Journal of Systems Architecture 169 (2025) 103598

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Linux for safety-critical systems: A survey

Markel Galarraga [a,b,*], Charles-Alexis Lefebvre [a], Jon Perez-Cerrolaza [a], Jose A. Pascual [b]

[a] Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), Arrasate/Mondragon, Spain
[b] Faculty of Informatics, University of the Basque Country (UPV/EHU), Donostia-San Sebastián, Spain

ARTICLE INFO

Keywords:
Linux
Functional safety
Safety-critical systems
Mixed-criticality
Literature review

ABSTRACT

Next-generation safety-critical systems, such as autonomous vehicles, are increasingly complex systems integrating high-performance computing devices, diverse software stacks, machine learning algorithms and software applications of different safety criticality. Industry and academia are showing growing interest in using Linux as a general-purpose operating system in these safety-critical systems due to its widespread adoption in embedded systems and critical domains (e.g., telecommunications, banking) and widespread support of computing devices, software stacks and machine learning software. However, meeting the requirements of safety standards, such as systematic error reduction techniques, random fault tolerance, and temporal and spatial independence, becomes a challenge. This is especially the case when integrating software applications of different safety criticality (mixed criticality). This literature survey examines works that propose, analyze and extend Linux for the development of safety-critical systems. We also identify the main challenges these works focus on. Finally, we also present an overview of the main industry efforts.

Contents

* Corresponding author at: Ikerlan Technology Research Centre, Basque Research and Technology Alliance (BRTA), Arrasate/Mondragon, Spain.
E-mail addresses: mgalarraga@ikerlan.es (M. Galarraga), calefebvre@ikerlan.es (C.-A. Lefebvre), jmperez@ikerlan.es (J. Perez-Cerrolaza), joseantonio.pascual@ehu.eus (J.A. Pascual).

https://doi.org/10.1016/j.sysarc.2025.103598
Received 22 July 2025; Received in revised form 19 September 2025; Accepted 10 October 2025
Available online 17 October 2025

1. Introduction

In recent years, diverse industry sectors have invested in pioneering safety-related systems such as autonomous vehicles and automatic train operation systems, which integrate high-performance computing devices (e.g., multi-core processors, GPUs), diverse software stacks (e.g., ROS, CUDA), and complex software applications (e.g., AI algorithms) of different safety criticality (i.e., mixed-criticality systems).

However, safety-critical systems composed of programmable electronic systems (electronics/hardware and software) need to be developed adhering to strict safety certification standards because their failure can have catastrophic consequences for humans or the environment. Two examples of such standards are the generic industrial IEC 61508 [1] or the automotive ISO 26262 [2]. Therefore, the underlying Operating Systems (OSs) should facilitate the integration of pioneering safety-related systems in compliance with applicable functional safety standards [3–5].

In that regard, both industry and academia have shown a growing interest in using Linux for safety-critical systems [6]. One reason for this is that Linux OS is the leading OS across a broad spectrum of computing domains, from embedded systems to supercomputers [7]. In addition, it has already been used in critical applications such as telecommunication and banking, as well as in dependable systems like spacecrafts (e.g., SpaceX Falcon 9, Dragon) [6]. Moreover, its open-source development model has been considered potentially suitable for certification under the IEC 61508 functional safety standard [8].

In this work, we review academic literature on the use of Linux in safety-critical systems. For that purpose, we classify contributions according to the intended safety-related usage of Linux.

(1) *Linux as a GPOS:* Linux is the General-Purpose Operating System (GPOS) that executes non-safety-related general-purpose software applications within a safety-critical system. The key challenge is ensuring the required isolation and independence from the safety-critical part of the system.
(2) *Linux as a SCOS:* Linux is the Safety-Critical Operating System (SCOS) that aims to support the safe execution of safety-related and mixed-criticality software applications. The key challenge is ensuring compliance with safety standard requirements such as timing behavior, diagnosis, fault tolerance, and temporal and spatial independence.

Alongside the academic literature review, we also survey key industrial initiatives, such as the Linux Foundation's ELISA (Enable Linux In Safety Applications) project, EB Corbos Linux for Safety Applications, and RedHawk Linux RTOS.

The rest of this paper is organized as follows: Section 2 presents the main concepts needed to understand this work. In Section 4, we explain the methodology and taxonomy followed for the literature review. In Section 5, we begin with the literature review, presenting the works that use Linux as a GPOS in a mixed-criticality system. Section 6 includes the works that use Linux as a SCOS. Section 7 presents the main industrial approaches focusing on including Linux in safety-critical systems. In Section 8, we summarize the main challenges extracted from the literature review. Finally, Section 9 concludes the paper.

2. Background

This section explains the basic concepts necessary to understand our work.

2.1. Safety-related systems

A safety-related system is the electrical, electronic, or programmable electronic part of a system designed to ensure the safety of its context by reducing the risk of danger to an acceptable level through functional safety principles. Functional safety specifically addresses the aspect of safety ensured by the correct operation of electrical, electronic, and programmable electronic systems. It aims to prevent exposing the context to an "unacceptable risk" by reducing the risk of danger to an acceptable level. Safety is also defined as "absence of catastrophic consequences on the user(s) and the environment" [9].

Functional safety standards define the requirements, techniques, and methods for risk assessment for safety-related systems, depending on their criticality levels. The standard IEC 61508 defines safety as "freedom from unacceptable risk" (IEC 61508-4 Ed2 Clause 3.1.11). On the other hand, the standard ISO 26262 defines safety as the "absence of unreasonable risk" (ISO26262-1 Ed1 Clause 1.103). Both standards pose a risk as a quantifiable measure and define "safety" in terms of reducing risk to below an acceptable level. This means that no part of this context should be exposed to an "unacceptable risk" induced by the system. For example, in the case of an autonomous vehicle, the "freedom from unacceptable risk" applies to various stakeholders in the context: (i) people and animals inside the vehicle (passengers, driver if applicable), (ii) people and animals outside the vehicle (pedestrians, cyclists, other drivers), (iii) infrastructure and environment (other vehicles, road structures, properties, wildlife, etc.).

Moreover, each functional safety standard defines criticality levels to maintain the risk below an acceptable level. For example, IEC 61508 defines Safety Integrity Levels (SIL) from 1 (lowest criticality) to 4 (highest criticality). Automotive ISO 26262 defines similar Automotive Safety Integrity Levels (ASIL) from A (lowest criticality) to D (highest criticality). In avionics, RTCA DO-178C defines Design Assurance Levels (DAL), from DAL E (lowest criticality) to DAL A (highest criticality).

However, many of these standards were established over 15 years ago, and the safety requirements, techniques, and methods have not evolved with newer technology. Compared to state-of-the-art safety-related systems, the growing complexity of modern systems (such as larger codebases, more peripherals, and additional functionalities) poses challenges for traditional risk assessment methods. These modern systems are composed of multiple interacting components that have intricate interdependencies and behaviors, such as external communications, updating needs due to security concerns, adaptive learning mechanisms (e.g., Machine Learning), and multi-layered software architectures. Autonomous vehicles are a widespread example of these. Consequently, safety standards are increasingly difficult to apply to state-of-the-art safety-related complex systems.

2.2. Linux and functional safety

Linux is an open-source OS that dominates most non-safety-related applications across various sectors, from smartphones and embedded systems to cloud computing and supercomputers [6,7]. Despite not having been developed to meet safety standards, Linux benefits from a vast ecosystem with extensive hardware and software support, which has led to its widespread adoption. Using Linux instead of a proprietary OS can (i) reduce development and deployment costs, as there are no licensing fees, (ii) increase confidence due to its open-source nature, making it potentially more resilient to vulnerabilities, and (iii) provide access to a strong community and support, given that many professionals in computer science have experience with this OS. As a result,

the industry is increasingly interested in adopting Linux for safety-related systems. Together with the industrial interest, there has also been an increasing interest from academia. In that regard, as described in the introduction, we have identified the works that focus on using Linux in safety-critical systems, and classified them in two main groups, according to the role of Linux in the system: (1) Linux as a GPOS, and (2) Linux as a SCOS.

It is important to note that no work has tried to make a specific Linux version suitable for functional safety. Linux is not a final product, but rather an evolving project. As such, it cannot be certified as a standalone component for use in any safety-related project. Each system that incorporates Linux must be justified and certified as a whole. Additionally, projects that aim to enable the use of Linux in safety-critical systems do not only focus on the Linux kernel but also consider libraries, tools, and all other elements that constitute the OS.

Safety standards define specific development processes, documentation practices, specification methods, and verification and testing activities that must be applied depending on the required SIL. The higher the SIL, the stricter these requirements become. For example, at the highest integrity levels (e.g., SIL4 in IEC 61508, ASIL D in ISO 26262, or DAL A in DO-178C), standards require:

- Rigorous specification and traceability, as every safety requirement must be formally specified and traceably linked through design, implementation, verification, and validation artifacts.
- Development activities must follow a well-defined and documented safety lifecycle, with roles, responsibilities, and independence in reviews and assessments.
- Complete verification and testing, with requirements such as Modified Condition/Decision Coverage (MC/DC) in DO-178C, systematic fault injection, and exhaustive testing.
- Qualification and evidence of development and verification tools.
- Strict control over versioning, change impact analysis, and re-verification.

Meeting these requirements with an open-source OS developed by the community and ever evolving, such as Linux, is a challenge. The Linux kernel is made up of more than 40 million lines of code, is developed by thousands of contributors around the globe, and has no centralized control over processes, documentation, or design artifacts. As a result, the traceability, process assurance, and systematic verification demanded by high-integrity levels in safety standards are fundamentally absent. This explains why achieving certification for a Linux-based safety-critical system is an open challenge being tackled by both the academy and the industry.

Finally, using Linux in safety-critical systems responds to an industrial need to use increasingly complex safety-critical systems. Therefore, the interest in using Linux responds to the need to use platforms, libraries, programs, etc., that can also support said complexity. In that regard, other authors have conducted literature reviews on different aspects of these systems, which we overview in Section 3.

2.3. Linux and real-time

Safety-related systems must exhibit predictable behavior, which also means they need to be time-predictable. IEC 61508 suggests using deterministic scheduling methods and strict priority-based scheduling to ensure temporal independence (IEC 61508-3 Ed2 Annex F.5). This requirement is why many safety-critical systems have real-time constraints. For example, a car's Anti-Lock Braking System (ABS) must activate within a specific time window [4]. Consequently, SCOSs are Real-Time Operating Systems (RTOSs), but not all RTOSs are SCOSs.

