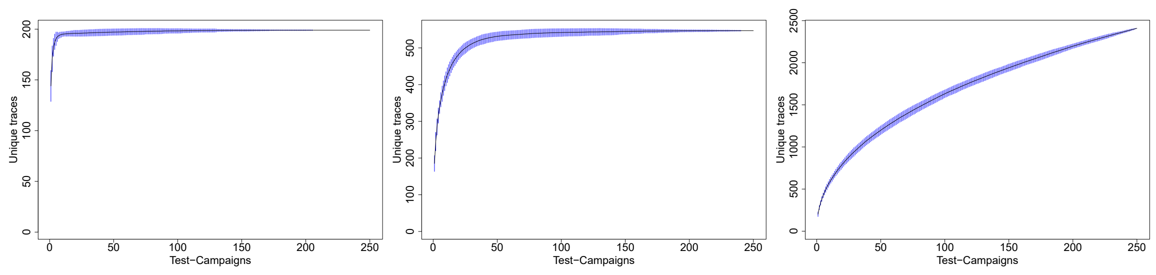
or equivalently,

$$\lambda_i = e^{\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip}} \tag{5.4}$$

5.3.1 Law of rare events

Poisson distribution is commonly employed to describe the distribution of rare events in a large set of observations [Ste10]. The *"law of rare events"* asserts that *"the total number of events will follow, approximately, the Poisson distribution if an event may occur in any of a large number of trials, but the probability of occurrence in any given trial is small"* [CT13]. Consequently, it is rational to assume that the emergence of a new *rare-trace* is at least approximately a Poisson process. In other words, a new *rare-trace* has a small probability of execution and is equally likely to appear at any time [Dow13]. We know from Chapter 2 that the non-determinism of execution paths is caused by different asynchronous events, including state changes in shared resources, that occur randomly [OSA]. Moreover, from Figure 5.1, it is possible to identify the different execution frequencies between paths. The frequency of execution of a path depends not only on the frequency at which the system-call was called but also on the probability of execution of each particular path. After all, some system states are more frequent than others. Therefore, each system-call with the same input parameters can follow different execution paths, but each existent path has a different execution probability.

Taking into account the features aforementioned, execution paths can be classified into two groups: *common-traces* and *rare-traces*. We consider *rare-traces* those execution paths that occur infrequently. If the non-parametric estimators employed in Chapter 6 are examined, it is possible to notice that they mainly focus on species that appear rarely (e.g., 1, 2, or less than 10 times). Chao et al. stated that events with a lower frequency of occurrence provide more information for species richness [Cha84, CC16]. After all, events that occur commonly provide almost no information about rare events [CC16], which are the ones we are examining and that may follow a Poisson distribution. One way to visualize this is by comparing the accumulation curves of common and rare traces. Figure 5.2 shows how the asymptotic behavior is significantly different between the data sets containing common paths and only rare-paths.



(a) >100 (b) >10 (c) all

Figure 5.2 Accumulation curve of: (a) path with execution frequency >100, (b) paths with execution frequency >10 and (c) the whole data set (frequency ≥ 1).

As Figure 5.2 shows, common paths have different behavior and do not provide information on the *rare-path's* behavior. Hence, it is legitimate to sort out common paths from the analysis and use an exponential family model for the *rare-traces*. From Table 4.1 in Chapter 4 we see that there is at least one common path that is executed the vast majority of the times (e.g., the *common-traces* of *write* or *sendto(hmi)* system-calls are executed more than 90% of the times). Hence, we expect that the rate of new paths observed over time steadily decreases in *rare-traces*, without taking into account *common-traces*. Furthermore, this assumption is reinforced by the information presented in Section 5.2, where it is shown that the traces have different execution probabilities. As a result, we can assume that the highest probable traces would appear at the beginning of the testing phase and that the lower probable paths would be distributed along with the testing phase.

The classification between common and rare traces is not straightforward. The main issue is to find an adequate method to determine a cut-off method for *rare-traces*. Which is the execution frequency to consider a trace rare? In the following Chapter, the reader will notice that the non-parametric species richness estimators use different cut-off frequencies (e.g., 1, 2, 10), which vary depending on the estimator. The majority of the estimators set the cut-off at a frequency equal to 2. However, we have not found an elaborate reasoning to extrapolate it to this method. In this case, in the absence of knowing an appropriate method to determine this value, we start performing the analysis with different cut-off limits and compare the results.

5.3.2 Poisson regression analysis

If the aforementioned theory is taken into account, including the law of rare events, it seems reasonable to analyze the appearance of previously untested *rare-traces* by means of Poisson regression. The appearance of *rare-traces*, at least theoretically, follows a Poisson distribution under the conditions of the system remaining in a more or less steady state of operations. In other words, constant mean execution load of the overall set of unrelated concurrent executing entities (processes, threads, irqs, etc.). Moreover, *rare-traces* have a low probability of execution, are likely to appear in any test-campaign and are independent between them. The response variable is a non-negative count per test-campaign, as we can count in each test-campaign the number of rare *unique traces* appearing for the first time. Furthermore, the count values are independent. As it is shown in Section 4.4.2, there is no correlation between the execution of paths. This means that the execution of a specific path does not cause another specific path to be executed. These results imply that the unobservable input(s) to the invoked functionality is in fact behaving like a truly random process as it is explained in Section 2.2.2. Consequently, the observations can be considered independent as the appearance of a previously untested path does not inform about the appearance of other path(s).

In order to perform the analysis, the obtained data set is reduced to paths that have occurred rarely. For instance, in the case that we set the cut-off limit of execution frequency to 10 (i.e. we only take those traces that have been executed 10 times or less in the whole data set), the data collected in Chapter 4 is reduced from 1996 to 1548 *unique traces*. According to this result, more than 75% of the traces are rarely executed, which confirms in a stark way the noted inadequacy of traditional testing methods. Algorithm 2 explains how we keep the *rare-traces* in the data set to model it later. So we inspect the execution frequency of every unique-trace (i.e., MD5) in the whole data set and the ones executed less than the specified cut-off limit (i.e., CutOffFreq) are kept for further analysis. After Algorithm 2 we execute Algorithm 1 to calculate the count values to model with Poisson. Thereby, the test-campaigns used in Algorithm 1 are formed by traces that occurred with a frequency below the specified cut-off, but in the whole data set (not just in that test-campaign). By applying both algorithms we achieve the counts of the appearance of rare unique traces.

Algorithm 2 Modeling of unique-traces appearance.

```
1: for $MD5 = 1,2,\dots,N$ do
2:     if $Freq(MD5) \leq CutOffFreq$ then
3:         Keep in the data set
4:     else
5:         Discard from data set
6:     end if
7: end for
```

Figure 5.3 depicts the appearance of *rare-traces* while the number of test-campaigns increases. As the graph shows, there is a tendency for the number of appearances to decrease as the testing effort increases. Therefore, a visual inspection of the graph confirms or, at least does not disprove, previously made assumption.

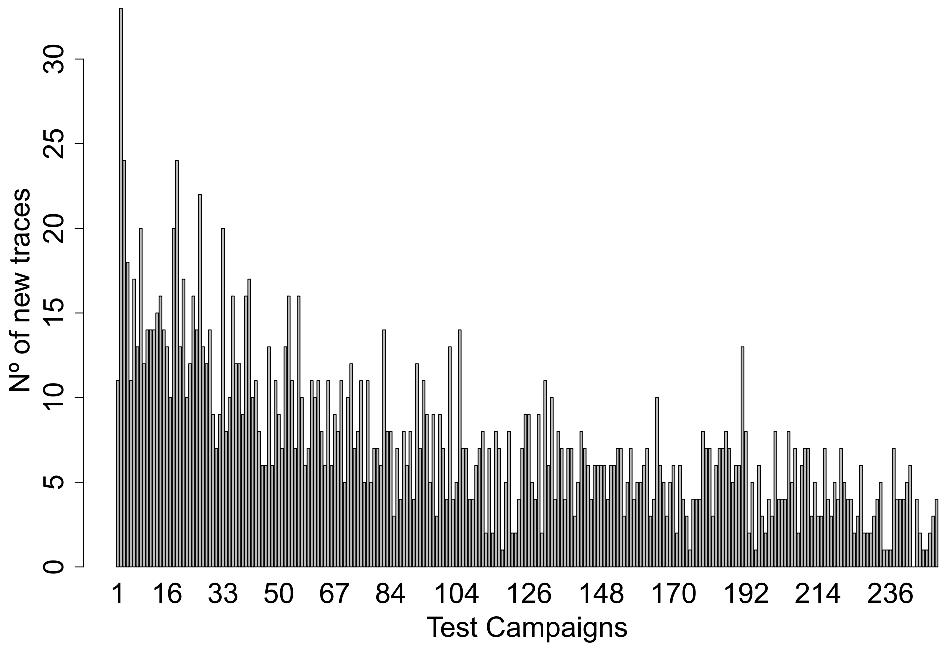


Figure 5.3 Rare-traces appearance through the test-campaigns.

Figure 5.5b shows, on the one hand, the points representing the number of *new* unique *rare-traces* identified in each test-campaign and, on the other hand, it also shows the estimated Poisson regression model. Besides, in order to reinforce this analysis, the CI is estimated by bootstrapping. Bootstrap is a common statistical tool used to estimate the accuracy of the model [JWHT13]. It randomly samples the data set with replacements in order to obtain *'new'* data sets. In this manner, it is possible to perform the regression analysis a large number of times with different data sets obtained with resampling the given empirical set and, therefore, estimating the CIs by examining the difference in the results. Figure 5.4 depicts the estimated model (black line) and the CI values (red and blue lines) obtained using bootstrapping.

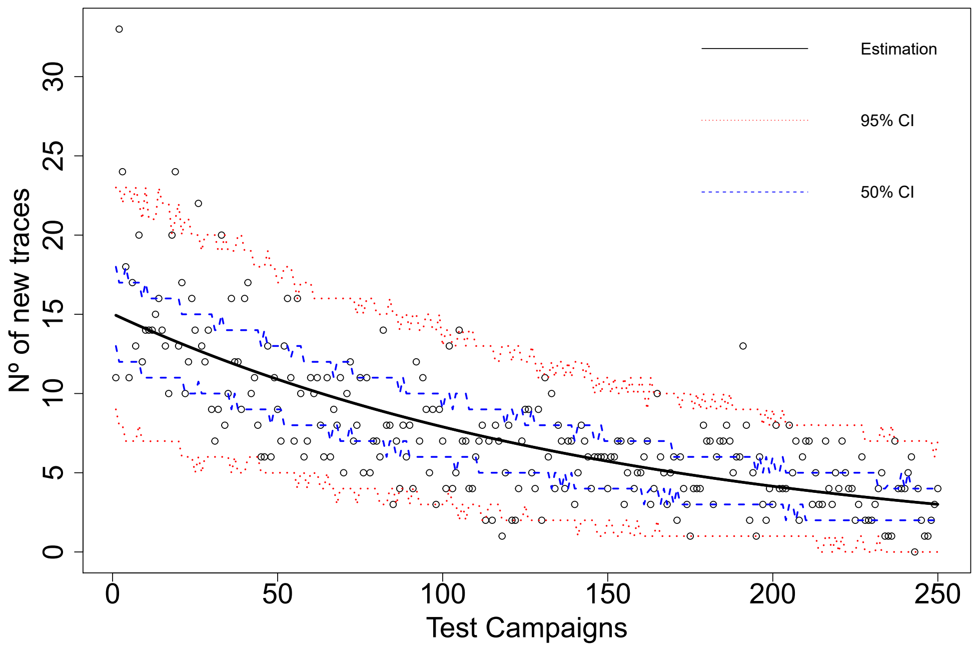
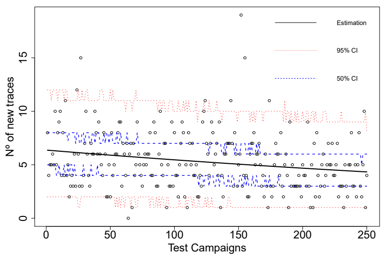


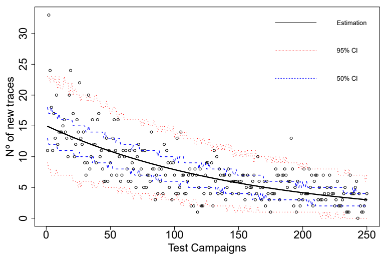
Figure 5.4 Number of new *rare-traces* identified by each test-campaign.

It is possible to perform a goodness of fit test by the visual inspection of the discrepancy between observed values and the Poisson regression. One of the indicators and arguments for this is that around 50% of the data set points are between the 50% upper CI and the 50% lower CI in Figure 5.4. Additionally, the dispersion parameter is calculated to assess that mean and variance are equal. Dispersion can be calculated with Pearson's Chi-squared statistic and the degree of freedom. If the dispersion value is close to 1, it means that the data set fits a Poisson model. For our data set, obtained through 250 test-campaigns, the resulting dispersion parameter is 1.46. Hence, we conclude that it is possible to use Poisson regression.

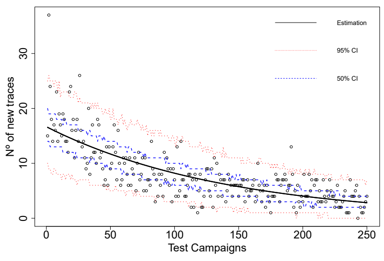
The fit of Poisson regression across different cut-off values is shown in Figure 5.5. The difference in the model can be observed if we estimate with a data set consisting of *unique traces* that have only been executed at least twice (Figure 5.5a), ten times (Figure 5.5b), or fifteen times (Figure 5.5c) times.



(a) ≥ 2



(b) ≥ 10



(c) ≥ 15

Figure 5.5 Poisson regression: (a) path with execution frequency ≥ 2, (b) paths with execution frequency ≥ 10, and (c) paths with execution frequency ≥ 15.

5.3.3 Merging all system-calls

Even if the traces are collected by differentiating the system-calls, this approach performs the analysis with all the system-calls at the same time in order to achieve the test coverage of the application. However, to carry out the study with all the system-calls together, it is necessary to validate some requirements of the data.

This only is a valid approach if the system-calls exhibit comparable path developments. Besides, any reasonable application is legitimately assumed to utilize a reasonable subset of system-calls, which avoids any extremes in path development. We also assume that the variability (effective permutations of path deviations) is caused by a common mechanism. This claim needs to be further investigated, but initial assessments indicate that the prime source of deviating paths are generic detours induced by functionally unrelated system states (e.g. low watermarks in the memory subsystem being hit), which are not call dependent. Indications of this are:

- **Positional independence in the call tree:** the same sub-sequence (e.g., *rcu_note_context_switch()*) is observed in different traces within the same system-call at different positions, though with identical sub-sequence (see Section 3.2.3).

- **Call independence across system-calls:** the same sub-sequence (e.g., *rcu_note_context_switch()*) is observed in different traces across different system-calls.

As these deviations are the cause of the path variability and show independence, we assume that the behavior of *rare-paths* is highly independent of the specific call. A manual inspection of a randomly selected set of path deviations was conducted within and across the recorded system-calls to provide a preliminary verification of this assumption. The method is to choose recorded paths that have the same length and check the difference of those traces. Inspection reveals that the sub-sequences are at different positions in the trace while, otherwise, the traces are identical. Given the relatively small number of inspected sequences and the error-prone nature of manual inspection we do not yet consider this legitimate evidence for safety assurance, but plausible for this research. If this assumption of independence of these unrelated calls intercepting holds, that is, they are not call specific but rather state-dependent, then we should expect that the distribution untested paths appearance should not be significantly impacted by the call set being inspected. This can be quantitatively evaluated using cross-validation methods.

Cross-validation is a model validation technique that assesses the independence of results of statistical analysis. Therefore, it allows evaluating the stability of the results with the incorporation of all system-calls to the analysis. K-Fold Cross Validation analyses if all system-calls follow equivalent new trace appearances and, hence, ensure the steadiness of the results of the statistical study. In other words, it allows confirming that all the system-calls belong to the same population (as they share the same distributional properties and thus may be treated as a single population). K-Fold is based on examining the MSE of the test coverage result with different combinations (round-robin) of the identified group of traces (folds) [JWHT13]. Therefore, each fold is formed by the traces of a specific system-call. Nonetheless, as Table 4.1 indicates, each system-call provides a different number of executions and *unique traces*. As each fold needs to have a roughly similar length, the analysis is performed by dividing some system-calls into different folds (e.g., clone) and joining some of them into one fold (e.g., futex(wait), futex(wake)).

The main objective of the present analysis is to see that all the folds result in a similar MSE. As expected (see Figure 5.6), there is an error in the test coverage with fewer traces, since we do not count the traces from the fold that has been withdrawn. But if every result has a similar MSE (unimodal and normal), it can be assumed that all folds exhibit comparable path developments. This would be very different if we got, for example, two bells in the histogram. Therefore, in that scenario, we could say that there are two populations. Figure 5.6 depicts the histogram of the obtained MSE results. Performing the analysis with 100 folds allows having folds formed with a reduced number of traces and, thus, more adequately represent the system-calls with a small number of traces. The obtained MSEs are adequate to validate that the system-calls come from the same population as they all are in the same range and form what seems a normal distribution.

Thus, taking into consideration these highly likely assumptions, the branching behavior is semi-independent of the system-call. We can expect that some detour paths are only in a specific system-call, thus, for some of the detours, we have a small count which will cause a noisy shape. But for those sharing calls, it smooths the Poisson model considerably.

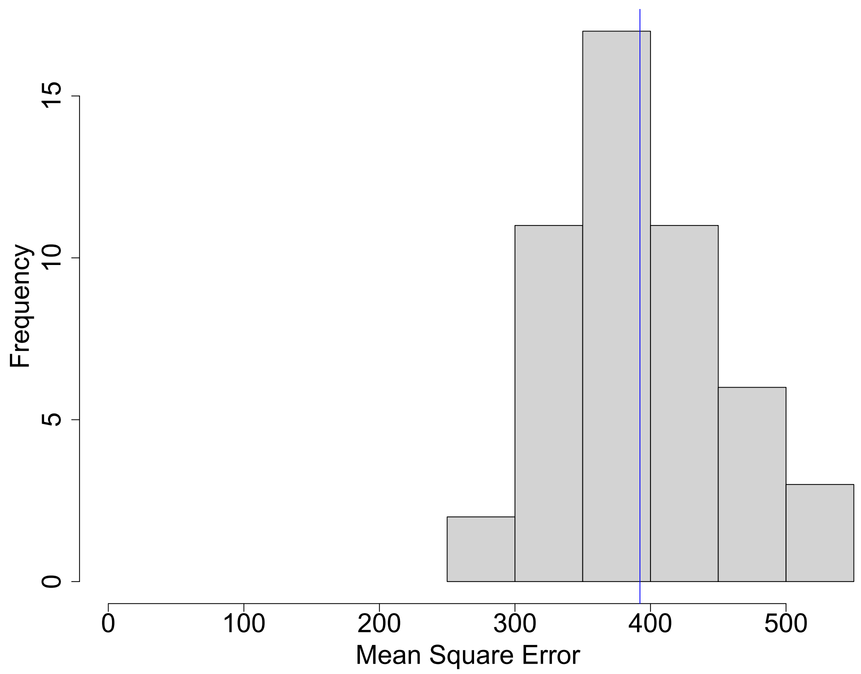


Figure 5.6 Histogram of obtained MSE values on 100 folds. Vertical line represents the mean value.

Additionally, this examination can be improved by using k-folds to perform Kolmogorov–Smirnov test. Similar to Section 4.4.2, Kolmogorov–Smirnov test allows to compare distributions, hence, it is possible to estimate if the whole data set or a date set withdrawing a fold is identically distributed. This analysis provides a positive result as withdrawing any of the folds does not change the distribution significantly and, thereby, can be considered identical.

5.4 Estimation of the number of paths

Having the model means that we have estimated the values of β_0 and β_1 of Equation 5.3. Therefore, we have the equation that describes the behavior of the system. The total number of traces that have a relevant probability of being executed can be estimated using this equation. The area under the curve that generates this equation represents the total number of traces that can be executed. Thereby, an improper integral to infinity of the mathematical expression of the obtained model provides the total number of rare *unique traces*. Equation 5.5 is the improper integral of the Poisson model, where β_0 and β_1 are the parameters that are estimated modeling the data set.

$$\int_1^{\infty} e^{\beta_0+\beta_1 x} dx = \lim_{b\to\infty} \int_1^{b} e^{\beta_0+\beta_1 x} dx \tag{5.5}$$

Figure 5.7 illustrates the area that is calculated to know the total number of execution traces that an application can exercise in the Linux kernel. In this study, the area is not just calculated in the 250 test-campaigns, but we examine the area of the curve if we were to exercise the system infinitely (i.e., infinite test-campaigns).

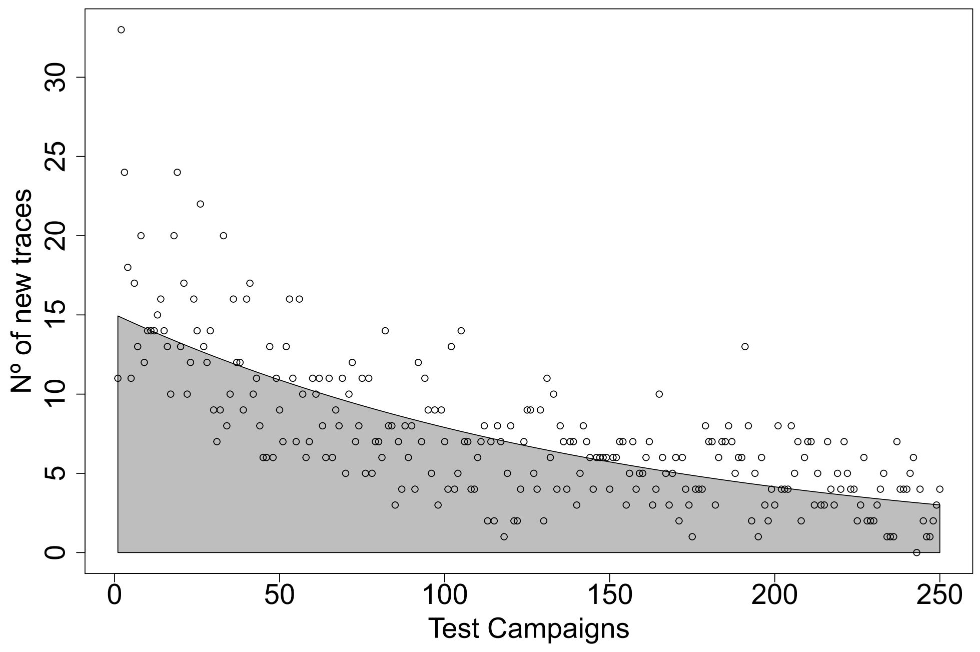


Figure 5.7 Area under the curve that is estimated to calculate the total number of traces.

The value obtained with this integral needs to be added to those that have been discarded for the analysis. That is, the number of *unique traces* that have been executed more times than the cut-off limit. In the case of cut-off ≤ 10 the calculated total number of traces is 2951, where 2409 have been already tested.