A real-time system is designed to ensure that outputs are delivered within specified deadlines, prioritizing reliability over speed. It is important to note that being a real-time system does not necessarily mean a fast system; rather, it is about meeting the deadlines. A real-time system can be categorized based on its compliance with deadlines and the consequences of non-compliance. In a *soft real-time system*, results may continue to be produced and used even after the designated deadline, yet with a degradation of the Quality of Service (QoS). While this flexibility allows for a higher uptime, overall system performance may suffer. A *firm real-time system* operates with a more constrained approach: if the system fails to deliver a result on schedule, it will not produce it since the output cannot be used after the deadline. However, this failure does not have serious consequences, which means that, although reliability is essential, the system tolerates missed deadlines without critical impacts. In contrast, a *hard real-time system* is characterized by inflexible deadlines. Meeting these strict deadlines is imperative, as any failure to meet them could lead to critical failures. This is why real-time safety-related systems are classified as hard real-time systems [10].

The vanilla Linux kernel is not an RTOS, but there are several approaches that transform it into one. As presented throughout this survey, some methods utilize a microkernel or a co-kernel that runs alongside the Linux kernel, while others, like the PREEMPT_RT patch, modify the kernel to make it real-time. This patch has been integrated into the mainline Linux kernel starting from version 6.12. Further details about methods to transform Linux into an RTOS can be found in the surveys conducted by Reghenzani et al. [10] and by Struhár et al. [11]. The former places greater emphasis on the PREEMPT_RT patch, whereas the latter concentrates on real-time container-based virtualization in Linux.

3. Related work

While Linux is the central part of this survey, all the parts of safety-critical systems play a role in their dependability and certifiability. In addition to the OS, the underlying hardware architecture, accelerators such as GPUs, middleware, AI frameworks, etc., play their own role and have their own challenges in safety-critical systems. Due to the increasing complexity of the safety-critical systems that the industry wants to build, these aspects have been thoroughly studied by the academy in recent years. In that regard, this section gathers other surveys that have explored these aspects and offer complementary perspectives that highlight the need to consider systems as a whole when finding solutions in safety-critical contexts.

Perez-Cerrolaza et al. [4] categorize and provide an overview of research contributions addressing fundamental safety requirements at different device abstraction levels (nanoscale, component, and device). They explain how safety certification of multi-core systems is a challenge and focus on aspects such as temporal and spatial independence, reliability, and diagnostic coverage, which are fundamental safety technical requirements.

Regarding GPUs, the survey by Perez-Cerrolaza et al. [5] categorizes and provides an overview of research addressing GPU devices' random hardware failures, systematic failures, and independence of execution. It emphasizes the challenges of integrating complex, parallel, and computationally demanding software functions with different safety-criticality levels on GPU devices with shared hardware resources.

The use of hypervisors in safety-critical systems has been reviewed by Lozano et al. [12]. They offer a comprehensive review of the works that use hypervisors as the base of their safety-critical systems, collect and categorize the information of each hypervisor, and compare them to each other.

Besides surveys of academic works, industrial surveys have also been carried out, where professionals have been interviewed. The survey by Kassab [13] gathers the testing practices for safety-critical software in the industry, while Pedersen et al. [14] identify and explore the main industrial needs and challenges.

There have also been surveys in specific industrial domains. As will be presented throughout this work, automotive and aerospace are the main domains for works that focus on the use of Linux for

Table 1
Results of the search for works.

Library	Search parameters	Found
ACM	Title, abstract, keywords	59
Google Scholar	Title	20
IEEE	Title, abstract, keywords	356
Inspec	Title, abstract, keywords	398
Science Direct	Title, abstract, keywords	29
Scopus	Title, abstract, keywords	703
Springer Link	Title	77
Web of Science	Title, abstract, keywords	290
Total		*1932*
Removing duplicates		*1336*

safety-critical systems, and this is not particular to Linux but to safety-critical systems in general. Surveys that focus on safety-critical systems for these domains can, therefore, be found. For automotive, Paden et al. [15] study the state of the art on planning and control algorithms for urban vehicles and compare approaches side by side; while Rabe et al. [16] explore the methodologies for the development of machine learning automotive applications in order to give an overview of the state of research and identify the main challenges. In the avionics domain, Boglietti et al. [17] study the electrification of safety-critical drives of aircraft, flight surface actuators, fuel pumps, and generators, and present the main technical challenges and research topics that derive from it. Finally, Perez-Cerrolaza et al. [3] study the development of AI-based safety-critical systems in the industrial and transportation domains. Specifically, they gather the works that address the challenge of joining state-of-the-art AI with safety standards, and they analyze the challenges, techniques, and methods for developing AI-based safety-critical systems.

Although they answer to different standards, safety and security are strongly linked, especially for today's safety-critical systems. As such, some authors have studied the security aspect of safety-critical systems. Kriaa et al. [18] survey the industrial and academic approaches to industrial facility design and risk assessment that consider both safety and security. Lisoba et al. [19] provide a literature review of safety and cybersecurity co-analysis methods, focusing on early system development stages. Kavallieratos et al. [20] extend the previous literature review by identifying additional methods of safety and cybersecurity co-engineering, thus studying the recent advances in the field. Works agree on the need for safety and security co-engineering in state-of-the-art safety-critical systems, and on the fact that more work is needed toward methodologies that consider them both. In that regard, it is important to note that, in this survey, we choose to focus only on the safety aspect of Linux-based safety-critical systems, and not on the security aspect, due to the fact that safety and security are still nowadays two separate — but interconnected — aspects of such systems, with different goals and answering to different certifications, especially when the focus is on Linux.

Shifting back to Linux, its use in safety-critical systems is inevitably connected to some other aspects, such as security, real-time behavior, or isolation. Some authors have surveyed the literature focusing on those aspects. Procopio [21] gathers studies and projects that focus on combined safety and security analysis, presents standards that consider both, lists RTOSs that comply with safety and security certification, and finally argues for projects that intend to comply with safety certification with their Linux-based systems to also consider security certification. Regarding the real-time behavior of Linux, the survey by Reghenzani et al. [10] gathers the state-of-the-art approaches for building real-time Linux systems, with a special focus on the PREEMPT_RT patch. Struhár et al. [11], on the other hand, focus on the real-time aspect of Linux together with isolation, by reviewing the literature around the efforts to bring real-time properties to Linux container-based virtualization.

When compared to these surveys, the novelty of our work resides in the fact that we review the literature that has used or intends to use Linux in safety-critical systems, identify the challenges Linux faces for its use in safety-critical systems, and categorize the works according to the role of Linux in their system and the challenge they focus on. In addition to the literature review, we include a section with an overview of the main industrial efforts aiming for Linux-based safety-critical systems, and connect them to the identified challenges.

4. Methodology and taxonomy

4.1. Search methodology

The methodology for this literature review consists of searching for works in the following libraries: ACM, Google Scholar, IEEE, Inspec, Science Direct, Scopus, Springer Link, and Web of Science. The search query was "Linux AND safety" and we searched it in the title, abstract, and keywords. We could not use those search parameters in Google Scholar and Springer Link, so we searched only in the title instead. We chose to search for "safety" and not "functional safety" or "safety-critical" because we have found different ways to refer to it in the works, such as "safety systems" or "safety-related systems". Therefore, we needed a word to include them all, despite knowing we would get many false positives.

The results are presented in Table 1. The search was conducted in March 2025. As can be seen, we found 1336 works that included the search terms. We screened through these works to identify works that use Linux in safety-critical systems, eliminating those that do not. For example, we found some works that focus on memory safety and others that, despite focusing on functional safety, mention Linux in the work but do not intend to use it in a safety-critical system. As explained in Section 3, we also chose not to include works that focus on the security aspect. After screening through the works, we were left with 102 of them.

4.2. Taxonomy

This section presents the taxonomy used in this survey to classify the reviewed works. As previously explained, the classification is based on the intended usage of Linux:

(1) *Linux as a GPOS:* Linux is the GPOS that executes non-safety-related general-purpose software applications within a safety-critical system.
(2) *Linux as a SCOS:* Linux is the SCOS that aims to support the safe execution of safety-related and mixed-criticality software applications.

After the first classification, each category is further divided. The classification of the works is summarized in Table 2. In the first case, where Linux is used as a GPOS, we further categorize the methods for isolating Linux from the safety-critical components of the system:

(1) *Hypervisors* provide a virtualization layer that allows multiple OS or bare-metal programs to run concurrently on the same hardware, effectively isolating safety-critical components.

Table 2
Summary of the main taxonomy of the works.

Main category	Subcategory	Works
Linux as a GPOS	Hypervisors	[22–39]
	Multiple-kernels	[40–46]
	Heterogeneous hardware	[47–54]
Linux as a SCOS	Certifiability	[6,21,55–66]
	Real-time and deterministic timing behavior	[67–92]
	Fault tolerance and recovery	[93–111]
	Linux-based safety-critical system design	[112–121]

Table 3
Domain of the works.

Domain	Linux as a GPOS	Linux as a SCOS
Automotive	[22,23,28,30,32–34,36,37,39,43, 44,47,49,50,52,53]	[6,71,72,75,79,82,93,113–116, 118]
Avionics	[24–26,31]	[56,61,62,68,70,73,85,90,100, 104]
Medical	–	[96]
Nuclear	–	[48,57]
Robotics	–	[78,92,103]
Space	–	[61,62,112]
Unspecified	[27,29,35,40–42,45,46,51]	[21,55,58,59,63–65,67,69,74,76, 77,80,86–89,91,94,95,97–99,101, 102,105–111,117,119–121]

(2) *Multi-kernel* approaches use multiple kernels running in parallel, with each kernel handling different tasks, thereby isolating the safety-critical functions from the GPOS.
(3) *Hardware-based* approaches utilize dedicated hardware components to enforce separation between the GPOS and the SCOS.

In the second case, Linux as a SCOS, we classify the specific aspects of Linux related to functional safety that are addressed, which are:

(1) *Certifiability*, which refers to the possibility of certifying a system that uses Linux as a SCOS.
(2) *Real-time and deterministic timing behavior*, which covers measuring latencies or ensuring deadlines for task completion, together with scheduling-focused approaches that propose resource allocation and task execution order for multi-core and mixed-criticality systems.
(3) *Fault tolerance and recovery*, which is related to injection, detection, mitigation, and tolerance of faults within the system to ensure that safety-critical functions can continue to operate correctly.
(4) *Linux-based safety-critical system design*, which includes design patterns and architectures proposed for using Linux as an SCOS.

In addition to the general taxonomy, we offer some secondary classifications here: according to the domain of the works (Table 3), in regard to the approach taken, if done so, to make Linux real-time (Table 4), and regarding the Instruction Set Architecture (ISA) of the target platform (Table 5).

Regarding the domain, as seen in Table 3, automotive is the main domain where these works locate themselves, both for using Linux as a GPOS and as a SCOS. Avionics is the domain with the second-most works. These two segments are the ones most interested in the possibility of including the general-purpose capabilities of Linux in safety-critical systems.

Regarding the real-time approach, as presented in Table 4, PREEMPT_RT is the most used approach, especially when Linux is the SCOS. More details about the characteristics of each approach can be found in the survey by Reghenzani et al. [10]. It is noticeable that, except for the PREEMPT_RT patch, the other approaches are unique to each of the roles of Linux. This separation is expected because the approaches used when Linux is the GPOS rely on adding a second kernel that takes care of the real-time tasks, so it makes sense to make that part of the system take care of the safety-critical functionality and use Linux as a secondary GPOS. In the case of the approaches used when Linux is the SCOS, they modify the Linux kernel to make it real-time so it can be in charge of the safety-critical functionality.

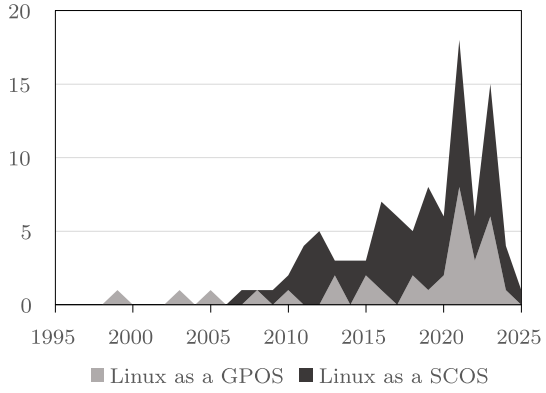

Fig. 1. Stacked area chart of the publications per year, divided according to the use of Linux, either as a SCOS or as a GPOS.

In regards to the target platforms, and specifically the ISA, which can be seen in Table 5, we can see that the 64-bit ARMv8-A and x86-64 are the most targeted architectures, both when Linux is the GPOS and the SCOS. This answers to the high availability of these kinds of platforms for academic work.

Finally, we visualize the number of works per year in Fig. 1, where we can see that there has been increasing interest in using Linux in safety-critical systems, especially in the last five years. Although we find an increase in works that use it as a GPOS in the last five years, some works were already doing so even before the year 2000. The works that use Linux as a SCOS began around the year 2007 and have gradually increased since, with a large spike in the last five years.

5. Linux as a GPOS in a mixed-criticality system

Due to its versatility and broad platform support, Linux has been used as a GPOS for many mixed-criticality systems, alongside a separate safety-critical component (e.g., SCOS). The primary challenge is ensuring that Linux does not interfere with safety-critical functions.

Table 4
Real-time Linux approach (if specified). Note that some works use multiple approaches.

Real-time approach	Linux as a GPOS	Linux as a SCOS
FIN.X-RTOS	–	[104]
Hierarchical scheduling patch	–	[76]
Linux/RK	–	[90]
LITMUS^{RT}	–	[83]
PREEMPT_RT	[24,26,28,31]	[69,71,73,78,79,81,87,89,94,96,105,115,121]
RTAI	[43,45,46,103]	-
RTLinux	[40,42]	-
SParK	[42]	-
Xenomai	[41,44,103]	

-

Table 5
Target Instruction Set Architecture (if specified). Note that some works target multiple platforms and ISAs.

Target ISA	Bits	Linux as a GPOS	Linux as a SCOS
ARMv7-A	32	[22,31,47,48,52,54]	[69,73,97,106,107,110,113]
ARMv8-A	64	[23,29,30,36,50,51,53]	[6,63–65,67–72,75,84,85,91,92,107–109,113,115,118]
PowerPC	32	[42,45]	[120]
RISC-V	64	[29,30,35]	[119]
SPARC (V7, V8)	32	–	[112]
x86	32	[25,27,40,42,43]	[74,82,93,94,116]
x86–64	64	[24,26,28,34,37,41,44,49]	[66,76,78–80,83,86–90,94,95,98,99,103,111]

The isolated safety-critical components can, for instance, include the timing- and safety-critical flight control tasks of a multicopter, while vision-based navigation tasks run in Linux [24]. Another example of an isolated safety-critical component can include the safety-critical ECU functions of a vehicle, such as the critical parts of the Advanced Driver-Assistance Systems (ADAS), In-Vehicle Infotainment (IVI), and Instrument Cluster (IC), which run with safety and real-time guarantees, while the non-critical parts run in Linux [34].

To address the isolation challenge, three categories of solutions have emerged in the scope of this survey:

(1) *Hypervisors*: Hypervisors provide a virtualization layer that allows multiple OS or bare-metal programs to run concurrently on the same hardware, effectively isolating safety-critical components.
(2) *Multiple-kernels*: Multiple-kernel approaches have been employed to guarantee deterministic scheduling of safety-critical tasks, isolating safety-critical and Linux general-purpose functions in the temporal domain.
(3) *Heterogeneous hardware*: Specific heterogeneous hardware architectures and components (e.g. Linux + FPGA, Linux + Safety CPU) have been proposed to enforce separation between the GPOS and the SCOS.

Each of these solutions has its own trade-offs. Hypervisors provide spatial and temporal separation by partitioning or virtualizing the system resources. This allows running Linux alongside the safety-critical functionality of the system. However, they introduce additional complexity, certification effort, and potential performance overhead due to virtualization. Multiple-kernel approaches integrate a real-time kernel with Linux, enabling low-latency response for the applications that require it and interaction between both OSs for applications that require running over Linux, but at the cost of weaker fault isolation, since Linux and the real-time kernel often share memory, drivers, and interrupt controllers. For that reason, a bug or misbehavior in Linux can affect the whole system. Finally, heterogeneous hardware designs depend on specific hardware and technology, such as ARM TrustZone, to partition system resources. They offer strong isolation and determinism with minimal interference, but they demand more complex system integration and specialized toolchains, and can limit flexibility compared to the other solutions.

Since certification is obtained for the system as a whole, in these types of solutions, the hypervisor, second kernel, and hardware also need to be certified. In that regard, several solutions have been developed with certification in mind. For example, PikeOS for the ARMv8 architecture has been certified for SIL4 of EN 50128 and EN 50657 (railway), and SIL3 of IEC 61508 (industrial). Another example is the seL4 microkernel, which claims that "The strength of seL4's formal proofs is considerably higher than what ISO26262's highest level, ASIL-D, requires" [122]. More details about the uses of hypervisors, heterogeneous hardware, etc., can be found in the surveys presented in Section 3.

A summary of the works categorized in this section is provided in Table 6.

5.1. Hypervisors

A hypervisor is a software layer that provides an abstraction layer of physical devices (e.g., CPU, memory) along with a set of functionalities, including resource scheduling and allocation to enable the concurrent execution of multiple OS or bare-metal applications on the same hardware while offering isolation and controlled resource sharing between them. This way, the system can integrate safety-critical and non-safety-critical functionalities in the same platform. There are two main kinds of hypervisors: Type-I or bare-metal hypervisors, which run directly on physical hardware, and Type-II or hosted hypervisors, which run on top of an OS and rely on virtualization. Contrary to hypervisors used in other scopes, such as servers, the main goal of hypervisors proposed for safety-critical systems is generally the partitioning of the system, not virtualization, and the isolation between guests. For that reason, Type-I hypervisors are the most common in this context, although some hybrids also exist.

In this regard, hypervisors of this kind have been developed and presented by some authors, with a focus on certifiability by minimizing source code size. Certain hypervisors avoid reliance on hardware extensions for virtualization by using static partitioning instead [35]. Static partitioning refers to assigning system resources to guests ahead of time and not changing them during runtime. Another two examples of static-partitioning hypervisors are *Jailhouse* [31] and *Bao* [30]. The former utilizes Linux for booting the system, while the latter does it itself.

Other hypervisors have been constructed over existing tools, such as *KHV* [29], which is built on the KVM ecosystem. Moreover, sector-specific hypervisors have also been designed, such as automotive (*DriveOS* [34], *Automotive Hypervisor* [39]), and avionics (*FlyOS* [24, 26]). A different approach in the avionics domain has been proposed

Table 6
Summary of the surveyed works where Linux is used as a GPOS.

Category	Goal	Technology used	Characteristics	Works
Hypervisors	Architectural design	Automotive Hypervisor	Supports different modes to select where each functionality runs	[39]
		CLARE	Linux communicates with the Deep Learning Processor Unit (DPU) to run AI algorithms	[25]
		Jailhouse and CLARE	Multi-SoC architecture	[23]
		KVM	Migrate the Apollo control component from Linux to Erika	[28]
		Manual HW configuration	Vision-based navigation tasks in Linux, flight control in the Quest RTOS	[24]
			Vision-based navigation tasks in Linux, flight control in the Quest RTOS	[26]
		Proof-of-concept	Proof-of-concept platform that integrates convenience and safety functions in the same platform	[33]
		Xen	Linux runs in the Xen control domain	[22]
		XtratuM	Platform for safety-critical Java applications	[27]
	Communication	VOSYSmonitor	Built in the physical layer of the network channel, to reduce complexity	[36]
		Xen	Built over open-source implementations of specifications, allowing reuse of existing drivers	[38]
	Partitioning hypervisor design	Bao	Standalone	[30]
		DriveOS	Booted by the Quest OS	[34]
		Jailhouse	Booted by Linux	[31]
		KHV	Based on KVM but not depending on Linux after boot	[29]
		Proof-of-concept	Temporal partitioning	[32]
		VOSYSmonitoRV	Does not require hardware virtualization extensions	[35]
	System reboots	Quest-V	Suspension and resumption mechanism	[37]
Multiple-kernels	Co-kernel design	RT-Linux	Run real-time Ada tasks within the Linux kernel address space	[40]
		SParK	Partition the system to minimize Verification & Validation effort	[42]
	Design methodology	RTAI	Design a methodology for distributed safety-critical real-time applications	[46]
	Evaluation of capabilities		Evaluation of partitioning capabilities of RTAI/LXRT with a time-triggered scheduler	[43]
	Extension of capabilities		Support the TMO model on the Time Triggered Architecture	[45]
		Xenomai	Hierarchical group scheduling of real-time containers	[41]
			Monitoring to prevent temporal faults	[44]
Heterogeneous hardware	Architectural design	CPU operating modes	x86 SMM to run safety-critical functionality	[49]
			Built over ARM TrustZone to run safety-critical functionality	[54]
		Heterogeneous SoC	Isolation by configuration options of the HW and both OSs	[47]
			Safety-critical system functionality by the use of the heterogeneous SoC, GPOSs isolated by a hypervisor	[52]
		Programmable logic SoPC	All safety-critical functionality in the FPGA	[48]
			Safety-critical functionality running in a RISC-V implemented in the FPGA	[50]
	Evaluation of latencies	Heterogeneous SoC	Comparisons of different architectures with and without OpenAMP	[51]
	Networking		Network filter for separating critical and non-critical traffic	[53]

by Schneider [32], who uses a partition manager in the form of a mutex that works like a hypervisor and manages access to shared resources.