5.4.1 Different cut-off limits

Table 5.1 collects the total number of *unique traces* for different cut-off limits. The analysis is performed by resampling the test-campaigns without replacement. In other words, the order of the test-campaigns is changed a large number of times to obtain different count values. This allows estimating the mean of *total traces* and the *standard deviation*. Besides, this is the reason why the results of the cut-off equal to 10 does not completely agree with the ones presented previously. The results are the mean values obtained with reordering the test-campaigns several times. Table 5.1 also collects dispersion, Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) values. AIC and BIC are model fit quality estimators. These quality estimators provide lower results when the model fits more adequately [Hil14]. It is not that there is a maximum value that indicates that the model is not appropriate, but rather they allow making comparisons between different models. Therefore, the lower the value the better.

Table 5.1 Estimated total number of traces depending on the cut-off and model validation.

Cut-off Freq	**Total traces**	**Std Error**	**Dispersion**	**AIC**	**BIC**
2	4715.38	44.19	1.30	1169.23	1176.27
3	3808.86	19.52	1.39	1212.48	1219.52
4	3423.22	10.73	1.38	1223.42	1230.46
5	3275.47	8.52	1.40	1236.84	1243.88
6	3170.18	6.92	1.39	1242.68	1249.72
7	3092.47	5.96	1.43	1254.82	1261.87
8	3021.01	5.23	1.43	1259.11	1266.15
9	2972.85	4.27	1.47	1269.98	1277.03
10	2939.19	4.26	1.48	1273.29	1280.33
...	...	...	...	...	...
20	2797.49	2.75	1.68	1325.38	1332.42
...	...	...	...	...	...
30	2753.05	2.60	1.95	1376.34	1383.38
...	...	...	...	...	...
40	2719.39	1.93	2.07	1403.19	1410.23
...	...	...	...	...	...
50	2706.83	2.10	2.18	1427.98	1435.02

Poisson model requires that the mean and the variance values are equal. Overdispersion or underdispersion occurs when data are correlated or not Poisson distributed. An indication of overdispersion or underdispersion is having a variance larger or lower than the mean, respectively [Hil14]. The model fits Poisson adequately when the dispersion parameter is ≈ 1. A value larger than 1 means overdispersion and lower than 1 underdispersion. Table 5.1 shows that dispersion values are in an adequate range. Besides, the obtained dispersion results show that the data follows the law of rare events as the dispersion increases with higher cut-off limits or with the incorporation of more *common-traces*. Thereby, it can be considered key evidence that the system follows a Poisson distribution.

Apart from that, the results with different cut-off limits (Table 5.1) can be considered stable. There are no significant variations between the values obtained. It could be stated that they all maintain a certain concordance. However, it can also be observed that stabilization is greater with higher cut-off limits. The standard deviations results show also certain stability, with a trend to decrease with higher cut-off limits. We can declare that the system is in some way in a monotone trend or, at least, not randomly jumping around the possible value space. AIC and BIC results are quite stable too and their minima values coincide. It is interesting to see how synchronously and systematic AIC/BIC is responding, which implies the existence of a systematic mechanism that is responsible for these paths generation/execution. Consequently, this supports our assumption of the mechanism behind the generation of inherent non-determinism (Section 2.2.2).

Even though the selection criteria for deciding the cut-off limit value is an open issue up to date, we can see that the test coverage percentage varies significantly depending on the frequency. There is a significant difference for test coverage in having a total of 2500 traces or 4000. One possible option would be to consider the optimal combination of dispersion value (closer to 1) and AIC/BIC minima, as they show alignment. At least, in this case, the optimal cut-off limit would be the lowest frequency (i.e., 2). However, the lowest frequencies may be less accurate for this study, as it is more probable to miss the appearance of these traces. For instance, to illustrate the problem, the paths that were observed 10 times are quite probably complete. It is unlikely that these test-campaigns missed a path that should have occurred 10 times but simply was never observed. Assuming a binomial distribution for the individual path, the probability of observing 0 paths where the expected value is 10 would be $4.5 \cdot 10^{-5}$. Thus, a 99.999% probability to observe such a path. These probabilities are calculated with the recorded data set and the Binomial Probability Mass Function (PMF) presented in Equation 5.6, where p is the probability of success and n represents the number of independent Bernoulli trials.

$$P(X = k) = \binom{n}{k} \cdot p^k (1-p)^{n-k} \quad \text{for } k = 0, 1, 2, \ldots, n. \tag{5.6}$$

For a path with an expected count of 2, we already are down to an 86% probability of observing such a path. For an execution frequency of 3, we obtain 95% probability and with an execution frequency of 5 we obtain 99% probability. Consequently, with a cut-off value of 2, we are likely to miss a significant number of rare paths. Therefore, one cannot draw many conclusions from the pure path coverage without providing a credible frequency estimate. Indeed, this also might be an argument against using small cut-off values for the frequencies. Moreover, as the testing process approaches full coverage the lower frequencies trend to disappear. This means that the few that appear are in the last test-campaigns and, therefore, the assumption that the number of *unique traces* tend to decrease is violated. For instance, if we take traces of frequency one, the majority of them are located in the last campaigns since the traces of the first campaigns had a higher chance of occurring multiple times. This is the reason why frequency cut-off 1 does not appear in Table 5.1, as it does not show a decreasing trend. At the same time, this implies that it is not possible to perform the improper integral at infinity since the area is not finite. Besides, lower cut-offs can also overestimate the total number of traces while they are losing their decreasing trend.

Taking these facts into account we can conclude that the cut-off limit must vary depending on the data set size. We believe that the cut-off estimator must derive from the frequency of frequency shape. In other words, depending on the number of traces that have occurred a defined number of times. When there are a large number of traces that have been executed a few times (e.g., 1, 2, 3) the cut-off will be lower than if there are few traces with that frequency of execution. Additionally, as there is no guarantee that there is a mathematically solid solution for the cut-off estimator, we provide a reasonable empirical approach to finding the most reliable cut-off value.

Section 4.4 uses entropy to empirically categorize traces between common and rare traces. The analysis quantifies the amount of information and, hence, we can divide the data set into two groups that provide the most equivalent amount of information. It that case, the analysis was based on the execution frequencies of the traces. Briefly, the database is formed by a few traces that are executed most of the time and then many traces that are rarely executed. Therefore, the entropy analysis separated these two groups. But in this case, the analysis is based on the frequency of execution frequency. We do not look at how many times a trace has been executed, but at how many traces there are with this execution frequency. Keep in mind that there are many more traces that are executed a few times than those that are executed many times. This can be graphically perceived in Figure 5.1, where there are a larger number of traces with low execution frequencies. Therefore, the entropy analysis with frequency-of-frequency data will result in a small cut-off value.

Consequently, using the following equations it is possible to estimate the cut-off frequency that separates the data set in two. The aim is to obtain two groups with the most similar amount of information. Note that Equations 5.7 and 5.8 use n to represent the maximum r (rate) value identified in the data set.

$$H_1(r) = -\sum_{i=1}^{r} p_i \log_2 p_i \quad \text{and} \quad H_2(r) = -\sum_{i=r+1}^{n} p_i \log_2 p_i \tag{5.7}$$

$$f(r) = |H_1(r) - H_2(r)| \tag{5.8}$$

$$r_{cutoff} = \min(\{f(r_1), \ldots, f(r_n)\}) \tag{5.9}$$

This allows obtaining the frequency or *rate* value (i.e., r_{cutoff}) that results in the minimum difference of the two entropies and, thus, the *rate* value that differentiates *common-traces* and *rare-traces* based on the frequency-of-frequency results. With the current data, the entropy analysis results in a cut-off of 11. Instead, if we had an alternative data set, the results would vary. For instance, if we reduce the current data set by only selecting the system-calls with less number of *unique-traces* the entropy value varies. The *Selected* data set is formed with system-calls *futex(wait), futex(wake), ppoll, read, and recvmsg*. In this case, entropy results in cut-off 18. We also perform the analysis only with *read* system-call. Table 5.2 collects the data obtained for the different data sets. As expected, the cut-off varies depending on the data set. Besides, in all cases, we found that the Poisson model fits correctly (dispersion around 1).

Table 5.2 Results obtained in the different data sets with entropy selection cut-off.

Data set	Cut-off	Recorded traces	Total traces	Test coverage (%)	Dispersion
All	11	2409	2846	84.55	1.18
Selected	18	128	140	91.42	1.37
Read	10	42	44	95.45	1.03

5.4.2 Validation of results

As we have seen in the previous chapter, we need to check the reliability of the proposed method. Although the method employs techniques that provide a certain level of assurance (e.g., KS-test, autocorrelation, CIs), in this section, we perform additional analyses to improve it. On the one hand, we examine the variation of results with smaller data sets with the purpose of contrasting them with larger data sets. The aim is to examine whether the results provided by the presented method remain within a CI even if the size of the data set increases. On the other hand, we examine the proposed method but with a different test-campaign size. In the case of the AEB case study, each test-campaign is formed with 250 iterations of a test-case. Thereby, we perform the same analysis but generating additional test campaigns based on the ones we have. That is, what is the response of the method if we have 500 campaigns of 125 executions instead of 250 of 250 executions?

The first analysis is performed with two different numbers of test campaigns: with 75% of the test campaigns and 90% of them. Indeed, test campaigns are randomly selected and ordered. This is performed repeatedly with the aim of calculating the standard deviation and the mean of all the results. Thus, we do not just get the total number of traces that a specific combination of test-campaigns provides, but the mean of a large number of combinations. That is the reason why the *Total traces* vary with Table 5.2's results, as Table 5.3 collects a mean result of the different data sets that are based on sampling. *Cut-off* frequency value shown in Table 5.3 is also a mean value, as depending on the sampling it may vary.

Table 5.3 Obtained results with different data set sizes.

	75%			**90%**			**100%**		
Data set	**Cut-off**	**Total traces**	**Std Error**	**Cut-off**	**Total traces**	**Std Error**	**Cut-off**	**Total traces**	**Std Error**
All	11	2603.76	4.39	11	2807	4.38	11	2912.68	3.87
Selected	21.62	128.69	0.38	19.36	132.68	0.3	18	134.36	0.25
Read	8.28	44.21	0.29	9.36	44.06	0.18	10	43.93	0.09

Table 5.3 shows the results obtained with 25% and 10% shorter data sets. It can be observed that the results are quite stable even if the size of the data set varies. *Selected* and *Read* data sets depict a variation of the cut-off frequency depending on the data set size. Although the data set size varies, we can consider that the estimated total number of traces is stable. In other words, there is no large variation in the total number of traces even if the data set increases. This means that even if the data set increases, it still follows a Poisson distribution. There are no major fluctuations in the results, so we can conclude that the model responds correctly to the behavior of the system. Additionally, varying the cut-off frequency depending on the data set stabilizes the *Total traces* result, otherwise the variation would be somewhat larger. Note that in the complete data set (*All*) a reduction of the 25% of the test campaign is not enough to vary the cut-off frequency. However, examining the other two data sets we can confirm that it will vary if the data set increases significantly. For example, with a 50% data length the cut-off decreases to 10. Moreover, the standard error tends to decrease as the data set size increases. So the results will become more accurate as the system is exercised.

It is also possible to observe that in the case of data set *All*, the results show a certain increase in the total number of traces. However, we do not consider this increase to be alarming. In addition, looking at the other two data sets we observe that as the data set is increased the estimation becomes more precise.

For the second analysis, we do a similar exercise. We generate twice as many test-campaigns with the same data set. For this purpose, we cut off by half the number of iterations each test-campaign has. In this case, from 250 to 125 iterations. Therefore, we get a total of 500 test campaigns. The analysis is also done randomly sampling and reordering the test-campaigns, but in this case without excluding test-campaigns. Table 5.4 collects the results obtained with 500 test-campaigns. Firstly, as dispersion value is around 1 we can conclude that the sample size of a test-campaign does not alter the Poisson behavior. Secondly, if we compare the obtained results with the ones provided in Table 5.2, we can see that they are similar. Therefore, we can say that varying the sampling size of test-campaigns does not alter the results obtained with the method.

Table 5.4 Estimated total number of traces depending on the cut-off limit.

Cut-off Freq	**Total traces**	**Std Error**	**Dispersion**	**AIC**	**BIC**
11	2888.48	2.87	0.99	1995.26	2003.69

The results shown in the studies increase the confidence that can be placed in the method. The obtained results are stable in the different analyzed scenarios. We have not found any situation where the method results in an out of the expected result. Furthermore, in all scenarios, the data has shown a suitable fit with the Poisson model. Therefore, it seems legitimate to use the method to perform the analysis.

5.5 Summary

This Chapter introduces a method to quantify the uncertainty caused by the unknown kernel paths of a specific application running on Linux. The proposed statistical analysis of recorded paths allows estimating the total number of expected paths and, thus, quantifies the uncertainty. We believe that the method is a step forward to pave the way towards the verification of Linux-based, or for that matter any complex software-intensive safety-related system, where 100% test-coverage is not achievable.

Taking into account the obtained data, the assumption that the behavior of the system follows a Poisson distribution seems reasonable and adequate. Consequently, it is rational to employ Poisson regression for the quantification of the uncertainty and, thus, test coverage measurement. Furthermore, the estimation of a cut-off frequency with an entropy analysis seems a plausible approach as it is a solution that exhibits the correct trends (long tail moves it to smaller frequencies). Moreover, empirical data confirms integration being stable. It is essential to note that while this is no mathematical proof, it is a reasonable empirical approach to finding the most reliable value. Nevertheless, it needs further analysis with distinct data sets and use cases (see Section A).

From a classical safety perspective, the untested paths that we observe here would be translated into a low test-coverage. If we consider 2912 the total number of traces, we get a test coverage of 82.72%. This value is far from full coverage that has been traditionally achieved (or attempted), where a full coverage was feasible controlling the input parameters of the respective test-function. In fact, if we consider the WCS among all cut-off frequencies, we would be at around 51% test coverage. Moreover, current safety standards do not discern between occurrence probabilities of unobserved (untested) paths as traditional safety operates in a *'software is deterministic'* paradigm. Traditionally, path coverage only considers path possibilities as a reference and, hence, path-coverage is simply the number of actually observed paths to the number of possible paths. Thus, a notion of path probability would not make much sense. However, with a non-deterministic system, several paths exist for the same input parameters and they have a different probability of execution. However, given the observed data, it would not be reasonable to claim that in the above example the coverage would be only 82.72% with 10 thousand hours of testing process. The unobserved paths have a low probability of occurring as they depend on a random state of the system. Therefore, we believe that there is a need to complement this result with the execution probabilities of the paths and, consequently, know the risk that entails a system with this coverage. This is analyzed and addressed in Chapter 7.

Consequently, we propose to look at code coverage in the sense of expected revisiting of observed and, thus, tested paths. From the data presented until now, one could observe that we cannot provide this probability of execution of a path without associating an expected frequency with those unobserved paths.

Given the need for probabilities to complement the results of paths' possibilities, we have established results that have led to the method presented in Chapter 7.

Chapter 6

Non-Parametric Statistical Method for Software Execution Test Coverage

As demonstrated in the previous chapter, certain statistical methods can assist in the estimation of code test coverage of next-generation safety-related systems. The present chapter pursues the same goal as the previous one (Chapter 5). However, in this case, the method is based on a non-parametric approach. The presented method is inspired by research in the field of biology and ecology, which examines how to estimate the number of species that exist in a geographical area based on past observations, in order to estimate the total number of traces that can be exercised.

For this purpose, we analyze the common features with the species richness estimation approaches to see if they are suitable for our objective. In addition, an alternative method to the one presented in Chapter 5 enhances the confidence on the results if they provide a certain level of concordance. Therefore, the objective is to have an extra method that verifies the results obtained with the approach based on the causal model.

The method is based on non-parametric statistics, a statistical branch that encompasses methods that make no assumptions about the underlying distribution of the data [HWC13]. Consequently, the method performs a statistical analysis without taking into account the underlying distribution. This chapter collects different non-parametric statistical estimators and compares the results obtained by them.

6.1 Description of the approach

For years, different statisticians and biologists have investigated the estimation of the number of species [Cha84, SvB84]. Estimating the species richness (number of different species) is a complex task as the number of observed species increases with every increment of the sampling effort [Mag07, CMC04]. The research found in the literature is based on the fields of ecology and biology. Therefore, they were not conducted with the intention of using them in technology domains and even less in functional safety. However, the literature collects different non-parametric estimators that could be appropriate for our research [Cha84, SvB84, OKL$^+$07, Oks13]. Consequently, the analysis described in this chapter is based on the species estimation techniques available in the literature with the aim of estimating the total number of execution paths [AGPC$^+$21].

In this case, the traces are identified as the system is exercised (dynamic analysis). Therefore, we do not know all the traces that can be exercised with a reasonable probability. Suppose each different unique trace is considered a *'specie'*. In that case, it is possible to employ the non-parametric species estimators to estimate the number of traces that have a relevant probability of appearing. Therefore, the study is based on one primary assumption:

- **Assumption:** the number of new execution *unique traces* (i.e., different from those that occurred previously) will increase as the number of test iterations increases. If the cumulative number of traces is plotted, it will result in a *species accumulation curve*, where there will be a moment that the curve approaches the asymptote. The asymptote represents the moment when all the possible execution traces have been exercised and, consequently, it represents 100% of test coverage.

Since reaching the asymptote with testing techniques is a potentially not feasible task, the aim is to estimate the total number of traces analyzing the recorded rare-traces. Chao et al. state that events with a lower frequency of occurrence provide more information for species richness [Cha84, CC16]. After all, events that occur commonly provide almost no information about rare events [CC16].

6.2 Trace accumulation

The data set obtained in Chapter 4 allows analyzing the accumulation curve of the executed traces. The obtained data set collects all the hash values of each of the system-calls that have been executed. Thus, the accumulation of new traces can be examined while the number of test-campaigns increases. Accumulation curves show the tendency to encounter new traces as the sample size increases.

Different approaches exist to display the accumulation curve. A common way to illustrate it is with the exact data that we have collected, i.e. the recorded accumulation. Nonetheless, the drawback is that by using this method it is not possible to know the CIs of the curve. For this purpose, an alternative method known as *Random method* exists. The random method allows estimating the accumulation curve based on a sample-based species frequency data set. The method calculates the mean and the standard deviations of the accumulation curve by sub-sampling the data set without replacement over several iterations [KVDS06]. In other words, it estimates a defined number of accumulation curves with a different order of the test-campaign occurrences.

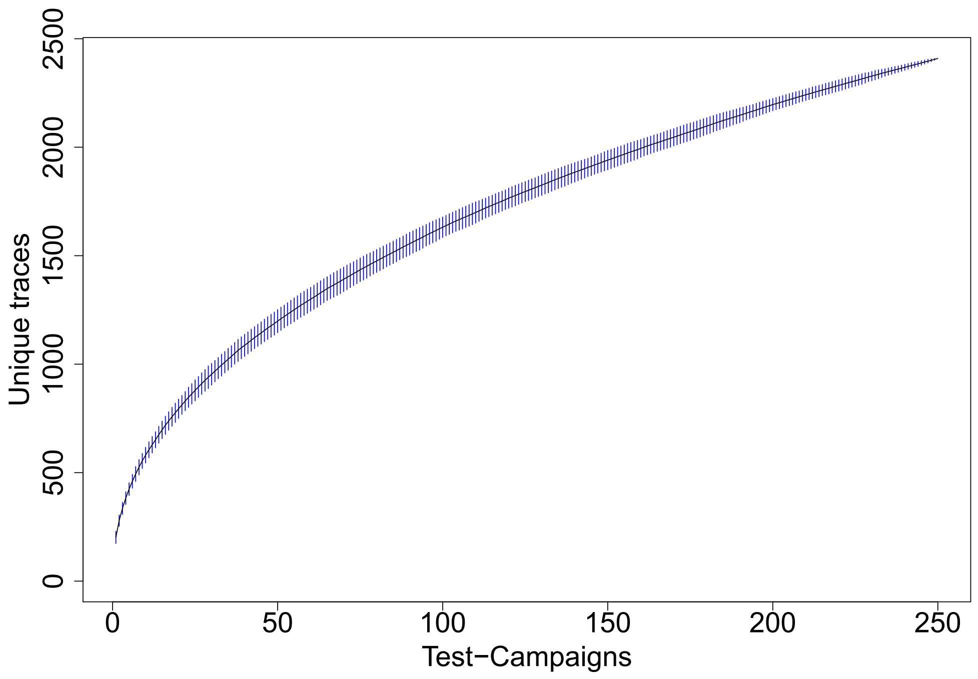


Figure 6.1 Accumulation curve estimated with the random method with 95% CI.

Figure 6.1 depicts the accumulation curve obtained with the *Random method*. It shows trace richness by sampling effort with 95% CI. As can be observed, the increase of the curve is more pronounced at the beginning while, afterward, the tendency is to grow more slowly. However, by a visual inspection of the curve, it is possible to affirm that the data set obtained has not yet reached the asymptote. In other words, if we were to continue collecting data we would still see a significant number of new traces. Once the asymptote is reached, one could say that it is relatively difficult to identify any new traces that remain to be executed.

We believe that even for safety systems we do not need to reach the asymptote. We need to reach a low enough number of unobserved paths that we consider the residual risks tolerable for the safety-related system (Chapter 7).

6.2.1 Merging all system-calls

In this section, we perform a similar analysis to the Section 5.3.3 of Chapter 5. Nevertheless, in this case, the analysis is based on the accumulation curve instead of the Poisson model. Therefore, we examine the MSE of the accumulative curve excluding each of the folds that form the data set. In this way, it is possible to verify that all systems share the same distributional properties and, hence, it is possible to treat it as a single population for the analysis of the non-parametric estimators.

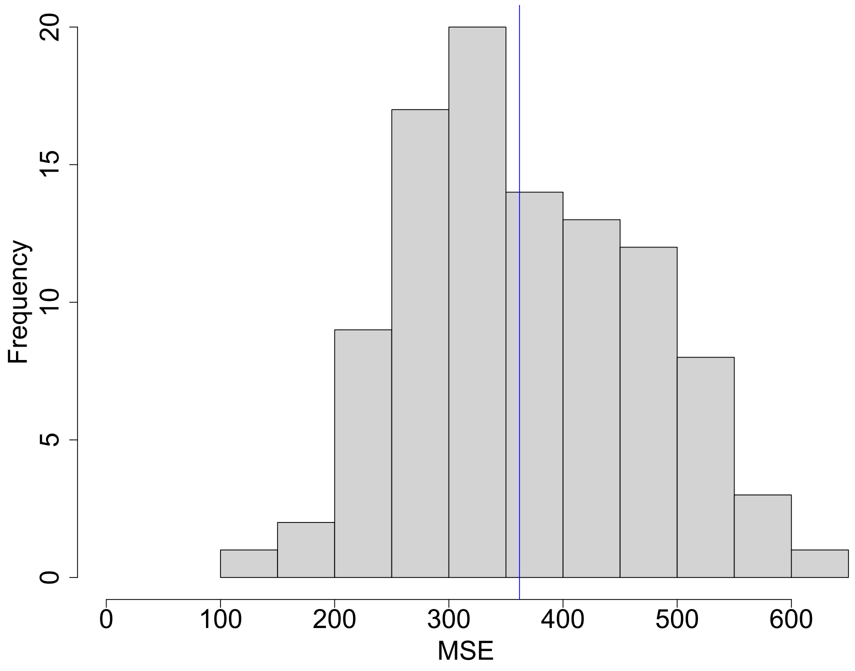


Figure 6.2 Histogram of obtained MSE values on 100 folds. Vertical line represents the mean value.

As Figure 6.2 shows, the results obtained are normally distributed. With the obtained result, it is possible to assess, as in Chapter 5, that test coverage analysis can be performed with all the system-calls together.

6.3 Estimation of the number of paths

The accumulation curve shows the tendency to find new traces while testing iterations continue. Nevertheless, it does not estimate the richness of the traces, i.e., the number of traces that an application can exercise in the kernel. The literature describes several non-parametric estimators [GC01, GC11, OKL$^+$07, Oks13]. The most commonly used estimators can be classified into two types [Bas17]:

- **Incidence-based estimators:** considers the number of test-campaigns in which each *unique trace* has been executed. For instance, how many traces have been executed only in one test-campaign? In two test-campaigns?
- **Abundance-based estimators:** considers the number of times that each *unique trace* has been executed. For instance, how many traces have been executed only once? Twice?

Note that *unique trace* is equivalent to a specie. Figure 6.3 illustrates the classification of several non-parametric estimators that are analyzed through this Chapter.

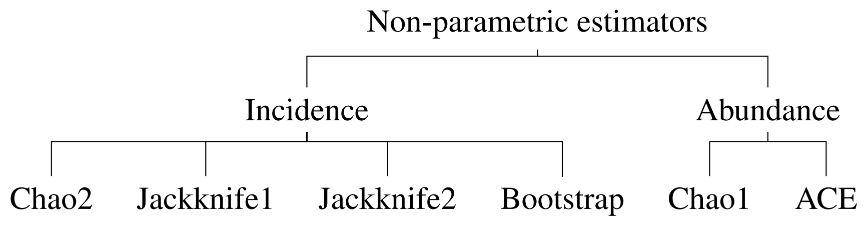


Figure 6.3 Relation among non-parametric estimators.

All the identified estimators (Figure 6.3) employ the less frequent traces (rare-traces) to estimate the richness as the majority of the information is provided by these traces [Cha84].

6.3.1 Incidence-based estimators

Chao2 estimator based on incidence takes into account traces that execute only in one test-campaign (i.e., uniques), the number of traces that occur in two test-campaigns (i.e., duplicates), and the total number of traces recorded among all the recorded test-campaigns [Cha87]. Equation 6.1 provides a bias-corrected form for the case that there are no *duplicates* [OKL$^+$07, GC11].

$$S_P = \begin{cases} S_0 + \dfrac{a_1(a_1-1)}{2(a_2+1)}\dfrac{N-1}{N} & \text{if } a_2 = 0 \\ \\ S_0 + \dfrac{a_1^2}{2a_2}\dfrac{N-1}{N} & \text{if } a_2 > 0 \end{cases} \tag{6.1}$$

- S_P is the total richness of the traces (observed + not-observed).
- S_0 is the number of traces presented in the recorded data set.
- N is the total number of samples.

- a_1 represents the number of traces that are executed only in one test-campaign (i.e., uniques).
- a_2 represents the number of traces that are executed in two test-campaigns (i.e., duplicates).

There are also estimators based on jackknife and bootstrap statistical techniques. Jackknife is a resampling method that recalculates the estimator leaving out subsets from the data set successively and, then, calculates the average value. Bootstrap is a widely used statistical resampling method [JWHT13]. It is a test based on random sampling with replacement. Therefore, it allows generating many *'new'* data sets by randomly sampling with replacement from the original data set.

First order Jackknife estimator is based on the fact that we have lost as many traces as the ones that appear in one test-campaign [BO78, SvB84].

$$S_P = S_0 + \frac{a_1(N-1)}{N} \tag{6.2}$$

Jackknife2 depends on the traces that are executed in one and two test-campaigns (i.e., uniques and duplicates) in the recorded data set [GC11].

$$S_P = S_0 + \frac{a_1(2N-3)}{N} - \frac{a_2(N-2)^2}{N(N-1)} \tag{6.3}$$

Bootstrap is based on the approach that considers that there are the same number of unseen traces as those lost after re-sampling with replacement [Oks13]. p_i execution frequency of trace i.

$$S_P = S_0 + \sum_{n=1}^{S_0} (1-p_i)^N \tag{6.4}$$

6.3.2 Abundance-based estimators

Chao et al. proposed an estimator that is based on the abundance of the species [Cha84]. The estimator analyzes the traces that have been executed once (i.e., singletons) or twice (i.e., doubletons). Equation 6.5 is similar to the bias-corrected Equation 6.1.

$$S_P = S_0 + \frac{a_1(a_1-1)}{2(a_2+1)} \tag{6.5}$$

Note that in this case a_1 and a_2 refer to singletons and doubletons:

- a_1 represents the number of traces that are executed only once (i.e., singletons).

- a_2 represents the number of traces that are executed twice (i.e., doubletons).

The abundance-based Coverage Estimator (ACE) analyzes species richness considering two groups of traces [Oks13, O'h05, GC11]. On the one hand, S_{abund} represents the number of abundant *unique traces*. The common approach is to declare abundant individuals with an occurrence higher than ten. S_{rare} represents the number of *unique traces* that are rare (e.g., Traces executed less than ten times). ACE estimator is described in Equation 6.6. N_{rare} refers to the total number of executions of rare traces.

$$\begin{aligned}
&S_P = S_{abund} + \frac{S_{rare}}{C_ace} + \frac{a_1}{C_{ace}}\gamma^2 \\
&where, \\
&C_{ace} = 1 - \frac{a_1}{N_{rare}} \\
&\gamma^2 = max[\frac{S_{rare}}{C_{ace}} \frac{\sum_{i=1}^{10} i*(i-1)a_i}{N_{rare}(N_{rare}-1)} - 1, 0]
\end{aligned} \tag{6.6}$$

6.4 Estimation of the total number of traces

Using the estimators we have introduced, both abundance and incidence-based, we calculate the number of total traces that are likely to be executed by the safety function. These estimations are performed using the recorded 250 test-campaigns. Table 6.1 provides the richness of results obtained by each estimator and the test coverage percentage.

Table 6.1 Estimated total number of traces and test coverage percentage.

Recorded traces	**Chao2**	**Jackknife 1**	**Jackknife 2**	**Bootstrap**	**Chao1**	**ACE**
2409	3916.45 (61.5%)	3411.97 (70.6%)	4079.93 (59.04%)	2833.95 (85%)	3898.5 (61.79%)	3976.23 (60.58%)

If we inspect Table 6.1, we see that there is no a significant difference between the results. The certain equivalence of the results obtained with estimators that are even based on different approaches (i.e., abundance and incidence) indicate that the approach can be relied upon, at least at this stage of the research. Although further analysis needs to be performed. At this stage of the research, we can state that the total number of traces is in the range of 2833.95 and 4079.93. As a worst-case approach, we could set the total number of traces to 4174, which belongs to the upper CI of Chao2 (see Table 6.2).

Furthermore, the CIs show also similar results along with these estimators. Table 6.2 collects the CI values for 95% CI. It should be noted that the Jackknife2 CIs could not be calculated because its variance equation could not be identified in the literature.

Table 6.2 Estimated total number of traces and the obtained CIs.

Estimator	**2.5%**	**Result**	**97.5%**	**Estimator**	**2.5%**	**Result**	**97.5%**
Chao2	3659	3916	4174	Bootstrap	2767	2834	2901
Jackknife1	3267	3412	3556	Chao1	3642	3899	4155
Jackknife2	NA	4080	NA	ACE	3905	3976	4047

Variances can be considered quite similar between the estimators. Some of them provide wider results, but none of them provide significantly different results. Both Chao1 and Chao2 are the estimators with wider CIs. The most pessimistic result considering the CIs is provided by Chao2 estimator with 4174 total traces, which leads to a test coverage of 57.71%.

6.4.1 Validation of results

A question that arises with this approach is: how do we know that the obtained results are reliable? Species richness estimators have been widely evaluated for species estimations [GC01, GC11, OKL$^+$07, Oks13]. Nevertheless, to our knowledge, they have not been used before to estimate test coverage. J. Béguinot stated that Jackknife 2 is usually the most suitable non-parametric estimator for species richness calculation [Bé16]. In addition, the author argued that the Chao1 estimator is also appropriate when the data set is close to the sampling completeness. In fact, one of the main issues with non-parametric estimators identified in the literature is the sampling size [CCLG09, HHRB01, CC94, Mel04]. Abundance estimators underestimate the richness with low sample sizes [HHRB01]. Melo concluded in a study that non-parametric estimators are reliable once the number of *rare-traces* starts decreasing [Mel04]. This implies enlarging the data set until the number of rare-traces becoming *common-traces* is higher than the new traces that appear.

Even though all these researches were focused on species estimations and not on software execution traces, both scenarios have some common features. Thereby, taking into account the sampling size issues identified in the literature, we believe it is interesting to examine the estimators with a smaller data set. For instance, which is the estimated value that we would have obtained if we only recorded 150 test campaigns? And with 200? This allows examining if there are significant differences between the results and the variances. Table 6.3 collects the estimated value for each estimator with different data set sizes. In fact, with

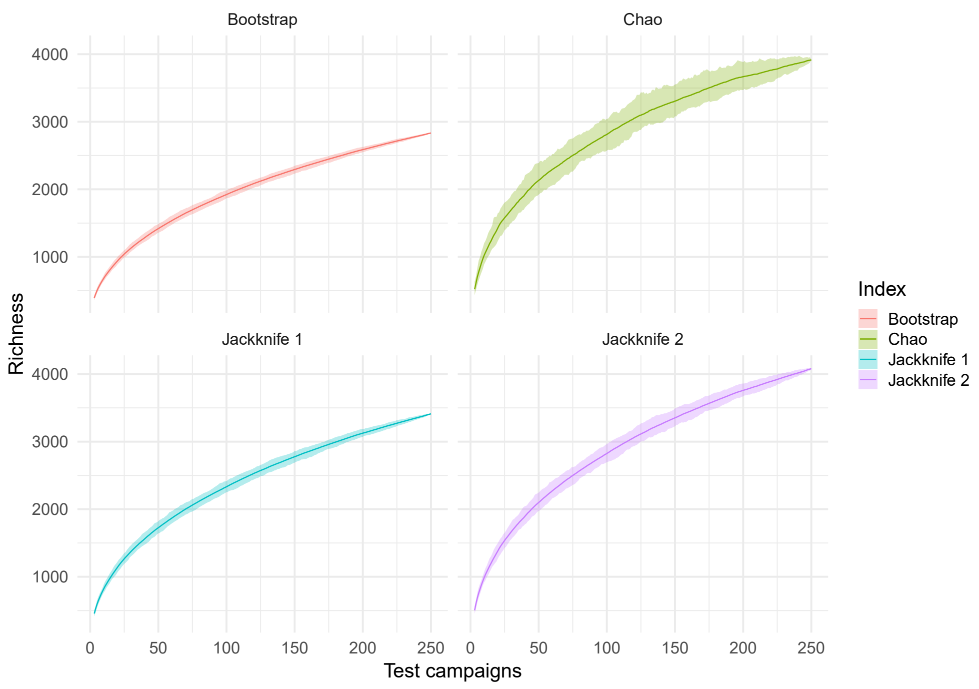
150, 200, and 225 test-campaigns. Table 6.3 also collects the standard error values for each estimator, which can be used to calculate the CI values.

Table 6.3 Estimated total number of traces for different data set sizes.

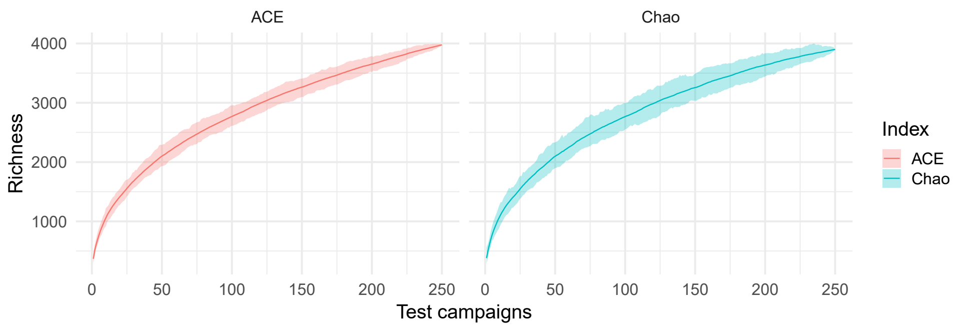
	150		**200**		**225**	
	Result	**Std. Error**	**Result**	**Std. Error**	**Result**	**Std. Error**
Observed	1949	NA	2211	NA	2332	NA
Chao2	3126.19	113.32	3713.95	136.1	3873.42	135.92
Jack1	2762.54	75.33	3158.24	76.26	3324.57	75.93
Jack2	3293.26	NA	3805.21	NA	3995.98	NA
Boot	2295.64	34.81	2609.7	34.72	2750.85	34.71
Chao1	3118.58	113.67	3692.47	135.09	3856.27	135.37
ACE	3206.4	33.09	3750.06	35.62	3933.52	36.55

The results in Table 6.3 confirm what Hughes et al. declare, the estimated values *"strongly correlate with sample size"* when the sample size is not large enough [HHRB01]. The estimated value increases while the data set gets larger, i.e., more test-campaigns are executed. This can also be analyzed by resampling the data set. We calculate the richness value with each test campaign's addition, but by resampling the whole data set several times and not using the variance equations. For example, we calculate one hundred times the estimated value with 50 test-campaigns, but each of the times the test-campaigns are randomly selected and ordered. This is performed for several data set lengths and, hence, we can see how the value varies. The results obtained with these techniques are illustrated in Figure 6.4. It shows the extrapolated richness indices estimated with the four abundance-based estimators and both incidence-based estimators. These plots, unlike Figure 6.1, extrapolate the accumulation and, hence, it estimates the accumulation curve including the not-observed *unique traces*. Besides, it provides the results with the 95% CI, which are calculated by resampling the whole data set.

Figure 6.4 depicts a certain concordance between the results of some estimators and a significant difference with other estimators. Nevertheless, all estimators show a clear increasing trend in the total value of *unique traces* as the system is further exercised. With a larger data set, the estimators provide a larger number of traces that have a relevant probability of being executed. Besides, the CIs of reduced data sets do not preview this estimation increment. If enlarging the data set size causes the estimated total number of paths to also increase out of the CIs of previous estimations, it means that the obtained results cannot be relied upon as the total number of executable paths if the system would be infinitely tested/executed.



(a) Incidence-based estimators



(b) Abundance-based estimators

Figure 6.4 Extrapolated richness indices estimations with the 95% CI.

If we analyze the test coverage percentage while the data set size enlarges, we obtain the results collected by Table 6.4. The results show a quite stable coverage while the data set gets larger. However, this also means that the coverage value is not precise enough with a reduced data set, as the percentage is overestimated taking into account the results of the total number of paths obtained with larger data sets. Thereby, in this case, the test coverage percentage value obtained should be considered: test coverage lower than X (i.e., test coverage < X %).

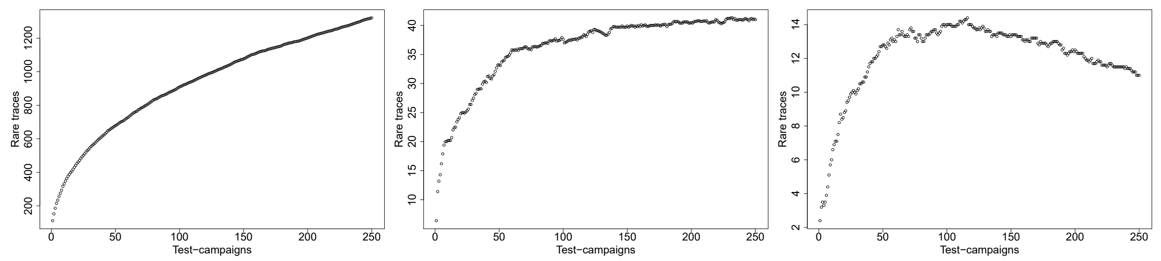
It seems that we do not have enough samples to perform an accurate analysis. The coverage rate is fairly constant, even though a larger number of traces have been tested. This is due to the fact that the number of traces that are estimated is also higher. If we look at the

Table 6.4 Test coverage depending on the estimator and for different data set sizes.

Test-campaigns	**Chao2**	**Jackknife 1**	**Jackknife 2**	**Bootstrap**	**Chao1**	**ACE**
50	54.9	69.47	59.09	84.28	59.32	56.59
100	58.15	69.06	56.26	85.05	60.51	58.5
150	57.93	70.41	57.24	84.59	62.49	60.78
200	58.9	70.69	57.6	84.78	59.87	58.95
250	61.5	70.6	59.04	85	61.79	60.58

equations of the estimators, we see that the reason for this is that the existence of rare traces does not decrease as more traces are tested. Besides, the literature states that the estimators are not accurate with an upward trend of the number of rare-traces [Mel04]. Thus, we inspect the number of *rare-traces* through the data set size increment.

Figure 6.5a provides the number of rare traces while the data set increases. The plot depicts the tendency for the number of rare traces to increase as the system is further tested. This means that more rare traces are collected (i.e., frequency ≤ 2) than those that are executed more times and can be considered *common-traces* (i.e., frequency > 2). But, does this occur with all the system-calls? It may happen that a system-call has a higher test coverage than others. From Chapter 4, we know that each system-call has a different variability. Each system-call can exercise a different number of traces, with a distinct execution probability. Therefore, it seems possible to have system-calls with a downward trend of the number of rare-traces. The *rare-traces* trend is examined with three different data sets: 6.5a shows the results of the whole data set with all system-calls, 6.5b only collects a selected number of system-call (i.e., *futex(wait), futex(wake), ppoll, read, and recvmsg*), and 6.5c shows the obtained graph only with system-call read. Note that the plots from Figure 6.5 are estimated with a random sampling of the test-campaigns. We calculate a high number of these curves with different test-campaigns' orders and after plot the mean curve.

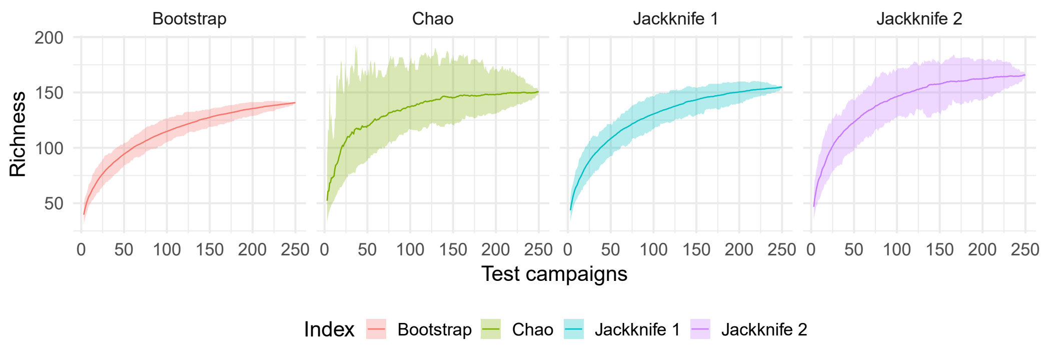


(a) All system-calls. (b) Selected system-calls. (c) Read system-call

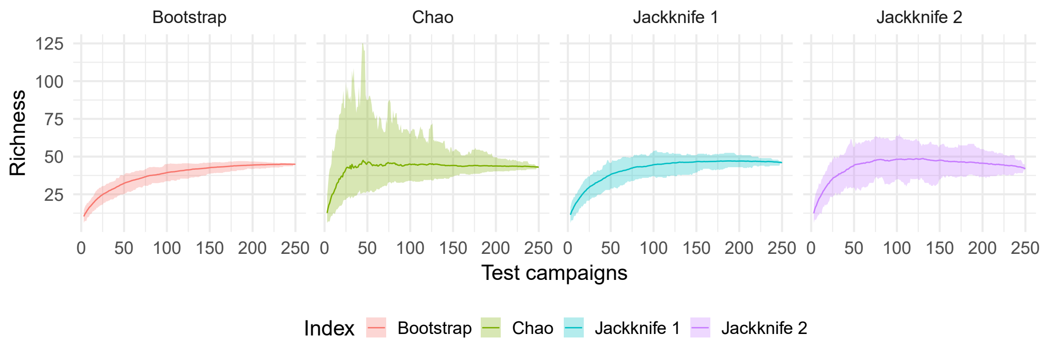
Figure 6.5 Number of rare-traces (≤ 2) through data set size increment.

The data set composed only by a reduced number of system-calls (6.5b) shows a stable number of rare-traces. These system-calls correspond to half of the system-calls with the lowest number of traces observed. It is essential to note that this does not imply that the number of *rare-traces* should be less, as traces have different execution probabilities. The data set that corresponds to system-call read (Figure 6.5c) depicts a clear downward trend of rare-traces. Approximately after test-campaign 125, the number of *rare-traces* starts decreasing. Consequently, it is interesting to check how estimators behave with these reduced data sets and whether they are reliable with them. Extrapolating a monotone increasing development of rare-traces could seems to err on the safe side - while very extensive campaigns could result in decreasing rare paths the estimates would imply a reliably higher value to take into account when establishing residual risk.

If it is feasible to reach significantly more reliably and more advantageous estimates by extending the campaigns beyond the inflection point (roughly around 125 campaigns in the case of *read()* - see Figure 6.5c) is possibly economically relevant future work.



(a) Selected system-calls.



(b) Read system-call.

Figure 6.6 Richness estimators with selected system-calls.

Figure 6.6 shows the estimated results as the data set increases. On the one hand, Figure 6.6a shows the results for the data set formed with the selected system-calls while, on the other hand, Figure 6.6b corresponds to the read system-call. Figure 6.6a shows certain stability in the results. In other words, the estimation curves do not indicate that the estimated value will increase significantly if more test campaigns are collected. Although some estimators still show a trend to slightly increase their estimation, there are also estimators that show stability in their results. For instance, Chao1 estimator maintains the estimated results stable after test-campaign 200. The stability of the results is clearer in Figure 6.6b. The estimated values for system-call read reach the asymptote in all estimators. Moreover, there are some of the estimators that reduce their estimation value at the last test-campaigns. If we check the equations of the estimators, we can see that the obtained reduction in the estimated value makes sense. After all, as Figure 6.5c shows, we have fewer rare traces. Note that abundance estimators have not been plotted but provide equivalent results to incidence estimators.

Taking into account the presented data, we can provide the interpretation of the estimators for test coverage calculation. From what we have seen and analyzed, the estimators are not reliable with data sets where the number of rare traces is still increasing. Thus, in these cases, these coverage estimates should be taken as rough references and not as precise coverage values. Moreover, each estimator offers a different value and, at present, we have no method to know which one is the most appropriate for test coverage. The values of test coverage become more precise as the size of the data set increases. Nevertheless, it is necessary to test the system sufficiently to see that the number of rare traces (i.e., frequency ≤ 2 2) is downward. Besides, if these estimators provide a result that remains stable while the data set enlarges, we can conclude that this is statistically the maximum number of traces that can be obtained as the system-context is stable.