Other authors have presented design approaches for safety-critical systems that are based on hypervisors such as Xen [22], Jailhouse [23], Xtratum [27], and CLARE [23,25]. These hypervisors help isolate Linux from different SCOSs, including Erika [22,23], FreeRTOS [25], and PaRTiKle [27]. In the automotive sector, specific implementations have been proposed for next-generation Software Defined Vehicles (SDVs), where the Quest-V hypervisor is used to isolate Linux and the Quest RTOS [37], and KVM is used to isolate Linux and the Erika RTOS [28].

These heterogeneous architectures are designed to be transparent to the guest OS, representing a potential evolution of automotive safety-related systems [33].

Approaches to communication between guests have also been proposed. For example, VOSYSVirtualNet [36] is a low-latency virtual network link between the secure and non-secure worlds. It works over the VOSYSmonitor [123] hypervisor and is implemented by one buffer per world and a signal to the hypervisor, which is in charge of orchestrating communication. A different approach with the same goal has been proposed for the Xen hypervisor [38], which consists of a framework for Inter-Domain Communication between unprivileged guests of the Xen hypervisor that is built over virtio, RPMsg, and the Xen bus and can improve latencies up to 3x compared to UDP communication between guests.

5.2. Multiple-kernels

Multiple-kernel approaches have been employed to guarantee deterministic timing behavior while also taking advantage of Linux's capabilities. In this survey, "multiple-kernel" refers to executing the standard Linux kernel (i.e., vanilla) alongside a secondary kernel known as the co-kernel. The standard kernel handles non-deterministic, non-safety-related tasks, while the co-kernel manages time-deterministic, safety-critical functions. Unlike a hypervisor, a co-kernel operates in parallel with the standard Linux kernel and has direct control over system resources (e.g., Central Processing Unit (CPU), memory, hardware interrupts) to ensure deterministic real-time responses.

Generally, co-kernels work as follows: they run at the lowest level of interrupt handling, below Linux. Therefore, all hardware interrupts go through the co-kernel, which decides whether it should handle it directly or forward it to Linux. This is decided depending on whether the interrupt is relevant to a real-time task or not. Therefore, two execution domains are created. The real-time domain, which comprises all critical functionality, is run in the co-kernel, while the general-purpose functionality is run in the Linux domain. Communication between the co-kernel and Linux usually happens via special APIs or shared memory. This, together with the specific capabilities of the co-kernel, is implementation-dependent for each solution.

Finally, while Linux has access to its standard range of schedulers, from the Completely Fair Scheduler (CFS) to the real-time FIFO and Round Robin (RR) schedulers, the co-kernel usually only has a simple, predictable, priority-based scheduler that is able to run hard real-time tasks.

The most common co-kernels for Linux are:

- Xenomai [41], including a specific focus on high-criticality tasks control and monitoring [44].
- RTLinux and its variants, such as bare-metal-based saRTL [42], or a custom variant for executing real-time Ada tasks of the multi-tasking runtime system GNARL [40].
- RTAI-based design methodology and framework implementing a Time-Triggered Architecture (TTA), a Time-Trigger Protocol, and adaptations for reducing temporal and spatial interferences [43,45,46].

5.3. Heterogeneous hardware

Specific hardware architectures can enhance safety assurance by isolating safety-critical functions from the Linux GPOS. Three types of hardware configurations can be found in the surveyed literature: multi-core System-on-a-Chip (SoC), multi-core System-on-Programmable-Chip (SoPC), and dedicated CPU operations. It is important to note that different approaches do exist, such as mixed approaches where a hypervisor works on top of a heterogeneous architecture. However, those approaches have not been identified in the surveyed works, so they are not included in this survey.

Commercial Off-The-Shelf (COTS) heterogeneous multi-core SoC platforms can provide dedicated CPUs for this isolation. In these setups, the safety-critical functions run on an independent set of cores, referred to as the Real-time Processing Unit (RPU) or Micro-Controller Unit (MCU), while the Linux OS runs on the general-purpose cores, known as the Application Processor Unit (APU) or Micro-Processor Unit (MPU). RPUs offer several advantages for the deployment of safety-critical functions, such as lock-step operation or freedom from interference with APUs [51,52]. Examples of RPUs include Cortex-R and Cortex-M based cores, which can run safety functions either in a bare-metal environment or on top of an RTOS (FreeRTOS [51], TI-RTOS [52], μC/OS-II [53]). There are also "manual" approaches where the resources of a regular COTS platform are distributed among guest OSs by configuring the hardware, the bootloader, and the OSs themselves [47]. The main disadvantage of these platforms is that the RPUs usually have limited capabilities and resources when compared to the APUs.

COTS SoPC devices offer a Programmable Logic (PL) area in addition to a SoC device, similar to an FPGA, where custom Intellectual Property (IP) blocks can be deployed. Safety-grade devices have their PL isolated from Linux APUs, ensuring freedom from interference. In this type of device, custom IP blocks can implement fault-tolerant components such as monitors [48] or dedicated soft-core processors that run an RTOS (e.g., AUTOSAR [50]) to execute safety-critical functions. While the programmable logic allows for greater flexibility, the complexity of the development rises, since it requires programming in Hardware Description Languages (HDLs), and designs must be thoroughly verified.

The last category of dedicated hardware utilizes specific CPU operating modes, instead of different CPU cores, to isolate safety-critical functions from the Linux GPOS [49,54], by running safety-critical functionality in the same core but in "protected" mode. These modes, such as x86 System Management Mode (SMM) or ARM TrustZone, can effectively host safety functions by enforcing hardware resource isolation and ensuring independence from the normal execution environment. These systems do not require extra cores, which can reduce development costs. However, ensuring proper isolation is not as simple as in the other approaches.

6. Linux as a SCOS

Because of the increasing complexity of safety-critical systems and applications (e.g., autonomous driving), there have been efforts to use Linux as a SCOS and put it in charge of the safety-critical functionality. This raises several challenges that the works included in this survey intend to address. These challenges are:

(1) *Certifiability*: Due to Linux's open-source and general-purpose nature, its development process and characteristics are very particular. For this reason, it does not align well with many requirements and guidelines of safety-critical standards. Still, authors have studied the possibility of certification and argued for some alternative routes to compliance.
(2) *Real-time and deterministic timing behavior*: As a GPOS, Linux does not guarantee real-time behavior. However, the PREEMPT_RT patch and other approaches can do so, and they can be used as enablers for running safety-critical functionality on Linux. Furthermore, ways to detect and avoid timing delays due to resource contention (CPU, memory...) are also necessary, together with proper schedulers that can ensure deterministic timing behavior of the safety-critical task.
(3) *Fault tolerance and recovery*: Random faults happen in every system, but eliminating their impact is especially crucial in safety-critical systems. This is also true for Linux-based safety-critical systems.

(4) *Linux-based safety-critical system design*: A Linux-based safety-critical system includes more components than just Linux, such as the hardware, possibly a hypervisor, middleware, libraries, etc. Safety-critical certification comes for the system as a whole, and not for its individual components, so the system definition and design stages are crucial.

A summary of the works categorized in this section is provided in Tables 7 and 8.

6.1. Certifiability

COTS open-source software, and particularly Linux, has been argued for use in safety-related systems according to the generic IEC 61508, first through the "proven-in-use" route [58] and then through Route 3_S of the standard, also known as "compliant non-compliant development" [59]. Through this route, the idea is arguing that the development of Linux has not followed guidelines from standards, but nevertheless satisfies their requirements.

Focusing on the same standard, and particularly on the 100% test coverage required by it, an argument has been made that traditional static analysis of the code is unfeasible in Linux, and that statistical methods can be used instead [6,63,64]. On the same line, there have been studies on the effect that different filesystems and stress have on the collected data required for the statistical methods [65], as well as the impact of a hypervisor (ACRN) on it [66].

There have also been other proposals regarding the certifiability of Linux, such as an automata-based formal verification approach [55], the implementation of a 3D graphics driver compliant with OpenGL SC 2.0.1 (Open Graphics Library – Safety Critical) [60], and a proposal to also take security certification into consideration when building Linux-based safety-critical systems [21].

The standards from specific domains have also been studied regarding the use and certifiability of Linux.

- *Aerospace* [61,62]: Authors argue that Linux must address at least four challenges for broad use in aerospace: "it must be fast, deterministic, embedded, and assured", and then expand on assurance, proposing that the most promising solution for assurance would be reverse engineering [61]. They also present the newly formed Aerospace working group of the Enabling Linux in Safety-Critical Applications (ELISA) project [62].
- *Civil aviation* [56]: Authors review the state of the art regarding RTOSes, programming languages, tools, and processes for certification in civil aviation. They indicate that previous authors consider Linux as a candidate for certification against the CO-178B standard and suggest that the steps would require generating a paper trail: requirement and design documents, including test plans and procedures, and test and verification procedures.
- *Nuclear power plants* [57]: Authors analyze the Linux functionality and ISO/IEC 23360:2006 (Linux Standard Base (LSB)) and compare them against IEC 62138:2004 (Nuclear power plants). They conclude that the justification for using Linux in Nuclear Power Plant Instrumentation and Control (I&C) systems is insufficient without substantial and laborious verification, validation, certification, and technical support.

6.2. Real-time and deterministic timing behavior

The real-time and deterministic timing behavior of the Linux kernel has been thoroughly studied, including analyses of latencies and deadlines in Linux systems, proposals to fix timing-related problems that have been identified, methods to study and/or ensure proper timing behavior, etc.

Comparisons between vanilla Linux and Linux patched with PREEMPT_RT have been made by some works [69,81]. Others have done the same, but with systems that incorporate Docker containers [79,80] or middleware such as ROS2 and OROCOS [78]. Works agree on the positive impact the patch makes on the timing behavior of Linux. They also indicate that Docker does not have much of an impact on the timing behavior when the patch is in use [79,80], and they recommend not using middleware for high-criticality tasks such as safety-critical applications [78], because middleware hides important system properties whose control is fundamental in a real-time system.