It is also essential to mention that the estimators may provide an overestimated test coverage percentage with a not large enough data set, but they would not state a full test coverage as long as rare traces exist. Therefore, even if they are not fully reliable with no large enough data sets, estimators will not declare full coverage. Moreover, the presented data indicates that the estimators are more accurate when the testing effort is closer to full coverage.

Table 6.5 collects the obtained test coverage percentages for the three data sets. As it can be observed, in the first two cases the percentage is quite stable despite the fact that more paths have been tested. This is because the estimated value also increases. But in the last case, the one representing the system-call read, the test coverage increases considerably as the size of the data set increases. Therefore, we come to the same conclusion, the test coverage value obtained has to be taken as a rough reference. At least, until near 100% coverage is achieved.

Table 6.5 Estimated total number of traces and percentage of unseen traces.

All system-calls

Test-campaigns	**Chao2**	**Jackknife 1**	**Jackknife 2**	**Bootstrap**	**Chao1**	**ACE**
150	57.93	70.41	57.24	84.59	62.49	60.78
200	58.9	70.69	57.6	84.78	59.87	58.95
250	61.5	70.6	59.04	85	61.79	60.58

Selected system-calls

Test-campaigns	**Chao2**	**Jackknife 1**	**Jackknife 2**	**Bootstrap**	**Chao1**	**ACE**
150	84.09	81.91	76.16	90.66	86.74	84.88
200	81.14	81.71	74.63	90.64	84.85	83.6
250	84.94	82.63	77.17	90.90	87.88	84.81

Read system-call

Test-campaigns	**Chao2**	**Jackknife 1**	**Jackknife 2**	**Bootstrap**	**Chao1**	**ACE**
150	82.53	80.95	74.64	90	87.15	85.16
200	75.32	92.08	73.34	90.74	85.41	85.95
250	97.69	91.34	99.9	93.47	99.22	96.5

Furthermore, Figures that show the trend of the estimators (such as Figure 6.4) allow us to get an idea of how much these estimates may increase. Therefore, this indicates that we can take them as a rough reference to get an idea of how many traces may appear.

6.5 Summary

The present chapter describes a method that estimates the number of executable paths that an application can exercise in the Linux kernel. By keeping track of those traces that have been tested and estimating those that remain untested, the test coverage can be calculated.

Test coverage estimation by non-parametric analysis seems an interesting alternative to traditional techniques. Although the accumulation curve shows the large number of different paths that an application such as AEB can exercise, it also exhibits a trend to achieve the asymptote. Consequently, we are able to estimate the total number of paths that have a relevant probability of occurrence and the number of paths that have not been covered.

One of the main issues with this type of estimators is the data set length. Having a too short data set or a reduced number of test-campaigns may entail an underestimation of the total number of traces. Considering the case study that has been analyzed, an automotive use case involves a large number of hours of testing before the number of rare traces starts to reduce. A question that remains open here is, is it feasible to exercise the system that much? Remember that our research level use case has been exercised for ten thousand hours. Contrary to biology, where obtaining new samples is generally not that easy, in the software case, it is potentially easier. Consequently, a significantly higher assurance should be doable by exercising the system continuously and using this approach during the intensive test campaigns used to validate safety-related systems. In the current era of data, it is possible to have the resources to exercise a large number of systems and, hence, obtain a larger data set to provide more precise results.

Non-parametric estimators may also show a significant variability between their results. Thus, we need to provide further analysis to know which one is optimal for test coverage estimation. With the data we have today, this is not feasible to conclude which is the estimator that needs to be used. Therefore, it is necessary to continue investigating which estimator fits more appropriately to this type of system. For this purpose, we should examine the estimators with other use cases and with other system-context. For instance, we believe it would be interesting to investigate the estimators' variability with the same application but different system-context, for instance, different CPU loads.

As a result, with all the information we have gathered, we believe that the non-parametric estimators are interesting to use as a reference or verification technique. The analysis provided in this chapter shows that the results need to be accompanied by: (i) the standard error, (ii) plots of the results as the size of the database increases, and (iii) tendency of the number of rare traces while data set gets larger. With all these data, we can improve the credibility of the estimators as rough references of test coverage.

Finally, the data presented in this chapter shows the complexity of achieving full coverage. It is observed that there are traces with a very low probability of being executed and that makes it very difficult to test them since it is not feasible to force their execution. However, test coverage is a measurement of possibilities and not probabilities. That is, it quantifies the number of traces that can be run and not the probability that they will be run. Therefore, we believe that these measurements would be more complete if accompanied by the probability of execution. With this objective in mind, we offer the analysis presented in Chapter 7.

Chapter 7

Estimation of Code Execution Path Uncertainty for Test Coverage

The non-determinism that characterizes next-generation autonomous safety-related systems hinders achieving full coverage during testing. The existence of untested traces that can be executed leads to the presence of uncertainty. However, we do not know how to translate the test coverage measure into failure rates and, hence, we do not know the risk of deploying a safety-related system with potentially untested paths.

In this chapter, we conduct different analyses in order to be able to estimate the execution probability of untested paths and, therefore, be able to estimate the risk entailed by the untested execution traces.

7.1 Description of the approach

This chapter describes the proposed method to estimate the Linux kernel execution path uncertainty during the testing phase of a Linux-based safety-related system, leading to the probability (estimation) of unseen execution traces which, following the ALARP principle, shall be minimized. Specifically, if coverage is smaller than 100%, a justification is called for and here is where the present method plays a key role. The method is executed in five consecutive steps:

1. Fitting the data to an appropriate model.
2. Estimating the probability of execution of unobserved events (P_{zero}) and known events with Simple Good-Turing equations.
3. Evaluate the obtained results analyzing the CI of the results.
4. Estimate the risk entailed to the examined system.

It is essential to note that the present method is independent of the results obtained with previous methods. The method complements the test coverage results but does not use the estimated total number of traces. In this way, we do not inherit possible estimation errors that we could have had in the previous methods and, therefore, we improve the justification from a functional safety point of view.

7.2 Fitting the data set

In order to make further statistical analysis, it is necessary to preprocess the data to get an adequate model of the recorded data set. As the aim is to estimate the probability of executions we need to work with execution frequencies. Frequency distribution models permit to represent the recorded system-call occurrences and, thus, describe the obtained data set. If we want to know which model fits the frequency distribution, we first need to understand how the data set is formed.

7.2.1 Understanding our frequency distribution

As Linux is developed with performance in mind, *common-traces* are expected to offer higher performance than traces that are rarely executed. As a general rule, a short trace offers higher performance than a long one. Furthermore, previous chapters show that when a system-call is called, the vast majority of times the same trace is executed, which we refer to as *common-trace* [OSA, AMGP$^+$21b]. However, there are times when a variation of this trace is executed. These traces are known as *rare-traces* and are executed less often.

As a result, the collected traces that a system-call executes can be classified into two groups: *common-traces* and *rare-traces*. If *rare-traces* are inspected, it is possible to observe that they are variations of the most *common-traces* (more detail in Section 3.2.2). They are mainly built by having the most common-trace as trunk and with a series of calls executed at different points of the trace due to variations of the global state of the system The variations are not under the control of the application. Although the execution probabilities of these variations are lower than the probability of the common-trace, each of the variations has a different execution probability. Figure 7.1 illustrates an example of a *common* and a *rare-trace* of system-call write. The beginning of the traces is the same; however, there is a point where the rare-trace executes a branch that starts with *rt_spin_lock_slowunlock()*. After the branch, both traces continue executing the same function sequence until the end. Furthermore, as Table 4.1 shows (Chapter 4), there are a few system-calls, such as *read()*, where their common path is not so commonly executed. However, this is due to the existence of several common paths. For example, in *read* system-calls the second most common path is executed 39.51% of the time.

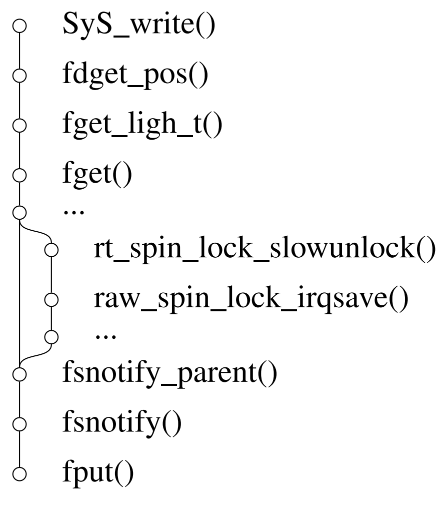


Figure 7.1 Execution trace example (system-call write) with two possible paths: *common-trace* (short) and *rare-trace* (branched).

7.2.2 Justification of Power-law

Power-law distributions are commonly used to model frequency distribution where the frequency of an event correlates with the size of the event [WEG08]. This assumption is expected to be valid in our case (see Figure7.1), but we need to demonstrate that a path formed by a reduced number of functions is more common than a path with a greater number of functions. For this purpose, we analyze the difference in the number of lines that a trace has according to its frequency of execution and we observe that, in the majority of the system-calls, the mean of lines in *common-traces* tends to be less than in *rare-traces*. For this analysis, firstly, it is necessary to perform the differentiation of *common* and *rare-traces*. Entropy analysis allows us to know the amount of information provided by each data set (see Section 4.4.2). Therefore, it is feasible to separate the data set into two parts that provide the most similar amount of information possible.

The box-plot, in Figure 7.2, depicts the number of functions (or LOC) that are executed in *common-traces* vs in *rare-traces*. As it can be seen, in the *rare-trace* group, differentiated in blue and with (r) suffix in the horizontal axis labels, both the average and the quartiles tend to be higher than in the *common-trace* group.

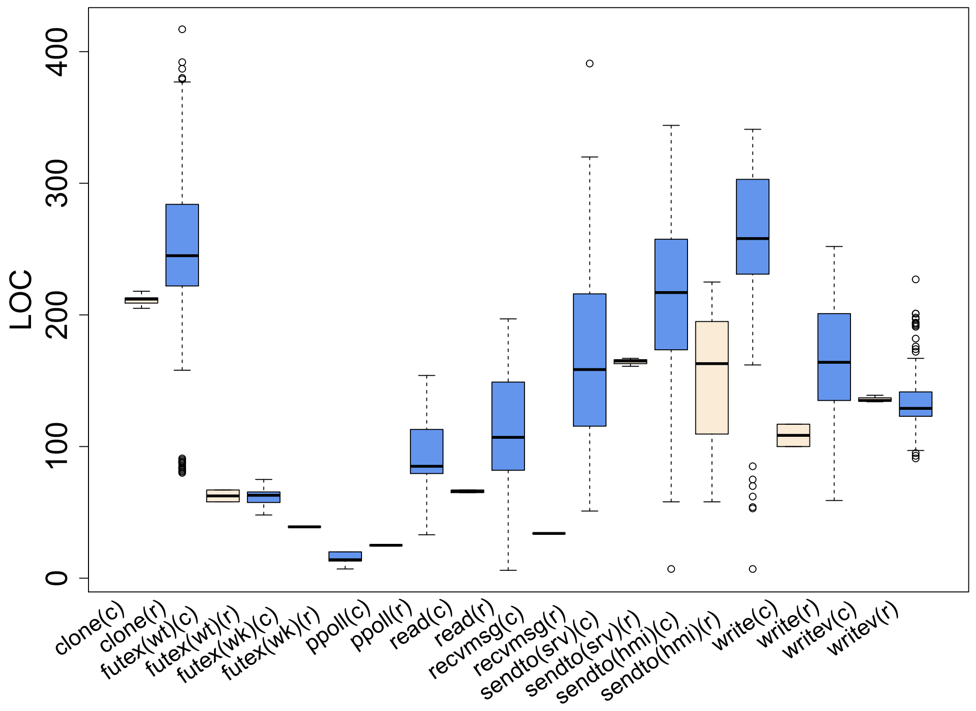


Figure 7.2 Execution trace length depending on common (c) or rare (r) trace.

Figure 7.2 validates the argument of employing the Power-law model in the vast majority of cases. Thus, as preliminary research, we believe it is valid to use this model. However, it also illustrates a small number of corner cases. For instance, the rare-paths of system-calls *futex(wake)* or *writev* show a slightly lower number than the common ones. Furthermore, there are some outliers too in some system-calls. If we manually check these outliers (e.g., read), we see that these traces belong to failed executions. Thus, we can consider them *'valid outliers'* as they have occurred, but they do not break with the assumption we have described.

7.2.3 Merging all system-calls

In order to get a single P_{zero}, it is advantageous to improve the data quality by merging all system-calls in the same model. However, as in Sections 5.3.3 and 6.2.1, it should be qualitatively proven that the system-calls follow an equivalent path development with the model to be used. In this case, we perform the K-Fold cross-validation analysis with the Power-law model. The MSE results we have obtained are shown in Figure 7.3.

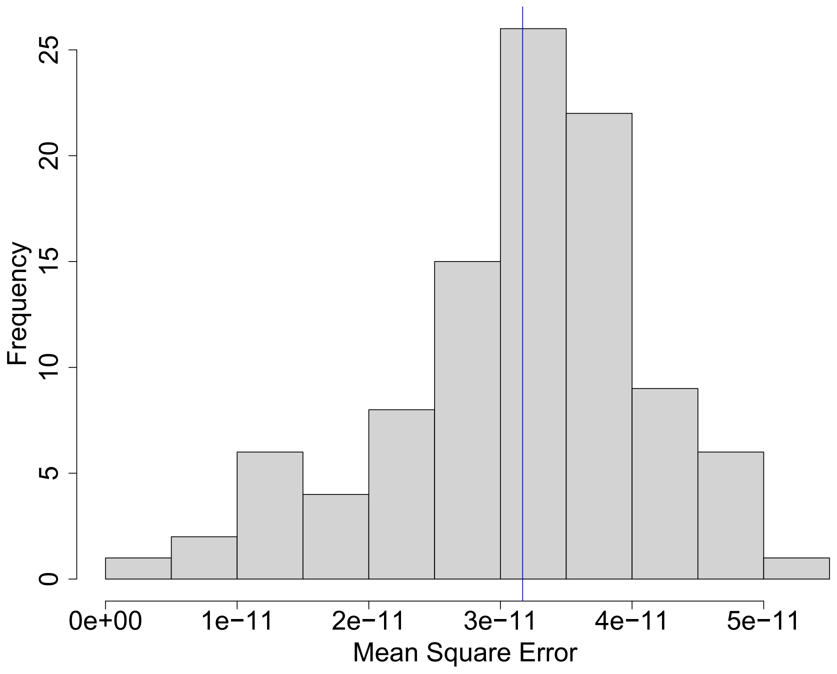


Figure 7.3 Histogram of obtained MSE values on 100 folds. Vertical line represents the mean value.

Figure 7.3 shows that the obtained MSEs are normally distributed and, hence, it assesses continuing the research with the whole data set together.

7.2.4 Modeling with Power-law

The reported results can be accurately modeled by fitting the count of rate r (N_r) with the power-law function $F(r) = ar^b$. The frequency distribution model of the data set is represented with a blue color line in Figure 7.4. The plot only shows the smallest occurrence rates (i.e., r) as the model trends to zero. So the vast majority of the information is in the preliminary ratios. If you examine Figure 7.4, it is possible to verify that the estimated model and the collected data are significantly similar. Therefore, by visual inspection, it can be stated that we have a suitable model to continue the analysis.

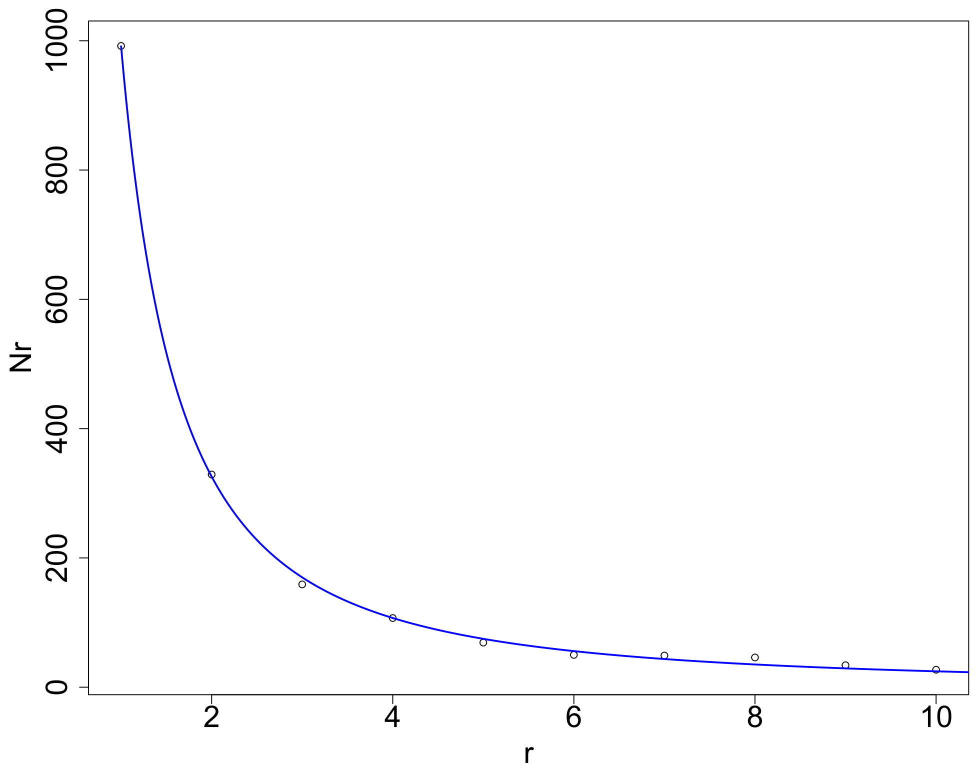


Figure 7.4 Frequency distribution of all recorded system-calls (black points) and the estimated power-law model with function $F(r) = ar^b$ (blue line).

Additionally, we can check the recorded data set in a log-log scale. Power-law is described with an exponential equation $F(r) = ar^b$ and, therefore, the cumulative distribution should follow a straight line on a log-log plot [New17]. The Cumulative Distribution Function (CDF) allows visually inspecting if the data set follows a Power-law distribution. Figure 7.5 collects the data set and the estimated power-law model, having as horizontal axis the frequencies of occurrence values and vertical axis the cumulative probability. By a visual inspection of Figure 7.5, it is possible to identify the straight line that forms the data. Besides, this is even more remarked with a blue line that represents an approximative power-law model. Consequently, the CDF plot shows that the data set formed by all recorded system-calls fits a power-law distribution.

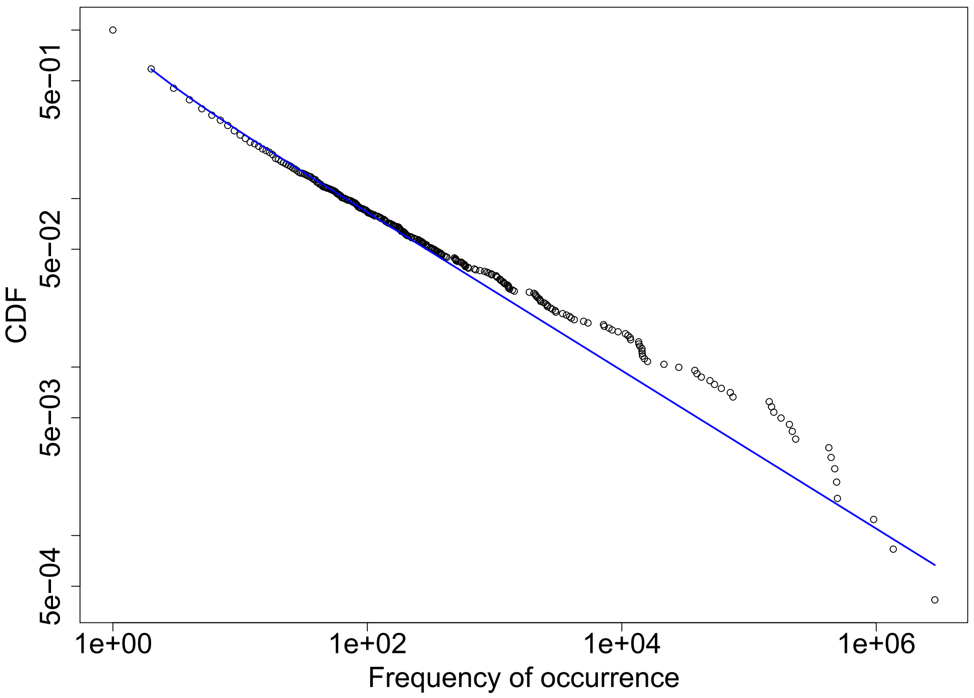


Figure 7.5 Data CDF of recorded data set (black points) with the fitted power-law (blue line).

7.3 Probability estimation

Good-Turing statistical technique allows estimating the probability of previously not executed traces [Sam02]. Good-Turing groups the data set by the rate occurrence (represented by r in Equation 7.1). Thus, for instance, r equal to 1 collects the traces that have only been executed once, and r equal 20 the traces that have been executed 20 times.

$$P_r = \begin{cases} \dfrac{N_1}{N} & \text{for } r = 0 \\ \\ (r+1) \cdot \dfrac{N_{r+1}}{N \cdot N_r} & \text{for } r \geq 1 \end{cases} \tag{7.1}$$

- r: defines the occurrence rate of the traces.
- N_r: defines the number of traces that have been executed with rate r.
- N_{r+1}: represents the count of rate $r+1$.
- N: total number of recorded traces.
- P_r: probability of traces with rate r.

The literature review permits identifying the downside of Good-Turing, which is not reliable with high rates [Vek]. The collected data set shows substantial large rates. For example, the data set has rates of 954385, 1360587, and 2886800. In these cases, the usual approach is to divide the data set to perform the analysis, on the one hand, with Good-Turing for lower rates and, on the other hand, with Maximum Likelihood Estimation (MLE) for higher rates.

- MLE is one of the simplest and most used methods to estimate the probability of events (P_{MLE}). However, this method only works when all the possible events are known, and as result, the probability of unknown events is considered equal to zero. However, for the current scenario, we know that there are several non-recorded traces that may be executed. Smoothing techniques permit adjusting the probability obtained with MLE to achieve a more accurate probability taking into account the unknown events [HM13]. For instance, Laplace estimator or additive smoothing pretends that all events have occurred at least once and, as result, the probability is no longer zero [Vek]. However, this technique gives only a small probability to each unseen event and it may be problematic when there are a huge number of unseen events.

- Good-Turing statistical technique includes unknown events on the probabilities estimations [Sam02]. The technique is based on pretending that an event with rate *r* occurs *r** times. Equation 7.2 shows the calculation of *r**, where N_r represents the counts of the events with rate r. Equation 7.3 calculates the probability for all *r** value higher than one.

$$r^* = (r+1)\frac{N_{r+1}}{N_r} \tag{7.2}$$

$$P_r = \frac{r^*}{N} \tag{7.3}$$

This classification is made by means of a Shannon entropy analysis as explained in Chapter 4. However, in this case, instead of dividing the data set formed by path execution frequencies, the data set is formed by frequencies-of-frequencies as explained in Chapter 5. Thus, we use Equations 5.7, 5.8 and 5.9 to estimate the cut-off rate. As the data and the approach is the same to the Chapter 5, it results in the same cut-off frequency.

Figure 7.6 illustrates the recorded data set results with the estimated model and the categorization of the two groups. The categorization is indicated with a dashed vertical red line. Hence, the probabilities of the ratios located on the left side are calculated with Simple

Good-Turing, while the ones situated on the right with MLE. Note that both axes of the plot are in logarithmic scale and that is the reason why the fitted model appears as a straight line.

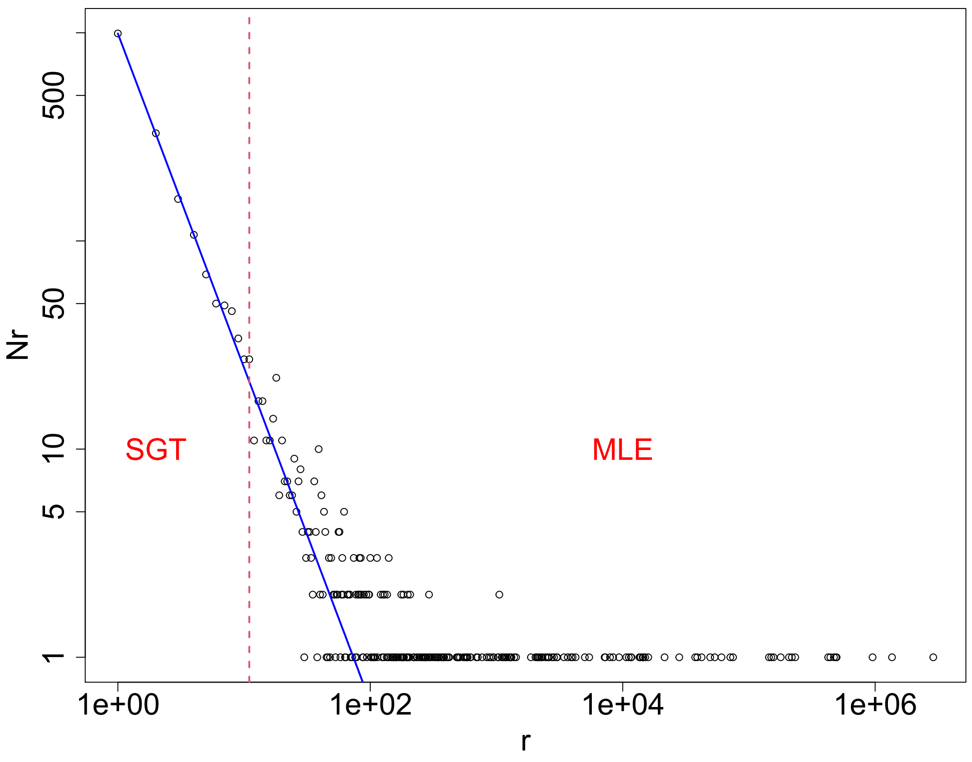


Figure 7.6 Power-law model with the categorization of common and rare based on freq-freq data.

The probabilities estimated with Good-Turing and MLE statistical techniques are gathered in Table 7.1. The results show the probability of execution of paths depending on the execution rate. Accordingly, P_{zero} (i.e., the probability of execution of untested kernel paths) is represented with rate 0. It is necessary to declare that the probabilities have been renormalized to obtain a sum of one. This has been done without decreasing the value of P_{zero}, i.e., only renormalizing the probabilities rates greater than zero. The renormalization is performed based on the weight of P_{zero} and the rate of each probability. For further information about the renormalization see: [Vek].

Although there are no pre-established criteria in the state-of-the-art to determine whether the results are adequate enough to consider the system safe, such a low value for P_{zero} seems promising. Examining Table 7.1 is possible to identify a significant difference between P_{zero} and P_1 or P_2. These data indicate that the Simple Good-Turing technique offers a rather high and, therefore, pessimistic probability. Although this may seem a negative sign of the approach, it is interesting for the functional safety domain as it could be considered the

Table 7.1 Execution probability results for each rate.

rate	**probability**	**rate**	**probability**
0	$1.028366 \cdot 10^{-4}$	8	$7.730282 \cdot 10^{-6}$
1	$6.813320 \cdot 10^{-8}$	9	$8.936446 \cdot 10^{-6}$
2	$1.621718 \cdot 10^{-7}$	10	$1.231514 \cdot 10^{-6}$
3	$2.609863 \cdot 10^{-7}$	...	...
4	$3.633041 \cdot 10^{-7}$	...	...
5	$4.644092 \cdot 10^{-7}$	954385	$9.893420 \cdot 10^{-2}$
6	$5.701453 \cdot 10^{-7}$	1360587	$1.410422 \cdot 10^{-1}$
7	$6.596722 \cdot 10^{-6}$	2886800	$2.992537 \cdot 10^{-1}$

WCS. It is true, however, that in order to make this kind of statement, the research should be analyzed in more detail.

7.3.1 Confidence Interval (CI)

A common approach for CI estimation is to perform bootstrap with re-sampling. However, in the case of Good-Turing this is not potentially possible as re-sampling on the actual data set distorts the frequency-of-frequency distribution. The reason is that too many of the low-frequency entries get re-sampled and are, thus, dramatically distorted while the high-frequency entries do not change much. The low-frequency entries get underestimated and the tail is shortened, resulting in an underestimation of unseen paths.

The alternative way to do this is to sample a subset of the complete data set, which means we have a smaller sample but without re-sampling, and thus a truthful (undistorted) sample from the real generative mechanism operating. In this case, we randomly sample the 90% of the data set a thousand times and calculate CIs with the obtained thousand results.

As it can be seen in Table 7.2, the lower CI is higher than the previously estimated value (Table 7.1). As aforementioned, the CI is calculated with a smaller data set (90%) than the estimated, calculated with the whole data set. As the data set grows, the number of known traces increases, and therefore, the estimated value of P_{zero} is reduced. This is a positive result, since it means that as the number of tests increases the probability of executing an untested trace decreases. Furthermore, the upper CI provides promising results although at the current state of the research we cannot assert if this can be considered adequate enough for a safety-related system.

Table 7.2 Confidence interval for untested execution probability.

Confidence Interval	**2.5%**	**Estimated**	**97.5%**
Estimated Probability	$1.079568 \cdot 10^{-4}$	$1.028366 \cdot 10^{-4}$	$1.119320 \cdot 10^{-4}$

7.3.2 Probability decrease with testing effort

The adequacy of the Good-Turing approach can be confirmed by comparing the estimated results (Simple Good-Turing) with the probability of the new traces that exist in the remainder of the data set (MLE). For instance, in the 100 test-campaign, we calculate the probability of unseen traces with Simple Good-Turing and the MLE probability of the traces that appear between the test-campaign 100 and 150. As we know the traces that appear in the remaining data set and the total number of traces, we calculate the probability using MLE. The results are shown in Figure 7.7.

It can be seen how the estimated probability decreases as the number of executions increases in Figure 7.7 (blue line). Therefore, the probability of executing untested traces decreases by gaining knowledge of the system. In the event that the probability is too high for a certain system, this probability can be decreased by continuing the analysis. Besides, we can confirm that the Good-Turing approach provides stable results after 100-200k executions. The graph also shows the real probability, that is, the one we calculate knowing the remaining data set. Note this probability is a rough estimation, as it does not take into account unknown traces, only the traces that appeared in the following test-campaigns.

7.4 As Low As Reasonably Practicable (ALARP) principle

Now the question that arises is if this probability is low enough for a SIL 2 system. Current safety standards do not provide any reference value to determine whether this probability is within a tolerable risk. However, the standards do provide reference ranges for dangerous hardware failures (i.e., PFH). In contrast to software, the standard considers the existence of random failures in hardware and, hence, requires compliance with the PFH ranges. But as observed in these research activities, the stochastic nature of the untested path is not in its code-sequence but in the occurrence of its activation. It is not possible to deterministically create a system state that would deterministically enter one particular path. Consequently, it seems legitimate to work towards obtaining ranges for the software execution on a given hardware, such as those that exist for hardware.

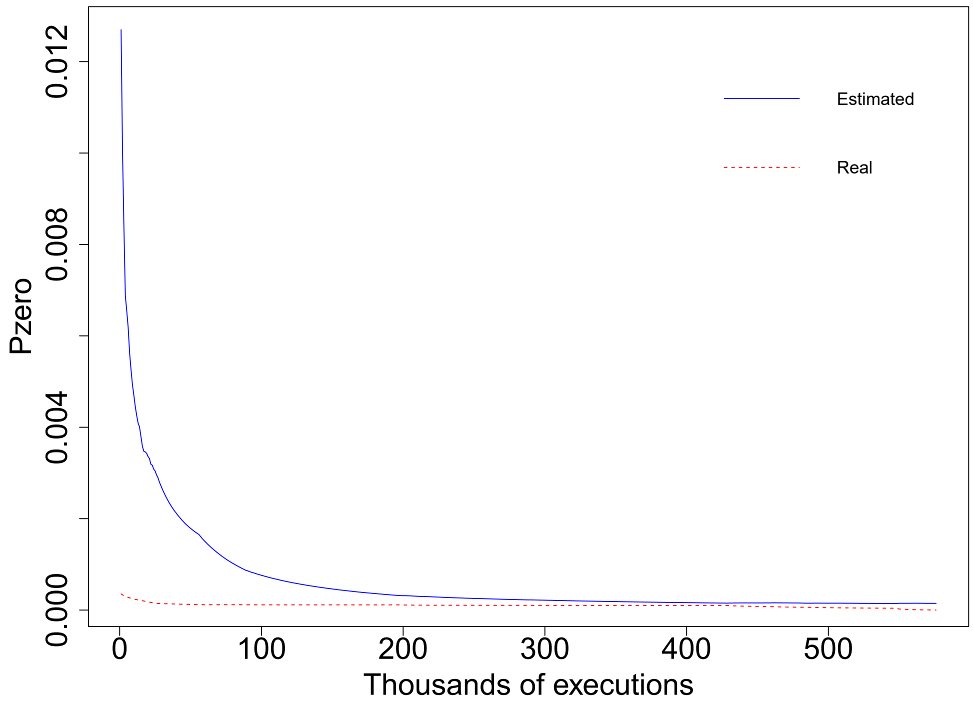


Figure 7.7 P_{zero} values along with execution traces increase (blue points). P_{zero} real value estimated with the remaining data set.

As there is currently no suitable reference for this software execution on a given hardware, in this manuscript we take as a reference the values established by PFH. In this manner, we can obtain an approximate idea if the described method (following the ALARP principle) may fit the reference ranges. For a SIL 2 the standard determines the PFH range to $10^{-6} < PFH \leq 10^{-7}$. If we follow the most conservative approach, where the execution of any untested trace leads to catastrophic consequences, we can estimate the probability of a dangerous failure per hour. In the case of a redundant architecture such as the SIL2LinuxMP's two-out-of-two (2oo2) architecture, the probability of executing an untested path in both cores at the same time is ${P_{zero}}^2$. It is essential to take into account that safety is a system property and, hence, the values set by the standard are more attainable with this type of architecture. Consequently, we would only need to estimate the average execution frequency of system-calls to be able to translate it to a PFH comparable value. SIL2 task is executed with a period of 50 milliseconds and exercises an average of 25 system-calls per period. As a result, the system exercises $18 \cdot 10^5$ system-calls per hour.

In order to translate the previously calculated probability, into probability of potential dangerous failures per hour (considering the given system architecture), we follow the next steps:

1. We need to know the frequency of execution of each system-call in order to know how many calls are made per hour. The AEB system has a period of 50 milliseconds with an average of 25 system-calls per period. This results in $18 \cdot 10^5$ calls per hour.

$$Freq = 3.6 \cdot 10^6 \cdot \frac{25}{50} = 18 \cdot 10^5 \tag{7.4}$$

2. Bearing in mind that we are following the most pessimistic approach, where the execution of any unknown trace is considered dangerous, we can estimate the probability of dangerous failure per hour. For this purpose, we use the Binomial probability mass function (see Equation 5.6).

$$Probability = 1 - \binom{Freq}{0} \cdot P_{zero}(1 - P_{zero})^{Freq} \tag{7.5}$$

   We rest to 1 the probability of not executing any untested path in *Freq* executions. In other words, the cumulative probability of at least one failure.

   If we have a redundant architecture, such as a two-out-of-two (2oo2), the probability of executing an untested trace simultaneously (in the same execution cycle) on both cores would be ${P_{zero}}^2$.

$$Probability = 1 - \binom{Freq}{0} \cdot {P_{zero}}^2(1 - {P_{zero}}^2)^{Freq} \simeq 18 \cdot 10^{-3} \tag{7.6}$$

As you can observe, the resulting probability is significantly higher than the range determined by the standard ($18 \cdot 10^{-3} >> 10^{-6}$). It is important to mention that the current manuscript does not aim to declare that the current Linux kernel is safe to be used in safety-critical systems (see SIL2LinuxMP [OSA], ELISA [elia]), but to contribute with a method that enables the probability estimation of untested Linux kernel paths.

It is essential to note that the calculations are done following the most pessimistic approach, where the execution of any untested path is considered dangerous. Even though the obtained results are significantly higher than the range determined by the standard, the results can be slightly improved by increasing testing effort as P_{zero} tends to decrease and results can be further improved by collaborating with the Linux community working towards the development of safety Linux.

Indeed, it is important to note that these results are achieved with around ten thousand hours of execution data. Therefore, the obtained risk values can be reduced with further testing exercise, as P_{zero} decreases tends to decrease. For instance, if we perform the risk calculation with a reduced data set (the equivalent of four thousand hours exercising effort), the obtained value is $15 \cdot 10^{-2}$. Therefore, the data set size or the testing effort implies a considerable difference in the risk reduction.

7.5 Summary

This chapter introduces a statistical method to estimate the probability of execution of the untested kernel paths due to a safety-related application with the aim of paving the way towards the quantification of residual risk caused by a software malfunction. Even though the study described in the paper is based on the Linux kernel, the approach is potentially applicable to complex software applications exhibiting non-deterministic behavior.

While systematic failures in requirements, design, and implementation generally are not perceived as probabilities, achieving very high systematic capabilities of systems requires stringent technical methods for the specification, design, verification, testing and validation. IEC 61508-3 B.2 states that appropriate justification needs to be provided in the case that 100% coverage cannot be achieved. From a certification criteria, the proposed method provides an objective acceptance criteria and high (justified) confidence as R2 rigor (adequate for SIL 3) states.

Furthermore, the described approach provides an advance in the quantification of risk at the software level as risk is calculated by the multiplication of probability of occurrence and the severity of consequences ($risk = probability * severity$). This is an advantage, which code-coverage percentages cannot provide. Consequently, we believe that the method described facilitates part of the testing phase of complex safety-related systems, where 100% test coverage is not be achievable.

IEC 61508 safety standard states in Part 3-B.2 that appropriate justification needs to be provided in the case that test-coverage is below 100%. Consequently, the proposed method provides an objective acceptance criteria and high (justified) confidence as R2 rigor (adequate for SIL 3) states.

Chapter 8

Statistical Path Coverage (SPC) - Discussion of the proposed technique

The research activities conducted during this thesis have concluded with several contributions in the field of testing software for next-generation safety-related systems. The methods that have been described in this document provide a series of results that allow advancing towards a SPC technique. But how do contributions complement each other?

In this chapter, we analyze the proposed SPC technique and describe in higher detail how to employ the proposed methods. For this purpose, in this chapter, we describe the relationship between the contributions and how to interpret the obtained results. Thereby, this chapter analyzes from a global perspective the contributions made in this thesis and describes how they are complementary. On the one hand, it analyzes the contributions on test coverage (Chapter 5 and 6) while, on the other hand, it examines how Chapter 7 complements the previous ones.

8.1 Discussion of the proposed technique

The research work developed throughout this thesis has resulted in three main contributions. Two of them with the objective of estimating the test coverage, having as reference the traces that have a relevant probability of being executed. The last contribution was aimed at estimating the risk associated with untested traces. In order to define the SPC technique, we have to see how these three contributions complement each other. How do we complement the two test coverage methods (parametric vs. non-parametric)? Should we select only one? Can we join both? Afterward, how do we add risk estimation to the technique? In this section, we propose a process to perform the SPC analysis.

The SPC technique that we propose is based on three main phases: data collection, test coverage and risk estimation. This was introduced in Chapter 3 with a simplified flowchart. Taking into account the analyses presented in this document, we can discuss the technique and define it in more detail. Figure 8.1 explains the overall flowchart of the technique based on the three phases. Firstly, the technique starts collecting the data and validating this data to perform further studies. Secondly, the parametric method is used to estimate the test coverage. The results obtained with the Poisson model are validated with the non-parametric method. Finally, the probability of an untested path is estimated in order to quantify the risk entailed to them.

8.1.1 Data collection

The *"Data collection"* phase defined in Figure 8.1 is explained in detail in Section 3.2.1 and 4.3. Section 3.2.1 introduces the tools and method that is used to record the traces during the testing process. The SPC technique that we are defining in this chapter does not require this exact tool (i.e., DB4SIL2). It is possible to use any other dynamic analysis tool. It is not even necessary to use the FTrace-based tool. We could use or develop other alternatives. In fact, although we do not have the data to confirm it, we believe that the method can be extrapolated to other OSs or software.

Briefly, the system is exercised with a series of test-cases that are executed repeatedly. These test-cases exercise a number of kernel traces and using dynamic analysis tools (Section 3.2.1) we are able to record and post-process the data. Afterward, it is checked whether the data set fulfills the quality and quantity requirements to conduct further analysis (Section 4.4). Quantity is validated with EVT. If EVT results in a non-stable return value, it means that we do not yet have enough test-campaigns to ensure that there will be no major variation in the results. Therefore, the system needs to be further exercised. Once we have enough data, we can examine the quality of it, checking if the data is i.i.d. After having a validated

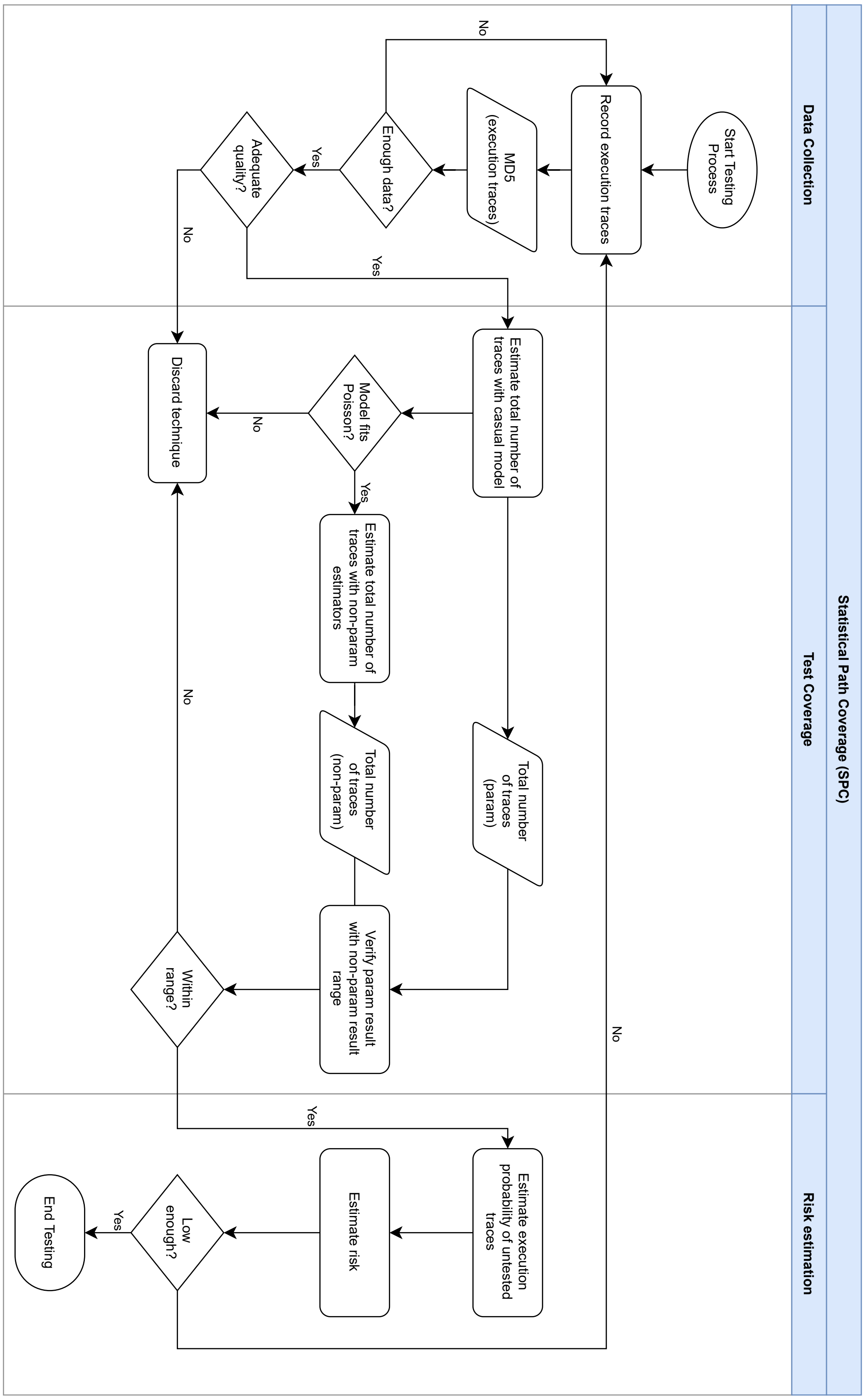

Statistical Path Coverage (SPC)
Data Collection
Test Coverage
Risk estimation
Start Testing Process
Record execution traces
MD5 (execution traces)
Enough data?
No
Yes
Adequate quality?
No
Yes
Estimate total number of traces with casual model
Model fits Poisson?
No
Yes
Discard technique
Estimate total number of traces with non-param estimators
Total number of traces (param)
Total number of traces (non-param)
Verify param result with non-param result range
Within range?
No
Yes
Estimate execution probability of untested traces
Estimate risk
Low enough?
Yes
No
End Testing

Figure 8.1 Flowchart summarizing the SPC technique.

data set it is possible to continue with the following phases, which estimate the test coverage of the recorded data set.

8.1.2 Test coverage

Both Chapter 5 and Chapter 6 share the same objective: estimate the number of traces that have a relevant probability of occurrence in order to quantify the test coverage. Besides, both Chapters are based on statistical techniques. However, the statistical methods are significantly different. On the one hand, we have non-parametric estimators that do not care about the statistical model of the data while, on the other hand, that analysis is mainly based on the modeling of the appearance of untested rare traces.

We believe that a causal model (parametric approach) is more appropriate from a safety argumentation point of view. One of the key benefits is that the model has been selected starting from a theoretical explanation and then validated with empirical data. In this case, the data belongs to the AEB case study. Although a "big-data" approach could have been followed, recording a large number of traces and examining the goodness of fit of a series of models, this can be problematic in the future. This type of approach proposes the best model for your concrete data but could be not valid for other case studies or even with corner cases of the same case study. Therefore, having a casual model based on a theoretical explanation that agrees with the system's behavior increases the reliability of the proposed method. It offers greater confidence that it will respond appropriately in other use cases (see Annex A), and that it will respond correctly to corner cases. Note that this is considerably different from other domains such as ML, where approaches are usually tested only empirically and not theoretically [Cho17].