The development of measurement methodology for timing aspects of the Linux kernel (by eBPF [71] or by statistical methods [67]) and active monitoring [77] have also been argued necessary for real-time safety-critical systems. The study of such measures has helped some authors find and fix timing delays in the CUDA framewok [82,83] and even find System Management Interrupts (SMIs) of x86 to be the culprit, raising the question of the x86 platform's suitability for safety-critical applications [83]. Shortcomings of the timing aspects of the Linux kernel have also been found by Zuepke et al. [73] in the fast mutex (futex) implementation, and they propose modifications for a futex implementation in PikeOS that is suitable for hard real-time and safety-critical systems but only provides a subset of the functionality of the Linux implementation.

Resource reservation (CPU, memory...) has also been used as a way to ensure the real-time behavior of the critical task in a Linux system. CPU budgeting [74], for example, relies on reserving CPU time for the safety-critical real-time task, thus ensuring its proper behavior. Linux containers and their orchestration are a different way to achieve this, as done in REACT [76], where containers are orchestrated by taking their real-time requirements into consideration and assigning them a proper CPU quota and budget to ensure their requirements are met. Control groups (cgroups), the Linux mechanism that is used to allocate and limit container resources, have also been used to implement memory request throttling and reduce the memory interference latency of critical applications in Linux-based mixed-critical systems [75]. Apart from cgroups, MemGuard, a memory bandwidth throttling mechanism that works at the hardware level, has also been used to eliminate unpredictable variation in execution times of the critical task due to memory contention [84]. PRedictable Execution Model (PREM) [72,124] is a completely different approach to prevent timing delays caused by resource contention. It relies on including resource-usage-related information in executables, according to annotations made by the programmer, thus resolving resource contention at a high level without depending on low-level arbiters. With this approach, the generated executables become highly predictable in their resource access behavior. Finally, widely used techniques, such as redundancy and diversity have also been used in Linux systems [68,70]. The idea is that timing outliers get masked by redundant execution and that the probability of all redundant executions failing at the same time is extremely low.

6.2.1. Scheduling

Scheduling plays a crucial role in the timing guarantees of safety-critical systems. As such, it has been the focus of multiple works.

Linux's default scheduler is the CFS. It is a general-purpose scheduler, where the next scheduled task is the one that has received less CPU time relative to its weight. Therefore, its goal is to fairly distribute CPU time, not to guarantee deterministic guarantees for deadlines or latencies. For that reason, several real-time scheduling policies are also included in the kernel. SCHED_FIFO and SCHED_RR implement fixed-priority scheduling, where the highest priority task is always executed. The difference between them is that the former executes the task until it blocks or yields, while the latter uses CPU time slices for tasks with the same priority. SCHED_DEADLINE is another real-time scheduling policy that implements Earliest Deadline First (EDF) scheduling. This policy, however, does not use priorities. Instead, three parameters are used: the deadline, the period, and the runtime or the amount of CPU time that the task needs to produce the output. This allows ensuring that the task will be correctly scheduled without needing to know what

Table 7
Summary of the surveyed works where Linux is used as a SCOS (part 1).

Category	Goal	Characteristics	Works
Certifiability	Certification strategy	"Proven in use" route of the IEC 61508 for Linux	[58]
		Paper trail strategy to certify Linux for civil avionics	[56]
		Linux Standard Base is insufficient for nuclear power plants without substantial work	[57]
		"Assessment of non-compliant development" route of the IEC 61508 for Linux	[59]
		Safety and security should be managed jointly	[21]
		Reverse-engineering as the most viable method for Linux certification	[61]
	Extension of capabilities	Safety compliant graphics device driver implementation	[60]
	Formal verification	Automata-model based formal verification approach for the Linux kernel	[55]
	Initiatives	Overview of the ELISA project	[62]
	Test coverage	Non-parametric estimation of the maximum number of execution paths of Linux system calls	[6]
		Parametric estimation of the maximum number of execution paths of Linux system calls	[64]
		Probability and risk estimation of finding untested execution paths of Linux system calls	[63]
		Effect of stress and filesystems in Linux system call execution paths	[65]
		Effect of the ACRN hypervisor in Linux system call execution paths	[66]
Real-time and deterministic timing behavior	Comparison	Comparison of vanilla and real-time Linux in two different boards	[69]
		Comparison of vanilla and real-time Linux, with and without Docker containers	[79]
		Comparison of OROCOS and ROS2 over vanilla and real-time Linux	[78]
		Comparison of vanilla and real-time Linux in different boards and under different load	[81]
	Scheduling	Scheduling metric that balances safety and performance	[92]
		Real-time one-gang-at-a-time scheduling	[91]
		Thermal aware scheduling	[85]
		Scheduling under overload conditions to meet timing requirements of critical tasks	[90]
		Mode- and criticality- aware scheduling	[86]
		Mixed-criticality federated scheduling algorithm for parallel real-time tasks	[89]
		Fault-tolerant scheduling of real-time tasks	[88]
		Time- and event- triggered scheduling	[87]
	Timing assurance	Apply the Predictable Execution Model to ARM-based heterogeneous platforms	[72]
		Predict memory contention and throttle non-critical tasks	[75]
		Timing Diversity, dual modular redundancy to mask timing outliers	[70]
		Fast recovery model for Timing Diversity	[68]
		pWCET estimation of Linux system call execution paths	[67]
		Resource reservation by CPU budgeting	[74]
		Real-time container orchestrator	[76]
		Active monitoring to predict incoming timing violations and steer the system away from them	[77]
		Find and mitigate timing delay causes in CUDA, pthreads, Linux, and the POSIX interface of QNX	[82]
		Memory bandwidth regulation system	[84]
	Timing measurements	New futex implementation based on the Linux implementation	[73]
		Study the possibility of using bpftrace (eBPF) to measure latencies	[71]
		Evaluation of networking latencies under real-time Linux and Docker	[80]
		Search for the cause of CUDA timing glitches and find that system management interrupts are the cause	[83]

Table 8
Summary of the surveyed works where Linux is used as a SCOS (part 2).

Category	Goal	Characteristics	Works
Fault tolerance and recovery	Comparison	Comparison of effects of bit-flips in CPU registers in Linux and bare-metal	[97]
		Comparison of fault tolerance between multi-core Linux with and without parallelization	[110]
		Comparison of fault tolerance between single-core baremetal and multi-core Linux with and without parallelization	[106]
	Fault detection	Fault detection a diagnosis scheme for robotic systems	[103]
		Detect application failures by monitoring multiple OS level resources and detecting anomalies	[100]
	Fault injection	Inject faults to CPU registers and study the effect at application level	[93]
		Software fault injection with minimal disturbance to the application	[94]
		Inject faults in the error reporting registers of the architecture	[98]
		Perform bit-flips in any memory address belonging to the kernel	[99]
		Inject faults in the error reporting registers of the architecture, at hypervisor and OS leve	[95]
		Include a model of the filesystem to be able to reach corner cases with fault injection	[104]
		SPTA analysis to determine hazards and potential causes, and software fault injection framework to simulate hazard scenarios	[96]
		Model kernel paths and insert timing delays in each state of the model, to find the most critical state and the maximum acceptable timing delay	[105]
	Fault mitigation	Force the critical application to use a specific pool of general-purpose CPU registers	[108]
		Force the critical application to use a specific pool of general-purpose CPU registers, in two different architectures	[107]
		Framework that includes fault injection, machine learning to correlate soft error results and system architecture parameters, and mitigation techniques	[109]
		Library for redundant execution in two cores, with staggering for diversity	[111]
		Triple redundancy of the mutex's count variable, to prevent errors caused by bit-flips	[102]
		Modify the Linux scheduler's struct to mitigate the effect of bit-flips	[101]
Linux-based safety-critical system design	Demonstrator	Architecture based on the Operator-Controller-Module concept	[113]
		AUTOSAR Adaptive implemented over Linux	[118]
		Fully virtualize hardware to allow reuse of certified software	[112]
	Design approach	Distributed, object-oriented and component-based approach for platooning	[115]
		Method to dynamically update components of cyclic control systems	[120]
		Guidelines based on SIL2LinuxMP to build safety-critical systems in RISC-V boards with Jailhouse, Linux and AI frameworks	[119]
		KVM as host, AUTOSAR for critical tasks and Linux/Android for general purpose tasks	[116]
		Guidelines to implement safety-critical functionality using Android devices	[121]
		Logical isolation, which means that a function is not led to unexpected behavior by any action of another function	[117]
		Design patterns in recent car developments for self-driving cars	[114]

else is running in the system, provided that the scheduler allows for the addition of the task, because it knows how much CPU time each task needs and can thus deny the addition of more tasks.

Despite the fact that these schedulers bring Linux from a GPOS toward an RTOS, it is important to note that the kernel's non-preemptible sections and the interference introduced by system services and interrupts prevent Linux from being an RTOS even when those schedulers are used. Those are precisely the aspects that the PREEMPT_RT patch modifies to make Linux an RTOS.

In this context, there are works that modify or extend Linux's scheduling capabilities toward its use in safety-critical systems. Some authors propose scheduling methods that rely on reserving CPU cores for the safety-critical task [89,91]. Others argue for the use of certain metrics that the scheduler has to consider, such as a novel safety-performance (SP) metric based on timely computation [92] or the overload-tolerance metric *ductility* [90]. *Ductility* refers to the system's ability to continue executing low-criticality tasks when overloaded by gracefully degrading capabilities instead of outright dropping them.

Finally, there have been proposals to include secondary aspects into consideration when scheduling tasks in safety-critical systems. Among them, we find a thermal-aware scheduling platform that can stop the platform from overheating [85], a scheduler that can consider

both mode and criticality changes for multi-modal mixed-criticality systems [86], a scheduling method to jointly manage synchronization and reliability based on several resource-oriented heuristics [88], and a scheduler that joins time- and event-triggered scheduling [87], which authors argue are often both required in safety-critical systems.