Regarding non-parametric richness estimators, we need to take into account that they have not been designed for test coverage estimation. Therefore, from a functional safety perspective, it is necessary to be cautious with the results obtained with these estimators. As they are not based on any model, it is difficult to get a generalized estimator that works well in different scenarios. In fact, species richness estimators leave several open questions: How do we know which estimator is the most adequate in all the cases? Maybe in this case study it is *Bootstrap* but in the next case study may be *Chao*. Which one responds more adequately to corner cases?

Taking these points into account, we propose considering the parametric approach as the main provider of the total number of traces and use the non-parametric method as a second source of confidence (apart from dispersion, AIC, BIC, etc.). The non-parametric method can be used as a Validation & Verification (V&V) method inside the proposed SPC technique. On the one hand, the non-parametric richness estimators validate that the use of the Poisson model is appropriate (we did the right thing) and, on the other hand, they also verify the results themselves (we did it in the right ways). Therefore, we need to examine if the total number of traces estimated by the parametric method is in concordance with the range of results provided by the richness estimators. Thus, we have a method with a theoretical basis that estimates the total number of traces and we add a second method that examines that the results have a certain concordance.

In order to evaluate this approach, it is interesting to check the differences between the results of both methods.

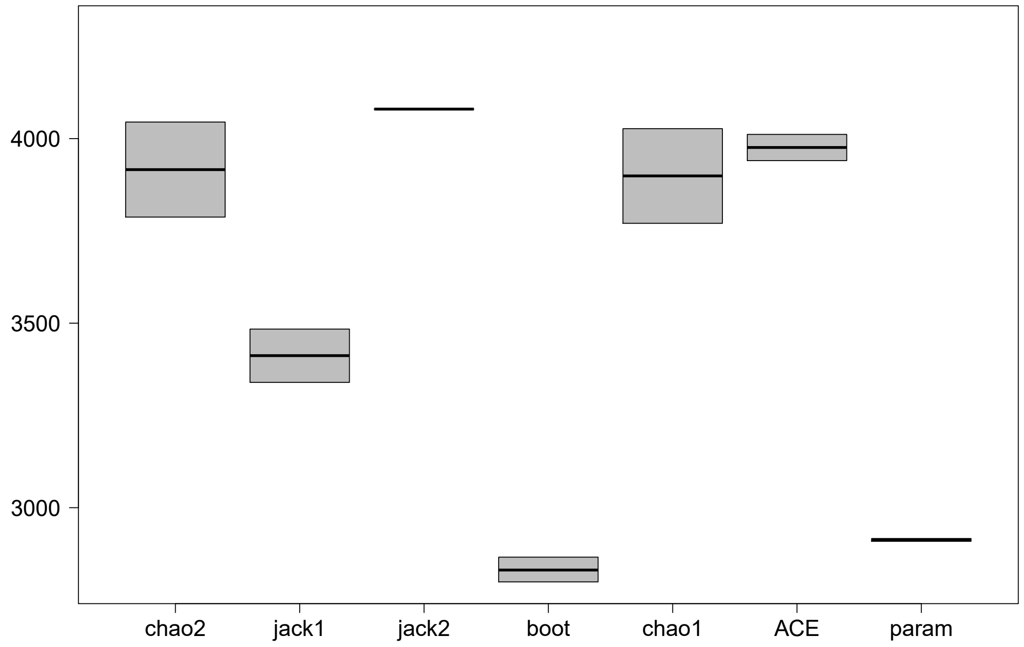


Figure 8.2 Comparison of the obtained results.

The results obtained from the different proposed methods are depicted in Figure 8.2. The box-plot shows the results with 95% CI, which means that each box shows the 95% of the results closer to the mean. Furthermore, it allows visualizing the difference or the alignment of the results. The first conclusion that can be drawn from these results is that they all fall under a similar range. That is, no approach offers a significantly different result than any other. Therefore, we can increase the confidence placed in the proposed methods.

Figure 8.2 shows that all the obtained results are located between ranges of 2767 and 4174. Among them is the value obtained with the parametric method with a value of 2912.68 (± 7.5852). Therefore, we can conclude that the result obtained by the Poisson-based method is within the range of results offered by the different non-parametric estimators. Recall that the richness estimators are based on different principles (abundance and incidence). Moreover, this does not only confirm the suitability of Poisson but also the use of Shannon entropy to select a cut-off frequency.

Although in this case it is within the range, a margin of error of, for example, 5% could be assessed. We believe that this margin may make sense since if we were to re-execute all the test-campaigns we would not have exactly the same results. They could vary slightly. However, currently, we have no formal argumentation to define the percentage of the margin of error. Additionally, if the results are not within may be because the data set is not large enough to make the comparison. Chapter 6 shows that the estimators can be unstable when the data set is not large enough. Therefore, in the cases where non-parametric and parametric approaches do not fit, it would be necessary to analyze as well: i) the standard error, (ii) plots of the results as the size of the database increases, and (iii) tendency of the number of rare traces while data set gets larger. With all these data, we could conclude if the issue is due to the data set size.

Furthermore, we can evaluate the approach also with the reduced data set that we have presented in Chapters 5 and 6. Table 8.1 collects the results with both methods. On the one hand, it provides the results with the parametric method in the three different data sets. On the other hand, it shows the result ranges obtained with the non-parametric estimators. The ranges are the limits with the upper and lower 95 % CIs included.As summarized in Table 8.1, the proposed SPC technique fits all considered scenarios. Thus, we can trust the SPC technique, including its verification process for the test coverage quantification. It is also interesting to note that the same estimators do not always behave equally. For example, the boot estimator is the most optimistic in the *All* data set, but the second most pessimistic in *Read* data set.

Table 8.1 Comparison between parametric and non-parametric results. Non-parametric results range includes the 95 % CIs.

	Parametric	**Non-parametric**	**Fits?**
All	2912.68	2766.80 - 4174.37	✓
Selected	134.36	127.38 - 173.13	✓
Read	43.93	37.19 - 49.88	✓

Once we have estimated the total number of traces with the parametric approach and V&V with the richness estimators, we obtain the test coverage percentage. Afterward, following the guidelines of the SPC technique, we estimate the risk of the traces that have not been tested.

8.1.3 Risk estimation

Although previous methods based on different statistical techniques (parametric vs non-parametric and abundance vs incidence) allow to determine the test coverage, neither the method based on the parametric approach nor the non-parametric one are able to provide a risk estimation.

The risk estimation is based on the simple Good-Turing equations, which use the frequency-of-frequency data to calculate the execution probability of untested paths. The values used for the estimations are based on a Power-law model of the recorded data. Besides, the calculated probability is used to estimate the risk considering that the untested traces' execution has catastrophic consequences. If the risk is not low enough, it is possible to further exercise the system to improve the test coverage and, hence, decrease the probability of execution of untested traces.

We consider the execution probability estimation of untested paths as a step forward in the field of test coverage of complex safety-related systems. Contrary to the techniques that have been employed traditionally, we take into account the uncertainty that these systems possess. In fact, the non-parametric estimators or the parametric approach show a test coverage percentage below 100%. Furthermore, they show that this value is hardly achievable and even to have the required evidences to assure it. Test coverage measures provide possibilities and not probabilities. Consequently, this analysis complements the previous ones and allows us to determine the risk that entails the non-covered paths. This method also contributes to providing adequate explanation when full coverage is not achievable, as stated by the IEC 61508 standard (IEC 61508-3 Ed2 Table B.2).

The latter method does not use the test coverage value provided by previous methods. Thus, it is possible to maintain independence between methods. However, the number of untested traces could be combined with the probability provided by Simple Good-Turing. Further analysis could be carried out to estimate a probability for each untested trace. However, we believe, at this stage of the research, it is more interesting to keep methods independent. At least until they are further analyzed by different experts.

8.2 Summary

This chapter discusses and defines with further detail the SPC technique that we propose in this thesis. The detailed definition of the SPC technique provides the required information to use the technique in additional use cases. The chapter specifies how contributions complement and shows how results based on different approaches exhibit concordance. SPC technique proposes combining both test coverage methods to ensure that the test coverage result is reliable. Hence, we use the non-parametric estimators to V&V that the parametric results are adequate. Additionally, this V&V approach is also examined with an alternative use case in Annex A. Apart from that, the chapter describes how the risk estimation is performed based on the collected data.

Chapter 9

Conclusions and Future Work

As a culmination of the different research activities conducted in this thesis, this final chapter summarizes the main conclusions, discusses the obtained results, points out the main contributions, and identifies the future lines. This chapter is organized as follows. Section 9.1 provides a brief summary of the thesis and collects the main conclusions. A critical review of the proposed methods is provided in Section 9.2. Section 9.3 highlights the principal contributions of the conducted research activities. Finally, Section 9.4 closes the thesis and comments on the future research lines.

9.1 Summary of the thesis and overall conclusions

During the state of the art review, we identified some of the main challenges that the functional safety domain is facing with the development and safety certification of next-generation autonomous systems, specifically in the testing process of Linux-based safety-related systems. The vast majority of the functional safety standards normatively require the testing process of safety-related systems. However, the complexity that characterizes the next-generation safety-related systems limits the ability to derive statements about the residual risk. Furthermore, techniques that adequately control the test coverage and that are capable of quantifying residual risk were identified as main gaps in the literature. In fact, through a deep review of the literature, we observed that in order to quantify the residual risk that entails software execution we need probabilities and not just possibilities. Accordingly, we investigated the development of methods that are able to calculate the number of traces that have a relevant probability of being executed by the safety application and estimate the execution probability of the ones that have not been tested. For this purpose, dynamic analysis together with statistics was identified as a possible approach to pave the way towards safety assurance in the field of testing complex safety-related systems.

The different research activities performed during this thesis have analyzed several statistical techniques. This thesis dissertation describes in detail the analyses that have been performed and contributes with different statistical methods. Additionally, the obtained results reinforce our interest in further research of the proposed methods in order to improve them and obtain more conclusive results.

During the course of the research, the challenge of achieving full coverage has been observed and, therefore, the risk that the software entails due to the existence of untested code. But is this risk low enough? In 2019, there was an average of more than 100 traffic fatalities per day in the United States. Although this involves a whopping rate of 38,000 annual deaths, this is a socially acceptable accident rate for human-driven vehicles. But would this accident rate be accepted in the case of autonomous vehicles? And the answer is no, it is socially not acceptable. Proof of this is the front pages that occupy the accidents of autonomous cars [ubeb, ubea, tesa, tesb]. Accordingly, the rates obtained from the proposed methods have not to be read as absolute rates, but rather as social acceptance rates. Acceptable rates are derived from societal expectations and are dependent on many subjective factors like natural or human/technical causes. If we look at where such acceptable rates might come from we can examine the Federal Aviation Administration (FAA)'s approach to a catastrophic loss of a type certified aircraft. The assumption made in 1980 roughly were [De 11, Gui18b]: there are 100 built aircraft of that type and each aircraft operates 3000h per year for 33 years. There are also 10 critical systems per aircraft. The FAA decided as an acceptable rate that no more than 1 plane of the fleet shall be lost during the types lifetime, which means $1 \cdot 10^{-9}$ catastrophic failures/h of operations. So, should the safety domain examine the acceptable acceptance rate of next-generation autonomous systems? Unfortunately, there are no zero-risk technologies so we have to agree on allocating some level of risk that we are willing to accept. At the same time, there simply are technical limits to the assured levels of failure that we can achieve which are in the $10 \cdot 10^{-9}/h$ range for any technical system [De 11]. Consequently, the contributions made by this thesis aim to provide qualitative and quantitative measures to estimate the test coverage and the residual risk. Consequently, the purpose is to pave the way towards the evaluation of whether this type of system complies with social acceptance rates following principles such as ALARP.

The research activities conducted in this thesis are presented and analyzed in the context of an AEB case study based on the Linux kernel. Even though this case study is not complete development of a road deployable AEB, it contains the main features of the autonomous safety systems that want to be developed by different industries. Therefore, the results obtained during the development of this thesis can be considered representative of next-generation safety-related autonomous systems.

9.2 Critical review of the proposed technique

This thesis introduces the SPC technique with the aim of progressing in the field of test coverage and risk quantification of the next-generation safety-related autonomous systems, focusing in this case specifically on the Linux kernel. The methods presented in this manuscript can be considered complementary as each one provides results that add extra information and knowledge. It is important to mention that the main objective of the conducted research is not to identify the specific statistical methods but rather to examine the viability and benefits of statistical-based alternatives to cope with these systems' complexity. Firstly, we analyze the modeling of the appearance of the previously untested path while the system is being under test. The model allows estimating the total number of paths that have a relevant probability of being executed and, hence, we estimate the test coverage with a given uncertainty. Secondly, this thesis studies different non-parametric estimators with the intent of quantifying the number of traces that are not covered during the verification phase. Afterward, we examine a technique to be able to estimate the execution probability of those untested traces. All alternatives show positive results in order to pave the way towards the safety assurance of next-generation safety-related systems. Therefore, we define the SPC technique explaining how to employ and combine the three methods.

Test coverage estimation by Poisson modeling seems an interesting alternative to traditional techniques. The parametric approach successfully models the appearance of previously untested traces with Poisson regression and allows estimating the total expected number of traces. Meanwhile, the non-parametric estimators can be considered as a second source of verification of the obtained results. Although the estimators do not give exactly the same results, all of them provide significantly similar values and, thus, we obtain a range of results where the parametric approach should fit. Moreover, this argument is strengthened by the fact that some of these estimators are based on different data analyses (i.e., incidence vs. abundance). From a general perspective, it is encouraging that the results of the parametric and non-parametric alternatives coincide. For example, if the Poisson regression would conclude twice as many total traces as the non-parametric estimators, we could be sure that at least one of the methods is not adequate. But in this case, we can see how the two alternatives based on different principles offer significantly similar result ranges.

As a complement to the proposed test coverage methods, we also consider the execution probability estimation of untested paths as a step forwards in the field of test coverage of complex safety-related systems. Contrary to the techniques that have been employed traditionally, we take into account the uncertainty that these systems possess. Consequently, this analysis complements the previous ones and allows us to determine the risk that entails the non-covered paths. Nonetheless, there is a need for a reference value, equivalent to PFH ranges for hardware failures, in order to comprehend the risk associated with untested paths. Establishing an equivalent range for software involves analyzing the social acceptance rate for this type of system and achieving an agreement from the community of functional safety experts. Consequently, we believe this needs further discussion with a Certification Authority (CA) and experts in the field.

It is important to note that the results obtained by the application of the described method are use case specific. They do not provide a generalized set of results valid for an arbitrary use case. The process itself is defined in a way that it can be applied to any use case though. While the process itself is use case agnostic, the numeric targets and thresholds of the process execution need to be adjusted to fit the requirements of the use case. Furthermore, the methods and the proposed SPC technique are also positively evaluated with an additional case study in Annex A.

9.3 Contributions of the thesis

The main contributions of the research activities conducted throughout the thesis are:

- **A state of the review of the safety assurance of next-generation safety-related systems** focusing primarily on test coverage of the Linux kernel.
- **Comparison between the safety technical requirements of GNU/Linux and a certifiable hypervisor**, identifying the high capabilities of GNU/Linux for a mixed-criticality system. The work also contributes to the definition of a safety architecture based on container technology, taking as reference the preliminary architecture design performed in the SIL2LinuxMP project [AMGP+19].
- **An analysis of the inherent non-determinism of the Linux kernel by data obtained from iterative test-case**, visualizing the different *unique traces* that occur per system-call. This study also leads to the improvement of the DB4SIL2 tool and to the data acquisition method.
- **A parametric approach to model the appearance of untested traces** and, hence, be able to estimate the total number of paths [AMGP+21a].
  - Find a model that fits adequately the data and concise with the theoretical explanation for its use.
  - Calculation of the total number of traces after obtaining the model's parameters.
  - Examination of the results depending on different cut-off limits.
- **A dedicated method based on non-parametric estimators to quantify the total number of paths** that have a relevant probability of execution by the application under test [AGPC+21].
  - Examines different estimators based on distinct principles (i.e., abundance and incidence).
  - Study of the accumulation curve achieved from the data set recorded during the testing of the system.
  - Analysis of the results obtained from the non-parametric estimators.
- **A statistical method for estimating the probability of execution of the untested kernel paths with the aim of paving the way towards the verification of safety systems** based on the Linux kernel, where 100% test-coverage cannot be achieved [AMGP+21b].
- **Definition of SPC technique based on the contributions of the thesis.** It explains and analyzes how the methods complement each other and how they should be used in an use case.

9.4 Closure and future lines

With the intention of improving the methods and assuring their suitability, we should continue the research activities by examining the proposed technique with additional use cases and other system-contexts. For instance, we believe it would be interesting to investigate the response of the methods with the same application but different system-context (e.g., different CPU loads). In addition, we believe it is necessary to research further into the branching model. The aim is to gain further knowledge about the mechanism(s) behind the non-determinism of the Linux kernel. So we need to find answers to several open questions: Which are the sources of asynchronouness? How do they behave? Are they dependent of the system-call?

It is essential to note that these methods are only justifiable performing continuous monitoring of the process. Continues monitoring allows updating the results constantly and, hence, it strengthens the results while the data set increases. Indeed, this would allow verifying whether the current results fit as more data is collected. Contrary to biology, where obtaining new samples is generally not that easy, in the software case, it is potentially easier. Consequently, a significant higher assurance should be doable by exercising the system continuously and using this approach during the intensive test campaigns used to validate safety-related systems.

In the research activities performed during this thesis, we have entirely focused on the system-call exercised by the SIL2 task, leaving aside small pieces of inter-calls. Since the obtained results seem promising, further analysis needs to be carried out to include the kernel function-calls that occur between system-calls. We believe that this can be achieved by sampling on internal kernel functionality in much the same way as we have sampled on the kernel's system-call interface.

It should also be mentioned that if the statistical methods are adequate, the tools used should be qualified (IEC 61508-4 Ed2 3.2.11). Many of the libraries that have been used for this research are open-source contributions that need further examination in order to demonstrate their correctness. For instance, during the research activities carried out, some bugs have been detected in different tools and libraries.

Finally, we believe it is interesting and necessary that the described methods undertake a thorough review by experts and a CA. As mentioned above, this study intends to gain insight into the benefits of statistical alternatives concerning traditional techniques. Nevertheless, it is also essential to identify the limits and drawbacks that these state-of-the-art techniques may have. Furthermore, we believe that these methods may be potentially valid for other OSs or software layers, although we cannot confirm it as the analysis has not been conducted.

Bibliography

[AAAAC17] I. Agirre, J. Abella, M. Azkarate-Askasua, and F. J. Cazorla. On the tailoring of CAST-32A certification guidance to real COTS multicore architectures. In *12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–8. IEEE, 2017. doi:10.1109/SIES.2017.7993376.

[AAAL+15] I. Agirre, M. Azkarate-Askasua, A. Larrucea, J. Perez, T. Vardanega, and F. J. Cazorla. A Safety Concept for a Railway Mixed-Criticality Embedded System Based on Multicore Partitioning. In *IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 1780–1787. IEEE, 2015. doi:10.1109/cit/iucc/dasc/picom.2015.268.

[AGPC+21] I. Allende, N. M. Guire, J. Perez-Cerrolaza, L. G. Monsalve, J. Petersohn, and R. Obermaisser. Statistical test coverage for Linux-based next-generation autonomous safety-related systems. *IEEE Access*, 9:1–14, 2021. doi:10.1109/ACCESS.2021.3100125.

[Ald13] T. Aldemir. A survey of dynamic methodologies for probabilistic safety assessment of nuclear power plants. *Annals of Nuclear Energy*, 52:113–124, 2013. doi:10.1016/j.anucene.2012.08.001.

[ALRL04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33, 2004. doi:10.1109/TDSC.2004.2.

[Alt09] J. Altenberg. Using the Realtime Preemption Patch on ARM CPUs. In *Real-Time Linux Workshop*, pages 28–30, 2009.

[Alt16] J. Altenberg. Introduction to Realtime Linux. In *Embedded Linux Conference Europe*, 2016.

[AMGP+19] I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve, N. Uriarte, and R. Obermaisser. Towards Linux for the development of mixed-criticality embedded systems based on multi-core devices. In *15th European Dependable Computing Conference (EDCC)*, pages 47–54. IEEE, 2019. doi:10.1109/EDCC.2019.00020.

[AMGP+21a] I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve, and R. Obermaisser. Towards Linux based safety systems—A statistical approach for software execution path coverage. *Journal of Systems Architecture*, 116:102047, 2021. doi:10.1016/j.sysarc.2021.102047.

[AMGP$^+$21b] I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve, J. Fernández, and R. Obermaisser. Estimation of Linux kernel Execution Path Uncertainty for Safety Software Test Coverage. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1446–1451. IEEE, 2021. doi:10.23919/DATE51398.2021.9473951.

[AOP18] H. Ahmadian, R. Obermaisser, and J. Perez. *Distributed Real-Time Architecture for Mixed-Criticality Systems*. Taylor & Francis Incorporated, 2018.

[AOP$^+$20] I. Agirre, P. Onaindia, T. Poggi, I. Yarza, F. J. Cazorla, L. Kosmidis, K. Grüttner, M. Abuteir, J. Loewe, J. M. Orbegozo, et al. UP2DATE: Safe and secure over-the-air software updates on high-performance mixed-criticality systems. In *23rd Euromicro Conference on Digital System Design (DSD)*, pages 344–351. IEEE, 2020. doi:10.1109/DSD51259.2020.00063.

[ARM] ARM. ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. Available: https://www.arm.com/company/news/2015/04/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade. [Online].

[ARP4754A] Society of Automotive Engineers International. ARP4754A: Guidelines for Development of Civil Aircraft and Systems. Standard, 2010.

[Bas17] C. V. Basualdo. Choosing the best non-parametric richness estimator for benthic macroinvertebrates databases. *Revista de la Sociedad Entomológica Argentina*, 70(1-2), 2017.

[Ben85] J. Bentley. Programmimg pearls. *Commun. ACM*, 28(9):896–901, Sept. 1985. doi:10.1145/4284.315122.

[BO78] K. P. Burnham and W. S. Overton. Estimation of the size of a closed population when capture probabilities vary among animals. *Biometrika*, 65(3):625–633, 1978. doi:10.2307/2335915.

[BOW13] L. Bulwahn, T. Ochs, and D. Wagner. Research on an Open-Source Software Platform for Autonomous Driving Systems. *BMW Car IT GmbH, Munich, Germany*, 2013.

[Bé16] J. Béguinot. Basic Theoretical Arguments Advocating Jackknife-2 as Usually being the Most Appropriate Nonparametric Estimator of Total Species Richness. *Annual Research & Review in Biology*, 10:1–12, 01 2016. doi:10.9734/ARRB/2016/25104.

[CAA$^+$16] F. J. Cazorla, J. Abella, J. Andersson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu, et al. PROXIMA: Improving measurement-based timing analysis through randomisation and probabilistic analysis. In *Euromicro Conference on Digital System Design (DSD)*, pages 276–285. IEEE, 2016. doi:10.1109/DSD.2016.22.

[CC94] R. K. Colwell and J. A. Coddington. Estimating terrestrial biodiversity through extrapolation. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 345(1311):101–118, 1994. doi:10.1098/rstb.1994.0091.

[CC16] A. Chao and C.-H. Chiu. Species Richness: Estimation and Comparison. *Wiley StatsRef: Statistics Reference Online*, pages 1–26, 2016. doi:10.1002/9781118445112.stat03432.pub2.

[CCLG09] A. Chao, R. K. Colwell, C.-W. Lin, and N. J. Gotelli. Sufficient sampling for asymptotic minimum species richness estimators. *Ecology*, 90(4):1125–1133, 2009. doi:10.1890/07-2147.1.

[CGSH$^+$12] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems*, pages 91–101. IEEE, 2012. doi:10.1109/ECRTS.2012.31.

[Cha84] A. Chao. Non-parametric Estimation of the Number of Classes in a Population. *Scandinavian Journal of Statistics*, 11:265–270, 01 1984. doi:10.2307/4615964.

[Cha87] A. Chao. Estimating the Population Size for Capture-Recapture Data with Unequal Catchability. *Biometrics*, pages 783–791, 1987. doi:10.2307/2531532.

[Cho17] F. Chollet. *Deep Learning with Python*. Manning Publications Co., 1st edition, 2017.

[CKH17] J. Corbet and G. Kroah-Hartman. Linux kernel Development Report, 2017.

[CKM$^+$19] F. J. Cazorla, L. Kosmidis, E. Mezzetti, C. Hernandez, J. Abella, and T. Vardanega. Probabilistic Worst-Case Timing Analysis: Taxonomy and Comprehensive Survey. *ACM Computing Surveys (CSUR)*, 52(1):1–35, 2019. doi:10.1145/3301283.

[CMC04] R. K. Colwell, C. X. Mao, and J. Chang. Interpolating, extrapolating, and comparing incidence-based species accumulation curves. *Ecology*, 85(10):2717–2727, 2004. doi:10.1890/03-0557.

[CPL$^+$02] R. Condit, N. Pitman, E. G. Leigh, J. Chave, J. Terborgh, R. B. Foster, P. Núnez, S. Aguilar, R. Valencia, G. Villa, et al. Beta-Diversity in Tropical Forest Trees. *Science*, 295(5555):666–669, 2002. doi:10.1126/science.1066854.

[CSL$^+$19] J. Cui, G. Sabaliauskaite, L. S. Liew, F. Zhou, and B. Zhang. Collaborative Analysis Framework of Safety and Security for Autonomous Vehicles. *IEEE Access*, 7:148672–148683, 2019. doi:10.1109/ACCESS.2019.2946632.

[CT13] A. C. Cameron and P. K. Trivedi. *Regression Analysis of Count Data*, volume 53. Cambridge University Press, 2013.

[De 11] F. De Florio. *Airworthiness Requirements*. Butterworth-Heinemann, Oxford, second edition edition, 2011. doi:https://doi.org/10.1016/B978-0-08-096802-5.10004-2.

[dis] DISA Has Released the Red Hat Enterprise Linux 8 STIG. Available: https://www.redhat.com/es/blog/disa-has-released-red-hat-enterprise-linux-8-stig. [Online].

[Dow13] A. Downey. *Think Bayes: Bayesian Statistics in Python*. "O'Reilly Media, Inc.", 2013.

[Dvo09] D. Dvorak. NASA Study on Flight Software Complexity. *AIAA Infotech at Aerospace Conference and Exhibit and AIAA Unmanned...Unlimited Conference*, 04 2009. doi:10.2514/6.2009-1882.

[elia] Enabling Linux in Safety Applications (ELISA). Available: https://elisa.tech/. [Online].

[ELIb] ELISA. elisa-tech/workgroups. Available: https://github.com/elisa-tech/workgroups. [Online].

[EN50126] European Committee for Electrotechnical Standardization. EN50126: Railway Applications. The Specification and Demonstration of Dependability, Reliability, Availability, Maintainability and Safety (RAMS). Generic RAMS Process. Standard, 2017.

[EN50128] European Committee for Electrotechnical Standardization. EN50128: Railway Applications. Communication, signalling and processing systems – Software for railway control and protection systems. Standard, 2011.

[EN50129] European Committee for Electrotechnical Standardization. EN50129: Railway applications. Communication, signalling and processing systems – Safety related electronic systems for signalling. Standard, 2003.

[eni] European Union Agency for Cybersecurity - Critical Infrastructures and Services. Available: https://www.enisa.europa.eu/topics/critical-information-infrastructures-and-services. [Online].

[EUC] The EU Cybersecurity Act. Available: https://digital-strategy.ec.europa.eu/en/policies/cybersecurity-act. [Online].

[Faw13] D. L. Fawcett. Lecture notes in statistics course, 2013.

[FDW13] M. Fisher, L. Dennis, and M. Webster. Verifying autonomous systems. *Communications of the ACM*, 56(9):84–93, 2013. doi:10.1145/2494558.

[FNT$^+$20] D. D. Fan, J. Nguyen, R. Thakker, N. Alatur, A.-a. Agha-mohammadi, and E. A. Theodorou. Bayesian learning-based adaptive control for safety critical systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4093–4099. IEEE, 2020. doi:10.1109/ICRA40945.2020.9196709.

[FPA+21] J. Fernández, J. Perez, I. Agirre, I. Allende, J. Abella, and F. J. Cazorla. Towards Functional Safety Compliance of Matrix-Matrix Multiplication for Machine Learning-based Autonomous Systems. *Journal of Systems Architecture*, page 102298, 2021. doi:10.1016/j.sysarc.2021.102298.

[ftr17] FTrace. Available: https://www.kernel.org/doc/Documentation/trace/ftrace.txt, Jul 2017. [Online].

[Ful99] R. Fullwood. *Probabilistic Safety Assessment in the Chemical and Nuclear Industries*. Elsevier, 1999.

[Gal16] Y. Gal. *Uncertainty in Deep Learning*. PhD thesis, University of Cambridge, 2016.

[GC01] N. J. Gotelli and R. K. Colwell. Quantifying biodiversity: Procedures and pitfalls in the measurement and comparison of species richness. *Ecology letters*, 4(4):379–391, 2001. doi:10.1046/j.1461-0248.2001.00230.x.

[GC11] N. J. Gotelli and R. K. Colwell. Estimating Species Richness. *Biological diversity: frontiers in measurement and assessment*, 12(39-54):35, 2011.

[gco] Using gcov with the Linux kernel. Available: https://www.kernel.org/doc/html/v4.15/dev-tools/gcov.html. [Online].

[GLB+19] H. F. Grip, J. Lam, D. S. Bayard, D. T. Conway, G. Singh, R. Brockers, J. H. Delaune, L. H. Matthies, C. Malpica, T. L. Brown, et al. Flight control system for NASA's Mars helicopter. In *American Institute of Aeronautics and Astronautics (AIAA) Scitech Forum*, page 1289, 2019. doi:10.2514/6.2019-1289.

[GLC+15] Z. Gao, Y. Liang, M. B. Cohen, A. M. Memon, and Z. Wang. Making System User Interactive Tests Repeatable: When and What Should we Control? In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 55–65. IEEE, 2015. doi:10.1109/ICSE.2015.28.

[Gru12] J. Gruen. Linux in space. *The Linux Foundation*, 2012.

[GT09] A. Ghorbel and A. Trabelsi. Measure of financial risk using conditional extreme value copulas with EVT margins. *Journal of Risk*, 11(4):51, 2009. doi:10.21314/JOR.2009.196.

[Gui18a] N. M. Guire. Building Safe Systems with Linux, 2018.

[Gui18b] N. M. Guire. Introduction to 61508 – Part II – Principles, 2018.

[Gui19] N. M. Guire. SIL2 61508 Ed 2 Compliance Route. *SIL2LinuxMP – The Safety Project for Linux*, 2019.

[Gus] A. Gustafsson. Egypt. Available: https://www.gson.org/egypt/egypt.html. [Online].

[HFP$^+$20] C. Hernandez, J. Flieh, R. Paredes, C.-A. Lefebvre, I. Allende, J. Abella, D. Trillin, M. Matschnig, B. Fischer, K. Schwarz, et al. SELENE: Self-Monitored Dependable Platform for High-Performance Safety-Critical Systems. In *23rd Euromicro Conference on Digital System Design (DSD)*, pages 370–377. IEEE, 2020. doi:10.1109/DSD51259.2020.00066.

[HHRB01] J. B. Hughes, J. J. Hellmann, T. H. Ricketts, and B. J. Bohannan. Counting the Uncountable: Statistical Approaches to Estimating Microbial Diversity. *Applied and environmental microbiology*, 67(10):4399, 2001. doi:10.1128/AEM.67.10.4399-4406.2001.

[Hil14] J. M. Hilbe. *Modeling count data.* Cambridge University Press, 2014.

[HM13] A. Hazem and E. Morin. A Comparison of Smoothing Techniques for Bilingual Lexicon Extraction from Comparable Corpora. In *Proceedings of the Sixth Workshop on Building and Using Comparable Corpora*, pages 24–33, 2013.

[Hob15] C. Hobbs. *Embedded Software Development for Safety-Critical Systems.* Auerbach Publications, 2015.

[Hos90] J. R. Hosking. L-moments: Analysis and estimation of distributions using linear combinations of order statistics. *Journal of the Royal Statistical Society: Series B (Methodological)*, 52(1):105–124, 1990. doi:10.2307/2345653.

[HWC13] M. Hollander, D. A. Wolfe, and E. Chicken. *Nonparametric statistical methods*, volume 751. John Wiley & Sons, 2013. doi:10.1007/978-1-4419-1698-3_313.

[IEC61508] International Electrotechnical Commission. IEC 61508 (1-7): Functional safety of Electrical/Electronic/Programmable Electronic safety-related systems (Second edition). Standard, 2010.

[IEC61511] International Electrotechnical Commission. IEC 61511: Functional safety – Safety instrumented systems for the process industry sector. Standard, 2011.

[IEC61513] IEC 61513 Nuclear Power Plants – Instrumentation and Control for Systems Important to Safety – General Requirements for Systems. Standard, International Electrotechnical Commission, 2011.

[Ins11] M. Instruments. An introduction to Functional Safety and IEC 61508. *AN9025, Available at: www. mtl-inst. com/product/mtl_safety_related_sr_series_isolators*, 2011.

[ISO21448] International Organization for Standardization. ISO/PAS 21448: Road vehicles – Safety of the intended functionality. Standard, 2019.

[ISO22201] International Organization for Standardization. ISO 22201: Lifts (elevators), escalators and moving walks. Standard, 2017.

[ISO26262] International Organization for Standardization. ISO 26262 Road vehicles – Functional safety. Standard, 2011.

[Iye02] M. Iyer. Analysis of Linux test project's kernel code coverage. *Austin, TX: IBM Corporation*, 2002.

[JGBF12] X. Jean, M. Gatti, G. Berthon, and M. Fumey. MULCORS-Use of Multicore Processors in airborne systems. *EASA, Tech. Rep.*, 2012.

[JKK05] N. L. Johnson, A. W. Kemp, and S. Kotz. *Univariate Discrete Distributions*, volume 444. John Wiley & Sons, 2005.

[Jr.51] F. J. M. Jr. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951. doi:10.1080/01621459.1951.10500769.

[JWHT13] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An Introduction to Statistical Learning*, volume 112. Springer, 2013.

[KBC15] A. Kendall, V. Badrinarayanan, and R. Cipolla. Bayesian SegNet: Model Uncertainty in Deep Convolutional Encoder-Decoder Architectures for Scene Understanding. *arXiv:1511.02680*, 2015.

[KC16] A. Kendall and R. Cipolla. Modelling Uncertainty in Deep Learning for Camera Relocalization. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 4762–4769. IEEE, 2016.

[kco] kcov. Available: https://github.com/SimonKagstrom/kcov. [Online].

[kera] Linux Kernel Statistics. Available: https://github.com/satoru-takeuchi/linux-kernel-statistics. [Online].

[kerb] The Linux kernel Archives. Available: https://www.kernel.org/. [Online].

[KFFW19] P. Koopman, U. Ferrell, F. Fratrik, and M. Wagner. A Safety Standard Approach for Fully Autonomous Vehicles. In *International Conference on Computer Safety, Reliability, and Security*, pages 326–332. Springer, 2019. doi:10.1007/978-3-030-26250-1_26.

[Koo19] P. Koopman. An Overview of Draft UL 4600:'Standard for Safety for the Evaluation of Autonomous Products.'. *Edge Case Research*, 2019.

[Kop19] H. Kopetz. *Simplicity is Complex*. Springer International Publishing, 2019.

[KVDS06] R. Kindt, P. Van Damme, and A. Simons. Patterns of Species Richness at Varying Scales in Western Kenya: Planning for Agroecosystem Diversification. *Biodiversity & Conservation*, 15(10):3235–3249, 2006. doi:10.1007/s10531-005-0311-9.

[LB78] G. Ljung and G. Box. On a Measure of Lack of Fit in Time Series Models. *Biometrika*, 65, 08 1978. doi:10.1093/biomet/65.2.297.

[Lep17] H. Leppinen. Current use of linux in spacecraft flight software. *IEEE Aerospace and Electronic Systems Magazine*, 32:4–13, 10 2017. doi:10.1109/MAES.2017.160182.

[Lev16] N. G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. The MIT Press, 2016.

[LH19] K. Lu and H. Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019. doi:10.1145/3319535.3354244.

[LHEM14] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An Empirical Analysis of Flaky Tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 643–653, 2014. doi:10.1145/2635868.2635920.

[LHRF03] P. Larson, N. Hinds, R. Ravindran, and H. Franke. Improving the Linux Test Project with kernel code coverage analysis. In *Proceedings of the Ottawa Linux Symposium*, pages 260–275. Citeseer, 2003.

[lina] Linux Grows on Government Systems. Available: https://www.govtech.com/security/Linux-Grows-on-Government-Systems.html. [Online].

[linb] Linux in 2020: 27.8 million lines of code in the kernel, 1.3 million in systemd. Available: https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-kernel-1-3-million-in-systemd/. [Online].

[Mag07] A. E. Magurran. Species abundance distributions over time. *Ecology letters*, 10(5):347–354, 2007. doi:10.1111/j.1461-0248.2007.01024.x.

[mck20] Rcu usage in the linux kernel: Eighteen years later. *SIGOPS Operating Systems Review*, 54(1):47–63, 2020. doi:10.1145/3421473.3421481.

[Mel04] A. Melo. A critique of the use of jackknife and related non-parametric techniques to estimate species richness. *Community Ecology*, 5(2):149–157, 2004. doi:10.1556/ComEc.5.2004.2.1.

[MGA20] N. Mc Guire and I. Allende. Approaching certification of complex systems. In *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 70–71. IEEE, 2020. doi:10.1109/DSN-W50199.2020.00022.

[MGOS09] N. Mc Guire, P. Okech, and G. Schiesser. Analysis of inherent randomness of the Linux kernel. In *Proc. 11th Real-Time Linux Workshop*, 2009.

[Mit09] M. Mitchell. *Complexity: A guided tour*. Oxford University Press, 2009.

[ncca] ncc – The new generation C compiler. Available: http://students.ceid.upatras.gr/~sxanth/ncc/. [Online].

[nccb] ncc(1): source code analysis – Linux man page. Available: https://linux.die.net/man/1/ncc. [Online].

[New17] M. Newman. Power-law distribution, 2017. doi:10.1111/j.1740-9713.2017.01050.x.

[oD12] D. of Defense. MIL-STD-882E: Standard Practice for System Safety. 2012.

[OGOO15] P. Okech, N. M. Guire, and W. Okelo-Odongo. Inherent Diversity in Replicated Architectures. *arXiv:1510.02086*, 2015.

[O'h05] R. O'hara. Species richness estimators: How many species can dance on the head of a pin? *Journal of Animal Ecology*, 74(2):375–386, 2005. doi:10.1111/j.1365-2656.2005.00940.x.

[OKL$^+$07] J. Oksanen, R. Kindt, P. Legendre, B. O'Hara, M. H. H. Stevens, M. J. Oksanen, and M. Suggests. The Vegan Package. *Community ecology package*, 10(631-637):719, 2007.

[Oks13] J. Oksanen. Vegan: ecological diversity. *R Project*, 2013.

[OMG10] P. Okech and N. Mc Guire. Analysis of Statistical Properties of Inherent Randomness. In *Proc. 12th Real-Time Linux Workshop*, 2010.

[OMGF14] P. Okech, N. Mc Guire, and C. Fetzer. Utilizing inherent diversity in complex software systems. In *Proc. of The Australian System Safety Conference (ASSC2014). Australian Computer Society, Inc*, 2014.

[OMGFOO13] P. Okech, N. Mc Guire, C. Fetzer, and W. Okelo-Odongo. Investigating execution path non-determinism in the Linux kernel. In *Proc. 14th Real-Time Linux Workshop, Lugano. OSADL*, 2013.

[Org] W. H. Organization. Risk factors for road traffic injuries.

[OSA] OSADL. SIL2LinuxMP – The Safety Project for Linux. Available: https://sil2.osadl.org/. [Online].

[OY17] S. Ozlati and R. Yampolskiy. The Formalization of AI Risk Management and Safety Standards. In *Workshops at the Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[PCOA$^+$20] J. Perez-Cerrolaza, R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and I. Allende. Multi-core devices for safety-critical systems: A survey. *ACM Computing Surveys (CSUR)*, 53(4):1–38, 2020. doi:10.1145/3398665.

[PGB18] A. Platschek, N. Guire, and L. Bulwahn. Certifying Linux: Lessons Learned in Three Years of SIL2LinuxMP. 02 2018.

[PGN$^+$14] J. Perez, D. Gonzalez, C. F. Nicolas, T. Trapman, and J. M. Garate. A safety certification strategy for IEC-61508 compliant industrial mixed-criticality systems based on multicore partitioning. In *Euromicro Conference on Digital System Design*, pages 394–400. IEEE, 2014. doi:10.1109/DSD.2014.38.

[PGTT15] J. Perez, D. Gonzalez, S. Trujillo, and T. Trapman. A safety concept for an IEC-61508 compliant fail-safe wind power mixed-criticality system based on multicore and partitioning. In *Ada-Europe International Conference on Reliable Software Technologies*, volume 9111, pages 3–17. Springer International Publishing, 2015. doi:10.1007/978-3-319-19584-1_1.

[Pla16] A. Platschek. DB4SIL2 – kernel assurance data for SIL2LinuxMP, 2016.

[PNAC92] I. A. Papazoglou, Z. Nivolianitou, O. Aneziris, and M. Christou. Probabilistic safety analysis in chemical installations. *Journal of Loss Prevention in the Process Industries*, 5(3):181–191, 1992. doi:10.1016/0950-4230(92)80022-Z.

[PWH$^+$17] C. R. Prause, J. Werner, K. Hornig, S. Bosecker, and M. Kuhrmann. Is 100% Test Coverage a Reasonable Requirement?Lessons Learned from a Space Software Project. In *International Conference on Product-Focused Software Process Improvement*, pages 351–367. Springer, 2017. doi:10.1007/978-3-319-69926-4_25.

[PZ17] F. Palomba and A. Zaidman. Does refactoring of test smells induce fixing flaky tests? In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, 2017. doi:10.1109/ICSME.2017.12.

[QLG16] F. Qin, B. Liu, and K. Guo. Using EVT for Geological Anomaly Design and Its Application in Identifying Anomalies in Mining Areas. *Mathematical Problems in Engineering*, 2016, 2016. doi:10.1155/2016/3436192.

[rea] Real-Time Linux Continues Its Way to Mainline Development and Beyond. Available: https://www.linuxfoundation.org/blog/real-time-linux-continues-its-way-to-mainline-development-and-beyond/. [Online].

[Rea20] Real-Time Linux. Available: https://wiki.linuxfoundation.org/realtime/start, Jan 2020. [Online].

[Rei16] P. Reichenpfader. Towards certification of software-intensive mixed-critical systems in automotive industry. In *Critical Automotive applications : Robustness & Safety (CARS)*, 2016.

[SAdOdO18] K. P. Silva, L. F. Arcaro, D. B. de Oliveira, and R. S. de Oliveira. An Empirical Study on the Adequacy of MBPTA for Tasks Executed on a Complex Computer Architecture with Linux. In *IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 321–328. IEEE, 2018.

[Sal10] N. J. Salkind. *Encyclopedia of research design*, volume 1. Sage, 2010. doi:10.4135/9781412961288.

[Sam02] G. Sampson. *Empirical Linguistics*. A&C Black, 2002.

[SBG14] C. Stangl, A. Braun, and M. P. Geyer. GECCOS-the new Monitoring and Control System at DLR-GSOC for Space Operations, based on SCOS-2000. In *SpaceOps Conference*, page 1602, 2014. doi:10.2514/6.2014-1602.

[Sha48] C. E. Shannon. A Mathematical Theory of Communication. *The Bell system technical journal*, 27(3):379–423, 1948.

[Sim20] M. Simunovic. CallGraph Tool – How (not) to develop a call graph detection tool. In *ELISA Workshop*, May 2020.

[spa] ELC: SpaceX lessons learned. Available: https://lwn.net/Articles/540368/. [Online].

[Ste10] F. H. Stephenson. Chapter 3 – cell growth. In F. H. Stephenson, editor, *Calculations for Molecular Biology and Biotechnology (Second Edition)*, pages 45–81. Academic Press, Boston, second edition edition, 2010. doi:10.1016/B978-0-12-375690-9.00003-6.

[sus] Trust SUSE US Federal Government Solutions. Available: https://www.suse.com/sector/federal/. [Online].

[SvB84] E. P. Smith and G. van Belle. Nonparametric Estimation of Species Richness. *Biometrics*, pages 119–129, 1984. doi:10.2307/2530750.

[syz] Syzkaller – kernel fuzzer. Available: https://github.com/google/syzkaller. [Online].

[tesa] NHTSA investigating 'violent' Tesla crash, Autopilot not ruled out yet. Available: https://www.cnbc.com/2021/03/15/nhtsa-investigating-violent-tesla-crash-autopilot-not-ruled-out-yet.html. [Online].

[tesb] Tesla driver dies in first fatal crash while using autopilot mode. Available: https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk. [Online].

[TKZS16] Ö. Ş. Taş, F. Kuhnt, J. M. Zöllner, and C. Stiller. Functional System Architectures towards Fully Automated Driving. In *IEEE Intelligent vehicles symposium (IV)*, pages 304–309. IEEE, 2016. doi:10.1109/IVS.2016.7535402.

[TLW$^+$16] M. Thomas, M. Lemaitre, M. L. Wilson, C. Viboud, Y. Yordanov, H. Wackernagel, and F. Carrat. Applications of extreme value theory in public health. *PloS one*, 11(7):e0159312, 2016. doi:10.1371/journal.pone.0159312.

[ubea] Self-driving Uber car that hit and killed woman did not recognize that pedestrians jaywalk. Available: https://www.nbcnews.com/tech/tech-news/self-driving-uber-car-hit-killed-woman-did-not-recognize-n1079281. [Online].