6.3. Fault tolerance and recovery

Safety-critical systems are susceptible to random faults, just as any other system. However, in safety-critical systems, such faults can produce catastrophic consequences for the well-being of humans or the environment, so techniques to mitigate their impact are necessary. Linux-based safety-critical systems are no exception, so many authors have focused their efforts in mitigating the impact of such errors in Linux-based safety-critical systems. We identify four areas of interest in this regard:

- *Fault tolerance*: Studies compare the impact of random errors in Linux systems and in Linux-less systems, with the goal of studying Linux's ability to stop such errors from affecting the system [97,106,110]. They all agree that bare-metal systems are more prone to errors than Linux systems and that the exceptions that Linux generates are helpful to stop errors before they propagate and manifest somewhere else, as happens in bare-metal systems.
- *Fault detection*: Authors have proposed methods to detect faults in Linux systems. These methods consist of the monitoring of OS-level resources in search of anomalies [100], and a diagnosis scheme for component-based robotic systems [103].
- *Fault mitigation*: Techniques for fault mitigation have been proposed by some authors. Among them, some are based on the redundancy of certain information fields in the Linux kernel [101,102], or on redundant and diverse execution of the critical application [111]. Other authors have taken a different approach, by forcing the critical application to use a specific pool of the processor registers. They call this approach Register Allocation Technique (RAT) [107,108]. Finally, SOFIA [109] is a framework that includes fault injection functionality and machine learning to correlate soft error results and system architecture parameters, together with implementations of fault mitigation techniques, namely triple modular redundancy (TMR) and RAT.
- *Fault injection*: Frameworks and techniques for fault injection in Linux systems have been areas of extensive study. Some of the frameworks proposed by authors rely on injecting faults in the underlying hardware [93,95,98], specifically in the error reporting registers of the architecture (MCA). Other frameworks inject software errors and are built as Linux kernel modules [94,99]. Finally, other authors propose model-based fault injection methods. Among these proposals, we can find using Systems Theoretic Process Analysis (STPA) to determine hazards and potential causes of errors and to simulate those hazard scenarios [96], adding a model of the filesystem in the fault injection process to make it possible to reach corner cases [104], and modeling kernel paths and inserting timing delays in each state of the model to find which state is the most critical and what the maximum acceptable timing delay is [105].

6.4. Linux-based safety-critical system designs

Safety-critical certification does not come for individual components but instead for the entire system as a whole. Therefore, the use of Linux in safety-critical systems requires more than just a "safe" version of Linux, such as the hardware, the design of the system, middleware, etc. In that regard, there have been proposals for how Linux-based safety-critical systems could be built.

The SIL2LinuxMP [125] logical isolation architecture has been proposed [117,119] as the base to build such systems. This architecture relies on functionalities of the Linux kernel to create (logically) isolated partitions in Linux, and hence separate safety-critical and general-purpose functionality. Logical isolation means that a function is not led to unexpected (dangerous) behavior by any action of another function, without necessarily implying it in the spatial and temporal dimensions. Combining this logical isolation with hypervisor-level isolation (with Jailhouse) has also been proposed [119]. The prediction by Messnarz et al. [114] based on their study of design patterns found in recent car developments for self-driving cars seems to go in the same direction. They predict that in the future, each component of the car (steering, motor, etc.) will be controlled by an app running on a Linux server that will act as the vehicle's central computer and generate driving inputs.

There have also been Linux-based system design proposals with specific purposes, especially focused on automotive systems. These include guidelines for using Android systems to implement safety-critical functionalities [121], software design for vehicle platooning [115], an architecture for telematics gateways [116], and a method for dynamic updates [120].

Finally, some authors have built demonstrators that display the capabilities of Linux for safety-critical systems.

- An Advanced Driver-Assistance System (ADAS) application in a radio-controlled car [113].
- The implementation of Autosar Adaptive over Linux to execute ADAS and IVI in the same platform [118].
- A Type-II hypervisor that supports full virtualization and is implemented as an extension to the Linux kernel [112]. It is a proof-of-concept for a future standalone hypervisor.

7. Industrial approaches

The safety industry's growing interest in using Linux as a SCOS has led to the emergence of various projects and working groups. This section extends the survey by highlighting initiatives focused on utilizing Linux as a SCOS and examining industrial approaches that currently employ Linux for this purpose. Since this section includes non-academic projects and initiatives, they fall outside of the methodology outlined in Section 4.

One significant initiative is the Linux Foundation's ELISA project [126] (Enable Linux In Safety Applications), which aims to "make it easier for companies to build and certify Linux-based safety-critical applications". This collaborative effort is supported by major organizations, including Boeing in the avionics sector, automotive companies like Bosch, Honda, and Nissan, as well as aerospace agencies such as NASA and EASA (European Union Aviation Safety Agency). The goal is to establish best practices and enhance the robustness of Linux to ensure its suitability for safety-critical applications. Additionally, sector-specific working groups are leading initiatives focused on next-generation aerospace, automotive, and medical devices.

Before this initiative, the SIL2LinuxMP project [125], led by the Open Source Automation Development Lab (OSADL), focused on qualifying the essential minimal components of an embedded GNU/Linux system running on a COTS multi-core platform (bootloader, root filesystem, kernel, C library interface).

The project aimed to achieve compliance of complex safety-related systems with the SIL2 safety standard IEC61508, specifically targeting compliance of pre-existing software according to route 3s (IEC61508-2 Ed2 Section 7.4.2.2). The SIL2LinuxMP project proposes a SIL2 architecture that emphasizes temporal and resource isolation, capitalizing on the diversity of its components to provide assurance through Layers Of Protection Analysis (LOPA).

Both ELISA and SIL2LinuxMP target Linux as SCOS, with their main emphasis on *certifiability*. To date, however, neither initiative has demonstrated a full certification of a Linux-based system. ELISA, in particular, could build on the foundations laid by SIL2LinuxMP and incorporate academic advances such as SPC-based certification

arguments and strategies, which could be further developed within its Special Interest Groups (SIG).

The automotive industry is particularly interested in using Linux as an SCOS, as indicated by the number of works listed in Table 3. The Automotive Grade Linux (AGL) [127] initiative is part of the Linux Foundation and focuses on "bringing together automakers, suppliers, and technology companies to accelerate the development and adoption of a fully open software stack for connected cars". This project has the backing of top automotive manufacturers, including Toyota, Mercedes-Benz, and Volkswagen. AGL focuses on various software applications within vehicles, particularly Advanced Driver Assistance Systems (ADAS), functional safety, and autonomous driving, with Linux serving as its core component.

Tesla has used Linux as the base OS for their autopilot [128]. Currently, the features available are classified as a maximum of SAE level 2 according to the J3016 guidelines [129]. These guidelines categorize levels of driving automation from 0 (assistance, such as AEB) to 5 (where the car drives autonomously in all conditions). Tesla's Autopilot is one of the most visible large-scale deployments of Linux for automotive. While its effectiveness is clear in SAE Level 2 automation, scaling to higher levels will require more rigorous timing guarantees and safety case evidence. Academic proposals on redundancy and scheduling provide potential techniques, but Tesla's proprietary system prevents direct comparison of effectiveness.

Elektrobit's EB corbos Linux for Safety Application [130] is an Ubuntu-based OS designed to achieve ISO 26262 ASIL-B/D certification for automotive High Performance Computing (HPC) systems. This OS is deployed on a safety architecture assessed by the CA TÜV and utilizes the ASIL-B EB corbos hypervisor to ensure deterministic scheduling, hardware virtualization, and memory protection. To maintain system integrity, EB corbos Linux for Safety Applications features a specific user-space initialization, strict application memory protection, and supervision measures that prevent the kernel from modifying user space. Additionally, it supervises the processor context switched to guard against unwanted modification. Even though this approach resembles the use of *Linux as a GPOS* in combination with a hypervisor to enforce isolation, supervision, and monitoring of the safety partition, as surveyed in Section 5, the particularity of EB corbos Linux is that the safety partition itself contains a Linux kernel, which is responsible for safety functions. While the hypervisor strengthens monitoring and fault containment, it also introduces additional complexity, cost, and potential risks to certification. Thus, EB corbos Linux represents a step toward the academic proposals on *Linux-based safety-critical system design*, but still illustrates the gap that remains for adopting Linux directly as the SCOS.

Another certifiable version of Linux is Codethink Trustable Reproducible Linux (CTRL OS) [131], which has obtained baseline safety assessment from the CA Exida: "The assessment of the process framework as applied to CTRL OS has shown that the relevant safety requirements of IEC 61508 at SIL 3 are met and a process compliance argument is complete with this baseline safety case assessment". Although not many details are given, CTRL OS is a Linux-based OS and associated Software Development Kit constructed from free and open source components and tools, including the Linux kernel, systemd, glibc, gcc, etc. It is designed to be integrated into critical and/or mixed-criticality systems based on multicore microprocessors.

CTRL OS represents an effort to position *Linux as a SCOS*, with a focus on process assurance that does not depend on additional components such as virtualization. In this sense, it aligns with the academic research surveyed in Section 6 as it addresses some of the identified challenges. It provides a framework to support certifiability and promotes a design approach that relies directly on Linux itself. Nevertheless, it still lacks a complete end-to-end use case that would validate the full certification process in practice.

The aerospace sector also has its dedicated solutions. Space Grade Linux [132] is part of the ELISA initiative, functioning as a special interest group. Its goal is to design a Linux-based SCOS that includes features such as deterministic real-time processing, low latency, and fault tolerance. The German Space Operations Center (GSOC) utilizes Linux for its spacecraft operations center, and NASA employed it as the GPOS for the Ingenuity Mars helicopter [133]. In this case, Linux is utilized in the navigation computer, which is based on a Qualcomm Snapdragon 801 core within a mixed-criticality architecture. Safety-critical operations are managed by an ARM Cortex-R5 processor that features dual-lockstep functionality [134]. SpaceX has also incorporated Linux-based flight mission-critical computers in its Dragon 9 spacecraft. This system relies on an architecture comprising three dual-core x86 processors, where each core performs identical calculations independently on separate Linux instances. The results of these calculations are then voted on before any actions are taken.

The aviation sector relies on the ARINC 653 standard for space and time partitioning in safety-critical RTOSs. ARINC 653 specifies partitioning mechanisms and API, known as the APEX (APplication EXecutive), to enforce isolation between partitions and to support inter-partition communication. In practice, Linux is often employed as the GPOS in such mixed-criticality environments. For instance, commercial solutions such as Sysgo PikeOS and Wind River VxWorks 653 can host Linux partitions alongside ARINC 653-compliant partitions, with the separation kernel ensuring freedom from interference. Despite the advances, significant challenges remain to use Linux as the SCOS in avionics, most notably guaranteeing temporal and spatial isolation, bounding interference on shared resources, and providing sufficient assurance for certification, which are central issues discussed in Section 6. Academic efforts try to answer these challenges, for example, by proposing Linux-based ARINC 653-aware partitions and schedulers [85,135,136], and hypervisor-based approaches [137].

In the defense sector, Redhawk Linux RTOS [138] is proposed as a commercial open-source SCOS for mission-critical applications (i.e., it is not safety-critical as per a safety standard described in Section 2). Based on Linux, it claims to guarantee determinism and maintain latencies under 5 μs on specifically optimized platforms. It is widely deployed in various defense programs of the U.S. Navy, such as its Aegis Weapon System, which is led by Lockheed Martin. Additionally, Curtiss–Wright has implemented the same RTOS in their Partvus DuraCOR 312 mission computer.

8. Linux in safety challenges

In past sections, we have identified what usage is given to Linux in safety-related systems and what the challenges of using Linux as a SCOS are. This section aims to answer the following Research Question (RQ):

RQ How do existing solutions — academic and industrial — address the key challenges of adopting Linux as a SCOS, and what gaps remain?