[ubeb] Self-driving Uber kills Arizona woman in first fatal crash involving pedestrian. Available: https://www.theguardian.com/technology/2018/mar/19/uber-self-driving-car-kills-woman-arizona-tempe. [Online].

[Vek] P. O. Veksler. Lecture notes of: Artificial Intelligence II. URL https://www.csd.uwo.ca/~oveksler/Courses/Winter2009/CS4442_9542b/L9-NLP-LangModels.pdf.

[WEG08] E. White, B. Enquist, and J. Green. On estimating the exponent of Power-law frequency distributions. *Ecology*, 89:905–12, 05 2008. doi:10.1890/07-1288.1.

[WH16] D. Watzenig and M. Horn. *Automated Driving: Safer and More Efficient Future Driving*. Springer, 2016.

[WM08] V. M. Weaver and S. A. McKee. Can hardware performance counters be trusted? In *IEEE International Symposium on Workload Characterization*, pages 141–150. IEEE, 2008. doi:10.1109/IISWC.2008.4636099.

[WTM13] V. M. Weaver, D. Terpstra, and S. Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 215–224. IEEE, 2013. doi:10.1109/ISPASS.2013.6557172.

Glossary of Terms

Common-traces

Execution traces that occur frequently. Thus, traces that have a higher probability of being executed..

Harm

"Physical injury or damage to the health of people or damage to property or the environment" [IEC61508].

Kernel

Software that constitutes the core part of an operating system. It is responsible for resource managing and enables the communication between applications and hardware..

Multi-core

Integrated circuit formed by two or more processors units.

Non-parametric statistics

A branch of statistics that encompasses methods that make no assumptions about the underlying distribution of the data.

Parametric statistics

A branch of statistics that encompasses methods that assume that the underlying population of a data set can be described with a model with a fixed set of parameters.

Path

Sequence of invoked function calls (equivalent to trace)..

Rare-traces

Execution traces that occur infrequently. Thus, traces that have a lower probability of being executed..

Residual risk

"Risk remaining after protective measures have been taken" [IEC61508].

Risk

"Combination of the probability of occurrence of harm and the severity of that harm" [IEC61508].

Safety standard

Standards designed to guarantee tolerable safety providing guidance for certification process.

Safety-related system

"Designated system that both: implements the required safety functions necessary to achieve or maintain a safe state for the EUC; and is intended to achieve, on its own or with other E/E/PE safety-related systems and other risk reduction measures, the necessary safety integrity for the required safety functions" [IEC61508].

Test coverage

Measurement of the source code degree that has been tested.

Test-campaign

Grouping of repeated runs of a specific application For example, a group of a thousand repeated runs of "Hello World".

Trace

Sequence of invoked function calls (equivalent to path)..

Unique trace

A specific sequence of functions that corresponds to one or more executions..

Appendix A

Additional case study

We present this annex with the intention of showing the potential of the proposed SPC technique. Although the results presented throughout the dissertation have shown favorable results for the technique, it is true that they are based on a single case study. Therefore, this chapter extends the research with an additional case study with aim of enhancing the confidence in the technique for future use cases.

A.1 Description

A simple, yet reproducible, case study is used to perform the analysis. This simple case study, which we named *randbyte*, also eases the analysis and explanation of obtained results without detailed knowledge of a given specific application. This application starts opening */dev/urandom* and */dev/null*, follows reading data from */dev/urandom* and writing this data in */dev/null*. Finally, it finishes closing both devices.

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define DEV1 "/dev/urandom"
#define DEV2 "/dev/null"

int main(int argc, char **argv)
{
  unsigned char result;
  int fd1, fd2, ret;
  char res_str[10] = {0};

  fd1 = open(DEV1, O_RDONLY);
  fd2 = open(DEV2, O_WRONLY);

  ret = read(fd1, &result, 1);
  sprintf(res_str, "%d", result);
  ret = write(fd2, res_str, strlen(res_str));

  close(fd1);
  close(fd2);

  return ret;
}
```

Although this application is not representative of the next-generation safety-related use cases for which the SPC technique is intended, this case study allows us to have greater control and understanding of what is happening in the system.

A.2 Experiment Setup

This section describes the setup of the experiments we have carried out to evaluate the technique described in Chapter 8. The Ultra96-v2 is a COTS platform based on the Xilinx Zynq UltraScale+ MPSoC that integrates a quad-core Cortex-A53 CPU, a dual-core Cortex-R5, and a Field Programmable Gate Array (FPGA). The analysis is performed using only the quad-core Cortex-A53 CPU where Linux runs. Hence, neither the FPGA nor the Cortex-R5 processors are used in the experimental setup. The selected Linux version is the current latest CIP SLTS Linux version, v4.19.75-cip11. The image is built with the default mainline arm64 *defconfig* plus FTrace (including the function graph tracer option).

In addition, this application has been run with a high CPU and interrupts load, just as it has been done in the use case presented in Chapter 4. The implemented stress benchmark exercises the kernel in the corner cases to trigger worst-case behavior as well as covering the usual type of requests. For the analysis presented in this publication, *hackbench* is configured to create 400 tasks, where each send passes 100k messages of 100 bytes. This generates almost 100 % CPU load for longer than the data set collection, in order to maintain the high CPU load during all the data recording process. Contrary to the case study described in Chapter 4, the SIL2LinuxMP architecture has not been implemented (see Section 4.2.1). Therefore, in this case, the application suffers a higher interference from the stress benchmarks. High CPU/IRQ load scenarios are used to increase the probability of occurrence of rare paths, however, these paths are still independent of one another. Therefore, we can assume that the dispersion of the paths is constant under a high CPU/IRQ load.

A.3 Data collection

For this case study we collect six system calls (i.e., *open(urandom), open(null), read(urandom), write(null), close(urandom), close(null)*). Note that this application also exercises additional system-calls, for instance, open system-calls for the libraries. However, we only focused on these six primary system-calls.

Figure A.1 shows the *unique traces* that have been recorded in each test-campaign. The analysis has been performed with a total of 87 test-campaigns.

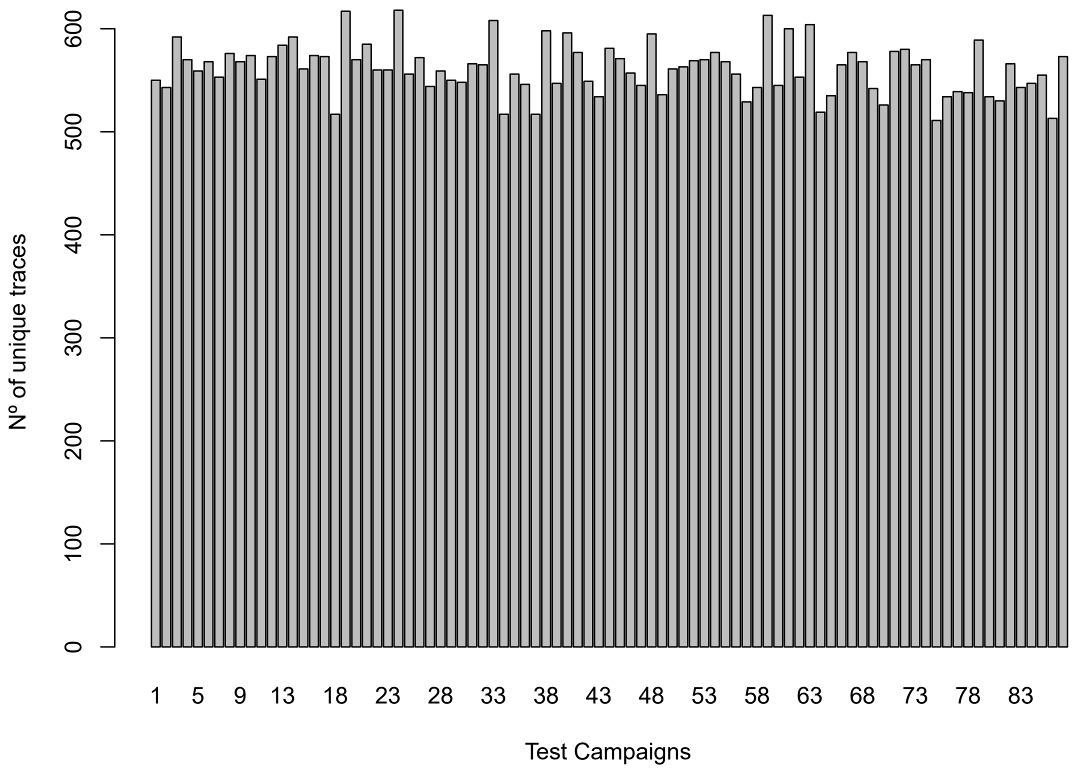


Figure A.1 Number of unique traces in each test-campaign.

Table A.1 collects the unique traces that have been recorded in the whole data set, which is formed of 87 test-campaigns. The results are provided by system-call. As with the use case of Chapter 4, the results show a variability difference between the system-calls. Some of them exercise a higher number of traces than others. For instance, *open()* system-calls show a significantly higher number of traces than the rest of system-calls.

Table A.1 Recorded unique traces in each system-call.

	open(null)	**open(urandom)**	**read**	**write**	**close(null)**	**close(urandom)**
unique traces	7768	8747	776	263	242	250

Once we have a data set, we proceed to validate it as it is indicated in Figure 8.1. The quantity validation is performed with EVT analysis. EVT analysis shows some stability in the results. As shown in Table A.2, we do not expect to see a significant variation of the results if we had collected 500, 1000, or 10000 test-campaigns. Therefore, we can conclude that the data set size is enough to follow the analysis.

Table A.2 Return level of the EVT analysis for the data set formed with 87 test-campaigns.

	2.5%	**Estimate**	**97.5%**
500-Test	594.4	606.3	619.8
1000-Test	595.1	608	623.8
10000-Test	596.2	611.6	633.1

From the quality perspective, both the Ljung-box test and the Kolmogorov-Smirnov test show that the data that form the data set are i.i.d. Consequently, we have a validated data set that can be used for test coverage analysis.

A.4 Test coverage

Figure 8.1 defines the steps that need to be performed in order to estimate the test coverage. Briefly, the steps are:

1. Model the appearance of untested traces with Poisson.
2. Calculate the total number of traces.
3. Verify the results with non-parametric estimators.

Hence, to estimate the test coverage we proceed to model the occurrence of rarely occurring traces as defined in Chapter 5. We use Poisson regression to perform this task. Figure A.2 shows the obtained model with cut-off frequency of 4. The graph is depicted with the 95% and 50% CIs estimated with bootstraping.

Once we have the models we calculate the improper integral at infinity $\int_1^\infty e^{\beta_0+\beta_1 x}dx$. Table A.3 provides the results for the whole data set (*All*). Table A.3 also provides the results for each system-call. This means that the analysis is performed with each system-call data too, which involves modeling the data for each of the system-calls. This allows us to perform system-call analysis as well, to see if the technique fits with reduced data sets and to see the difference in test coverage depending on the system-call. Test coverage results can be considered low for the application. However, this varies depending on the system-call. There are system-calls that have a significantly higher coverage than others. Besides, as Table A.3 shows, the sum of total traces of each system-call is approximately the same as the one obtained modeling the whole data set. Thus, this is another indicator that enhances the confidence in the method.

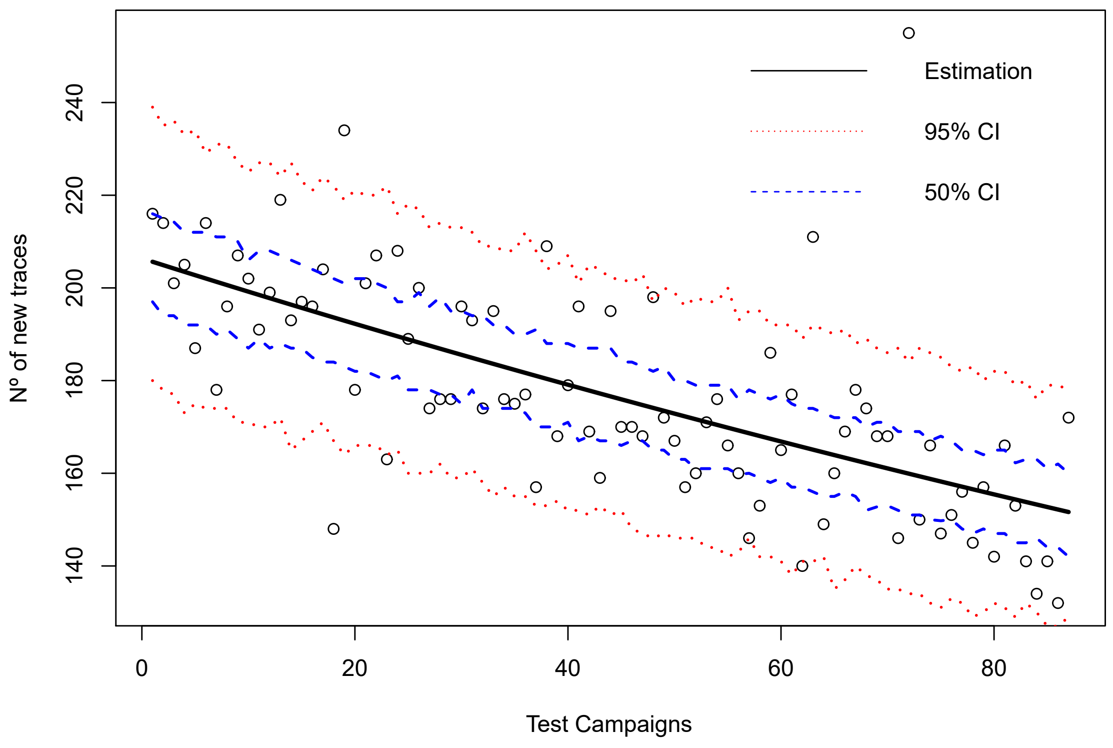


Figure A.2 Number of new *rare-traces* identified by each test-campaign.

Table A.3 Test coverage obtained with the parametric method.

Data set	Cut-off	Recorded traces	Total traces	Standard Error	Test coverage (%)	Dispersion
All	4	16654	74710	718.87	22.29	1.92
open(null)	4	7769	30733	334.65	25.27	1.38
open(urandom)	5	8748	45559	537.09	19.2	1.07
read	6	777	1528	14.79	50.85	0.99
write	8	264	431	4.62	61.25	1.05
close(null)	6	243	349	3.34	69.62	1.07
close(urandom)	7	251	347	2.55	72.33	1.00

Table A.3 also shows that in all studies cases the Poisson model fits correctly. Either the data set that collects all the system-calls or each system-call independently offers a dispersion value close to 1. Therefore, we can verify that the Poisson model fits correctly. Additionally, as defined by the SPC technique in Chapter 8, the result of the total number of traces is verified with non-parametric estimators. Table A.4 collects the results obtained with non-parametric results for the whole data set and each system-call. Table A.4 collects also the total number of traces calculated with the parametric method in order to make the verification.

In all the cases, without exception, the obtained result is in the range of non-parametric results. Consequently, we can rely on the results.

Table A.4 Estimated total number of traces and percentage of unseen traces.

Data set	Param	Chao2	Jackknife 1	Jackknife 2	Bootstrap	Chao1	ACE
All	74710	119868	30429	43147	21895	119999	118959
open(null)	30733	60932	14063	19921	10161	61604	49740
open(urandom)	45559	71495	16211	23154	11576	71924	77005
read	1528	2340	1274	1688	973	2310	2465
write	431	702	411	532	322	688	697
close(null)	349	530	364	459	292	520	469
close(urandom)	347	513	373	466	301	505	484

With the current case study, we can extract some relevant data related to the estimators. In Chapter 8, we argue the main role of the parametric approach to estimate test coverage instead of the non-parametric estimators. The reason for this is that we could not be sure of the behavior of the estimators in other use cases. For example, the most suitable estimator for a particular use case could be Chao 2, but for another use case could be Jack 1. With the data we have obtained from the use case presented in this appendix, we observe that the indicators do not behave in the same way as in the AEB use case. In Figure 8.2 of Chapter 8, we observe that Chao 2 and Jack 2 offer very similar results. However, in this case, this is not the case. Therefore, this reinforces the argument for defining the estimators as a form of verification for the SPC and not as final coverage results.

A.5 Residual risk estimation

Finally, as the SPC technique defines, we focus on the estimation of the residual risk that the untested traces entail. As it is stated by the technique, we model the data set with power law as it is depicted in Figure A.3.

Figure A.3 shows how the model fits adequately to the rare-traces. The graph shows the number of traces that have occurred once, twice, etc. With this model, we can begin to calculate the probability of execution of the untested. Simple Good-Turing equation is used to calculate the execution probabilities. The equation provides the results presented by Table A.5.

Once we obtain the execution probability of untested paths (P_{zero}) we can estimate the risk that they entitle. For this purpose, we follow the points defined in Section 7.4. We need the execution frequency of the task. So, for instance, if the application is executed with

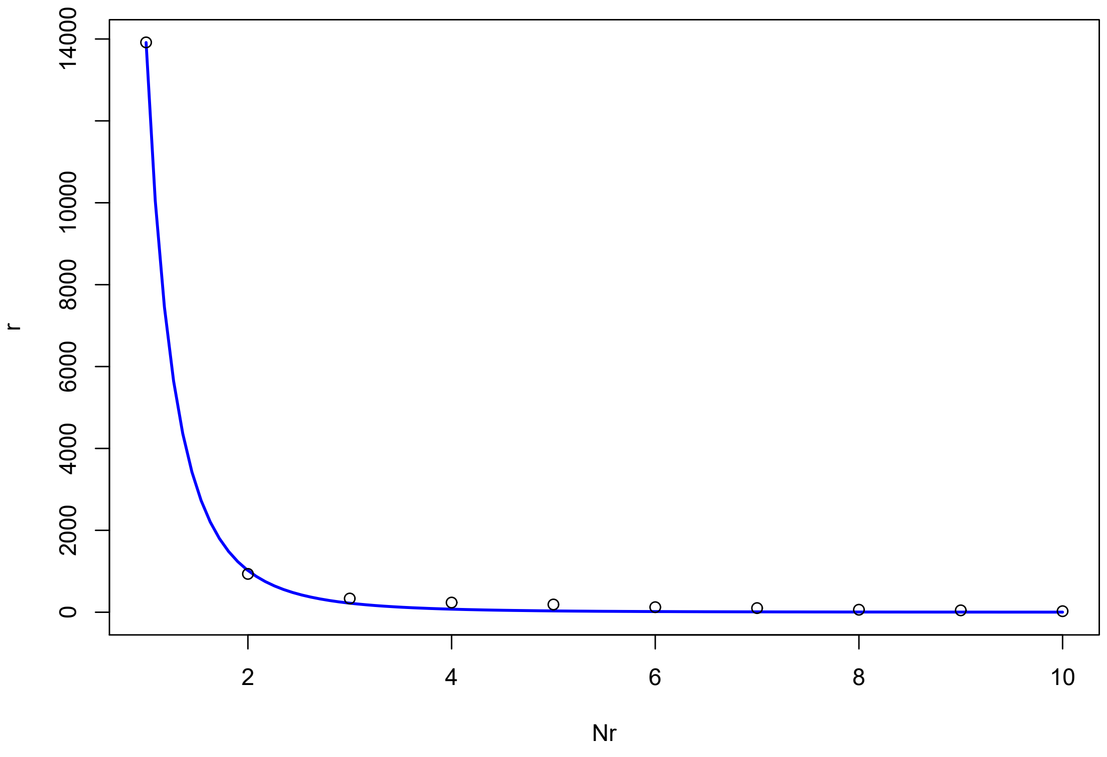


Figure A.3 Power law model for the recorded data set.

Table A.5 Execution probability results for each rate.

rate	probability	rate	probability
0	$5.344141 \cdot 10^{-3}$	6	$2.303663 \cdot 10^{-6}$
1	$5.623670 \cdot 10^{-8}$	7	$2.687606 \cdot 10^{-6}$
2	$2.498084 \cdot 10^{-7}$	8	$3.071550 \cdot 10^{-6}$
3	$1.640013 \cdot 10^{-6}$	...	...
4	$1.535775 \cdot 10^{-6}$	760438	$2.919654 \cdot 10^{-1}$
5	$1.919719 \cdot 10^{-6}$	861956	$3.309426 \cdot 10^{-2}$

a period of 100 milliseconds, we execute 6 system-calls every 100 milliseconds. In other words, $216 \cdot 10^3$ calls per hour.

$$Probability = 1 - \binom{Freq}{0} \cdot {P_{zero}}^2 (1 - {P_{zero}}^2)^{Freq} = 0.9979067 \tag{A.1}$$

Therefore, the probability is practically 1. So, basically, the risk is high although currently there is no predefined limit. This was already indicated by the low percentage of test coverage. In order to reduce the risk, the SPC technique recommends testing the system further.

Dissemination Activities

<ins>JOURNAL PUBLICATIONS:</ins>

I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve, J. Petersohn and R. Obermaisser. "Statistical test coverage for Linux-based next-generation autonomous safety-related systems", in IEEE Access, 2021, doi: 10.1109/ACCESS.2021.3100125.

I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve and R. Obermaisser. "Towards Linux based safety systems - A statistical approach for software execution path coverage", in Journal of Systems Architecture, 2021, doi: 10.1016/j.sysarc.2021.102047.

J. Perez , R. Obermaisser, J. Abella, F. J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, and **I. Allende**. "Multi-core Devices for Safety-critical Systems: A Survey", in ACM Computer Survey, 2020, doi: 10.1145/3398665

J. Fernández, J. Perez, I. Agirre, **I. Allende**, J. Abella, F. J. Cazorla. "Towards Functional Safety Compliance of Matrix-Matrix Multiplication for Machine Learning-based Autonomous Systems", in Journal of Systems Architecture.

<ins>CONFERENCE PUBLICATIONS:</ins>

I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve and R. Obermaisser. "Estimation of Linux kernel Execution Path Uncertainty for Safety Software Test Coverage", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2021, pp. 1446-1451, doi: 10.23919/DATE51398.2021.9473951.

I. Allende, N. Mc Guire, J. Perez, L. G. Monsalve, N. Uriarte and R. Obermaisser. "Towards Linux for the Development of Mixed-Criticality Embedded Systems Based on Multi-Core Devices", 15th European Dependable Computing Conference (EDCC), Naples, Italy, 2019, pp. 47-54, doi: 10.1109/EDCC.2019.00020.

N. Mc Guire and **I. Allende**. "Approaching certification of complex systems", 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Valencia, Spain, 2020, pp. 70-71, doi: 10.1109/DSN-W50199.2020.00022.

C. Hernandez, J. Flieh, R. Paredes, C. Lefebvre, **I. Allende**, J. Abella, D. Trillin, M. Matschnig, B. Fischer, K. Schwarz, N. Mc Guire, F. Rammerstorfer, C. Schwarzl, F. Wartet, D. Lüdemann, M. Labayen. "SELENE: Self-Monitored Dependable Platform for High-Performance Safety-Critical Systems", 23rd Euromicro Conference on Digital System Design (DSD), Kranj, Slovenia, pp. 370-377, doi: 10.1109/DSD51259.2020.00066.

AWARD:

Ada-Europe 2021: The Best Presentation Award. "Towards Linux based safety systems - A statistical approach for software execution path coverage".