In Sections Section 5, 6, and 7 we have identified the current issues of using Linux in safety-critical systems by studying both academic and industrial work. From these issues, we have extracted the following key challenges of using Linux inside a safety-related system.

(1) *Certifiability:* SCOSs usually bring a qualification packet to ensure the system's certification (e.g., Greenhills Integrity RTOS SIL3 pre-certified kernel). Even without a qualification package, approaches to justify the use of Linux as a SCOS can be found in the literature. For example, the SIL2LinuxMP project proposes that Linux could be certified up to SIL 2 level by following Route 3s of the IEC 61508. In addition, the work by Allende et al. [6,63,64,117] and other works that build upon it [65–67] propose statistical or probabilistic methods as the way forward. Finally, it is important to note that two industrial Linux versions claim safety-critical certifiability, EB Corbos Linux and CTRL OS.

(2) *Real-time and temporal determinism:* Ensuring temporal determinism is key in safety-critical systems as explained in Section 2.3. It requires a deterministic scheduling and control over resources to avoid any timing delay. Bringing this requirement to Linux, it has been implemented either by using the Linux full preemptible kernel (PREEMPT_RT), a co-kernel approach (e.g., Xenomai, RTAI), or a tailored Linux kernel (e.g., Redhawk Linux). Although it is strongly platform-dependent, the latest data exhibit a higher jitter by an order of ten for the preemptible kernel against co-kernel approaches.

(3) *Fault-tolerance:* Because random errors can have catastrophic consequences in safety-critical systems, faults have been thoroughly studied in Linux-based safety-critical systems. The literature focuses on four main aspects: (i) fault tolerance, where works have compared the fault tolerance of Linux and Linux-less systems and have found that Linux systems are less error-prone than bare-metal systems. (ii) detection and (iii) mitigation, where authors have proposed modifications for the Linux kernel in order to monitor and stop faults. Some authors also propose modifications to how the critical application interacts with the hardware by forcing it to use a specific pool of processor registers. Finally, (iv) injection, where faults have been injected both at the hardware and software levels, sometimes including formal methods and modeling. In this regard, one main challenge is creating a tool or method that can tackle all four aspects, which has already been tried by some authors [109].

(4) *System partitioning and critical resources access:* In mixed-criticality platforms, the critical part of the system is isolated using mechanisms that ensure access to shared hardware, e.g., hypervisors, dual-kernels, or heterogeneous hardware. Using Linux as a SCOS hence requires using an isolation mechanism besides Linux or inside Linux. These are widely used to run Linux as a GPOS and a different SCOS in the same platform while properly isolating them from each other. However, these approaches, and particularly hypervisors, are also used to isolate a Linux running as a SCOS from another GPOS (or another instance of Linux). Container-based logical isolation has also been proposed as a possible approach to create mixed-criticality systems running over the same Linux kernel. In the automotive domain, EB corbos Linux by Elektrobit is based on a proprietary hypervisor that partitions the system and allows for running two Linux kernels, one for QM and one for the safety-critical tasks.

9. Conclusions

In this literature review, we have presented an overview of the academic works that have focused on using Linux in safety-critical systems. We have categorized the works according to the role of Linux in their systems, separating those that use it as a GPOS isolated from another safety-critical part in the same system and those that use Linux to implement safety-critical functionality. In addition, we have grouped works according to the aspect of Linux they focus on and identified the main challenges for its use in functional safety. When using Linux as a GPOS, isolation of Linux from the SCOS is the main challenge. We have identified that works that do so have used (i) hypervisors, (ii) multiple kernel approaches, and (iii) hardware-based isolation to achieve this. On the other hand, many aspects have been studied when Linux is used as a safety-critical OS, which we classify as (i) certifiability, (ii) real-time and deterministic timing behavior, (iii) fault tolerance and recovery, and (iv) Linux-based safety-critical system design. Moreover, we have briefly presented the main industrial approaches that are trying to use Linux in safety-critical systems. Finally, from the literature and industrial efforts, we have identified the main challenges regarding the use of Linux in safety-critical systems.

CRediT authorship contribution statement

Markel Galarraga: Writing – original draft, Investigation, Writing – review & editing. **Charles-Alexis Lefebvre:** Writing – review & editing, Writing – original draft, Investigation. **Jon Perez-Cerrolaza:** Writing – review & editing. **Jose A. Pascual:** Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work is supported by project MEDUSA (CER-20231011, CERVERA Network of Excellence, financed by the Ministry of Science, Innovation and Universities, Spain through the Centre for Technological Development and Innovation E.P.E. (CDTI), under the European Union's Recovery and Resilience Mechanism (RRM)) and by the Economic Development and Infrastructure Department of the Basque Government (Emaitek program and projectsKK-2023/00012, KK-2023/00090, KK-2024/ 00030, KK-2024/00068 and Consolidated Groups grant IT1504-22). It is also supported by grant PID2023-152390NB-I00 funded by the MICIU/AEI/10.13039/501100011033 and by "FEDER funds".

Data availability

No data was used for the research described in the article.

References

[1] International Electrotechnical Commission, IEC 61508 (1-7): Functional safety of electrical/electronic/programmable electronic safety-related systems (second edition), 2010.

[2] International Organization for Standardization, ISO 26262 (1-10): Road vehicles — Functional safety, 2018.

[3] J. Perez-Cerrolaza, J. Abella, M. Borg, C. Donzella, J. Cerquides, F.J. Cazorla, C. Englund, M. Tauber, G. Nikolakopoulos, J.L. Flores, Artificial intelligence for safety-critical systems in industrial and transportation domains: A survey, ACM Comput. Surv. 56 (7) (2024) http://dx.doi.org/10.1145/3626314.

[4] J. Perez-Cerrolaza, R. Obermaisser, J. Abella, F.J. Cazorla, K. Grüttner, I. Agirre, H. Ahmadian, I. Allende, Multi-core devices for safety-critical systems: A survey, ACM Comput. Surv. 53 (4) (2020) http://dx.doi.org/10.1145/3398665.

[5] J. Perez-Cerrolaza, J. Abella, L. Kosmidis, A.J. Calderon, F. Cazorla, J.L. Flores, GPU devices for safety-critical systems: A survey, ACM Comput. Surv. 55 (7) (2022) http://dx.doi.org/10.1145/3549526.

[6] I. Allende, N.M. Guire, J. Perez-Cerrolaza, L.G. Monsalve, J. Petersohn, R. Obermaisser, Statistical test coverage for Linux-based next-generation autonomous safety-related systems, IEEE Access 9 (2021) 106065–106078, http://dx.doi.org/10.1109/ACCESS.2021.3100125.

[7] The Linux Foundation, 2021 Linux foundation annual report. New Horizons for Open Source, 2021, https://www.linuxfoundation.org/resources/publicationslinux-foundation-annual-report-2021. (Last Accessed on 27 March 2025).

[8] A. Platschek, N. Mc Guire, L. Bulwahn, Certifying Linux: Lessons Learned in Three Years of SIL2LinuxMP, in: Embedded World, 2018.

[9] A. Avizienis, J.-C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Trans. Dependable Secur. Comput. 1 (1) (2004) 11–33, http://dx.doi.org/10.1109/TDSC.2004.2.

[10] F. Reghenzani, G. Massari, W. Fornaciari, The real-time Linux kernel: A survey on PREEMPT_RT, ACM Comput. Surv. 52 (1) (2019) http://dx.doi.org/10.1145/3297714.

[11] V. Struhár, M. Behnam, M. Ashjaei, A.V. Papadopoulos, Real-time containers: A survey, in: Workshop on Fog Computing and the Internet of Things, 2020, URL https://api.semanticscholar.org/CorpusID:216086064.

[12] S. Lozano, T. Lugo, J. Carretero, A comprehensive survey on the use of hypervisors in safety-critical systems, IEEE Access 11 (2023) 36244–36263, http://dx.doi.org/10.1109/ACCESS.2023.3264825, URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85153367542&doi=10.1109%2fACCESS.2023.3264825&partnerID=40&md5=b400a143ab34e61934623d2fe545d8cd.

[13] M. Kassab, Testing practices of software in safety critical systems: Industrial survey, in: ICEIS 2018 - Proceedings of the 20th International Conference on Enterprise Information Systems, vol. 2, 2018, pp. 359–367, http://dx.doi.org/10.5220/0006797003590367, URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85047779106&doi=10.5220%2f0006797003590367&partnerID=40&md5=e27b9f16b26c56c372cea91f8fd5bcd6.

[14] J. Pedersen Notander, M. Höst, P. Runeson, Challenges in flexible safety-critical software development - an industrial qualitative survey, Lect. Notes Comput. Sci. (Including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinform.) 7983 LNCS (2013) 283–297, http://dx.doi.org/10.1007/978-3-642-39259-7_23, URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84884946145&doi=10.1007%2f978-3-642-39259-7_23&partnerID=40&md5=5e8e0bfb802853c416671dc909956bf5.

[15] B. Paden, M. Čáp, S.Z. Yong, D. Yershov, E. Frazzoli, A survey of motion planning and control techniques for self-driving urban vehicles, IEEE Trans. Intell. Veh. 1 (1) (2016) 33–55, http://dx.doi.org/10.1109/TIV.2016.2578706, URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85041966386&doi=10.1109%2fTIV.2016.2578706&partnerID=40&md5=83612f8453c213d8ea4497e362b8df51.

[16] M. Rabe, S. Milz, P. Mader, Development methodologies for safety critical machine learning applications in the automotive domain: A survey, in: IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops, 2021, pp. 129–141, http://dx.doi.org/10.1109/CVPRW53098.2021.00023, URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-85115853735&doi=10.1109%2fCVPRW53098.2021.00023&partnerID=40&md5=c8f2f9f704b204d0365c654152183066.

[17] A. Boglietti, A. Cavagnino, A. Tenconi, S. Vaschetto, The safety critical electric machines and drives in the more electric aircraft: A survey, in: IECON Proceedings (Industrial Electronics Conference), 2009, pp. 2587–2594, http://dx.doi.org/10.1109/IECON.2009.5415238, URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-77951645264&doi=10.1109%2fIECON.2009.5415238&partnerID=40&md5=25ac2eec58231269d9c82c4e936c833b.

[18] S. Kriaa, L. Pietre-Cambacedes, M. Bouissou, Y. Halgand, A survey of approaches combining safety and security for industrial control systems, Reliab. Eng. Syst. Saf. 139 (2015) 156–178, http://dx.doi.org/10.1016/j.ress.2015.02.008, URL https://www.sciencedirect.com/science/article/pii/S0951832015000538.

[19] E. Lisova, I. Šljivo, A. Čaušević, Safety and security co-analyses: A systematic literature review, IEEE Syst. J. 13 (3) (2019) 2189–2200, http://dx.doi.org/10.1109/JSYST.2018.2881017.

[20] G. Kavallieratos, S. Katsikas, V. Gkioulos, Cybersecurity and safety co-engineering of cyberphysical systems—A comprehensive survey, Futur. Internet 12 (4) (2020) http://dx.doi.org/10.3390/fi12040065, URL https://www.mdpi.com/1999-5903/12/4/65.

[21] G. Procopio, Safety and security in GNU/Linux real time operating system domain, in: P. Ciancarini, M. Mazzara, A. Messina, A. Sillitti, G. Succi (Eds.), Proceedings of 6th International Conference in Software Engineering for Defence Applications, Springer International Publishing, Cham, 2020, pp. 245–254.

[22] A. Avanzini, P. Valente, D. Faggioli, P. Gai, Integrating Linux and the real-time ERIKA OS through the Xen hypervisor, in: 10th IEEE International Symposium on Industrial Embedded Systems, SIES, 2015, pp. 1–7, http://dx.doi.org/10.1109/SIES.2015.7185063.

[23] A. Biondi, D. Casini, G. Cicero, N. Borgioli, G. Buttazzo, G. Patti, L. Leonardi, L.L. Bello, M. Solieri, P. Burgio, I.S. Olmedo, A. Ruocco, L. Palazzi, M. Bertogna, A. Cilardo, N. Mazzocca, A. Mazzeo, SPHERE: A multi-SoC architecture for next-generation cyber-physical systems based on heterogeneous platforms, IEEE Access 9 (2021) 75446–75459, http://dx.doi.org/10.1109/ACCESS.2021.3080842.

[24] A. Farrukh, R. West, FLYOS: Integrated modular avionics for autonomous multicopters, in: 2022 IEEE 28th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2022, pp. 68–81, http://dx.doi.org/10.1109/RTAS54340.2022.00014.

[25] E. Cittadini, M. Marinoni, A. Biondi, G. Cicero, G. Buttazzo, Supporting AI-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology, Real-Time Syst. 59 (4) (2023) 609–635, http://dx.doi.org/10.1007/s11241-023-09402-4.

[26] A. Farrukh, R. West, FlyOS: rethinking integrated modular avionics for autonomous multicopters, Real-Time Syst. 59 (2) (2023) 256–301, http://dx.doi.org/10.1007/s11241-023-09399-w.

[27] A. González, W. Mata, A. Crespo, M. Masmano, J.M. Félix, A. Aburto, A hypervisor based platform to support real-time safety critical embedded java applications, Comput. Syst. Sci. Eng. 28 (2013) URL https://api.semanticscholar.org/CorpusID:8668879.

[28] L. Belluardo, A. Stevanato, D. Casini, G. Cicero, A. Biondi, G. Buttazzo, A multi-domain software architecture for safe and secure autonomous driving, in: 2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2021, pp. 73–82, http://dx.doi.org/10.1109/RTCSA52859.2021.00017.

[29] C. Li, R. Guo, X. Tian, H. Wang, KHV: KVM-based heterogeneous virtualization, Electronics 11 (16) (2022) http://dx.doi.org/10.3390/electronics11162631, URL https://www.mdpi.com/2079-9292/11/16/2631.

[30] J. Martins, A. Tavares, M. Solieri, M. Bertogna, S. Pinto, Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems, in: M. Bertogna, F. Terraneo (Eds.), Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020), in: Open Access Series in Informatics (OASIcs), 77, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 3:1–3:14, http://dx.doi.org/10.4230/OASIcs.NG-RES.2020.3, URL https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.NG-RES.2020.3.

[31] R. Ramsauer, J. Kiszka, W. Mauerer, A novel software architecture for mixed criticality systems, in: S. Keil, R. Lasch, F. Lindner, J. Lohmer (Eds.), Digital Transformation in Semiconductor Manufacturing, Springer International Publishing, Cham, 2020, pp. 121–128.

[32] J. Schneider, Overcoming the interoperability barrier in mixed-criticality systems, in: J. Stjepandić, G. Rock, C. Bil (Eds.), Concurrent Engineering Approaches for Sustainable Product Development in a Multi-Disciplinary Environment, Springer London, London, 2013, pp. 1093–1104.

[33] S. Patrick, Advanced in-vehicle systems: A reference design for the future, ISBN: 01487191, 2016, http://dx.doi.org/10.4271/2016-01-0085.

[34] S. Sinha, R. West, Towards an integrated vehicle management system in DriveOS, ACM Trans. Embed. Comput. Syst. 20 (5s) (2021) http://dx.doi.org/10.1145/3477013.

[35] F. Caforio, P. Iannicelli, M. Paolino, D. Raho, VOSySmonitoRV: a mixed-criticality solution on Linux-capable RISC-V platforms, in: 2021 10th Mediterranean Conference on Embedded Computing, MECO, 2021, pp. 1–4, http://dx.doi.org/10.1109/MECO52532.2021.9460246.

[36] J. Vetter, J. Fanguede, K. Chappuis, D. Raho, VOSYSVirtualNet: Low-latency inter-world network channel for mixed-criticality systems, in: 2018 IEEE 13th International Symposium on Industrial Embedded Systems, SIES, 2018, pp. 1–9, http://dx.doi.org/10.1109/SIES.2018.8442097.

[37] A. Farrukh, R. West, JuMP2start: Time-Aware Stop-Start Technology for a Software-Defined Vehicle System, in: R. Pellizzoni (Ed.), 36th Euromicro Conference on Real-Time Systems, ECRTS 2024, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 298, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 1:1–1:27, http://dx.doi.org/10.4230/LIPIcs.ECRTS.2024.1, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2024.1.

[38] F. Lesniak, T. Harbaum, J. Becker, Low-latency inter-domain communication on the Xen hypervisor , in: 2023 IEEE 16th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip (MCSoC), IEEE Computer Society, Los Alamitos, CA, USA, 2023, pp. 340–346, http://dx.doi.org/10.1109/MCSoC60832.2023.00057, URL https://doi.ieeecomputersociety.org/10.1109/MCSoC60832.2023.00057.

[39] K. Kim, Architectural design on hypervisor-based guest OSs and safetyrtos in automotive platform, in: Proceedings of the 2023 International Conference on Research in Adaptive and Convergent Systems, RACS '23, Association for Computing Machinery, New York, NY, USA, 2023, http://dx.doi.org/10.1145/3599957.3606220.

[40] H. Shen, A. Charlet, T.P. Baker, A "bare-machine" implementation of Ada multi-tasking beneath the Linux kernel, in: M. González Harbour, J.A. de la Puente (Eds.), Reliable Software Technologies — Ada-Europe' 99, Springer Berlin Heidelberg, Berlin, Heidelberg, 1999, pp. 287–297.

[41] M. Barletta, M. Cinque, R.D. Corte, Hierarchical scheduling for real-time containers in mixed-criticality systems, in: 2021 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, 2021, pp. 286–287, http://dx.doi.org/10.1109/ISSREW53611.2021.00082.

[42] S. Ghaisas, G. Karmakar, D. Shenai, S. Tirodkar, K. Ramamritham, SParK: Safety partition kernel for integrated real-time systems, in: K. Sachs, I. Petrov, P.E. Guerrero (Eds.), From Active Data Management To Event-Based Systems and more - Papers in Honor of Alejandro Buchmann on the Occasion of His 60th Birthday, in: Lecture Notes in Computer Science, vol. 6462, Springer, 2010, pp. 159–174, http://dx.doi.org/10.1007/978-3-642-17226-7_10.

[43] R. Obermaisser, B. Leiner, Temporal and spatial partitioning of a time-triggered operating system based on real-time Linux, in: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, ISORC, 2008, pp. 429–435, http://dx.doi.org/10.1109/ISORC.2008.10.

[44] D. Loche, A. Génèrès, M. Lauer, J.-C. Fabre, Run-time monitoring and control for temporal fault prevention in mixed-criticality systems, in: 2021 17th European Dependable Computing Conference, EDCC, 2021, pp. 53–60, http://dx.doi.org/10.1109/EDCC53658.2021.00015.

[45] R. Obermaisser, E. Henrich, K. Kim, H. Kopetz, M. Kim, Integration of two complementary time-triggered technologies: TMO and TTP, From Specif. Embed. Syst. Appl. 184 (2005) 211.

[46] R. Obermaisser, P. Peti, A framework for rapid application development of distributed embedded real-time systems, in: The IEEE Region 8 EUROCON 2003. Computer As a Tool., vol. 1, 2003, pp. 80–84 vol.1, http://dx.doi.org/10.1109/EURCON.2003.1247983.

[47] J. Schneider, T. Nett, Safety issues of integrating IVI and ADAS functionality via running Linux and AUTOSAR in parallel on a dual-core-system, in: Automotive - Safety & Security 2014, Gesellschaft für Informatik e.V., Bonn, 2015, pp. 55–68.

[48] C. Toner, H. Boukabache, G. Ducos, M. Pangallo, S. Danzeca, M. Widorski, S. Roesler, D. Perrin, Fault resilient FPGA design for 28 nm ZYNQ system-on-chip based radiation monitoring system at CERN, Microelectron. Reliab. 100–101 (2019) 113492, http://dx.doi.org/10.1016/j.microrel.2019.113492, URL https://www.sciencedirect.com/science/article/pii/S0026271419304822. 30th European Symposium on Reliability of Electron Devices, Failure Physics and Analysis.

[49] N. Mouzakitis, M. Paolino, M.D. Grammatikakis, D. Raho, X86 system management mode (SMM) evaluation for mixed critical systems, in: S. Saponara, A. De Gloria (Eds.), Applications in Electronics Pervading Industry, Environment and Society, Springer International Publishing, Cham, 2021, pp. 164–170.

[50] L. Cuomo, C. Scordino, A. Ottaviano, N. Wistoff, R. Balas, L. Benini, E. Guidieri, I.M. Savino, Towards a RISC-V open platform for next-generation automotive ECUs, in: 2023 12th Mediterranean Conference on Embedded Computing, MECO, 2023, pp. 1–8, http://dx.doi.org/10.1109/MECO58584.2023.10154913.

[51] S. Alonso, J. Lazaro, J. Jimenez, L. Muguira, U. Bidarte, Evaluating the OpenAMP framework in real-time embedded SoC platforms, in: 2021 XXXVI Conference on Design of Circuits and Integrated Systems, DCIS, 2021, pp. 1–6, http://dx.doi.org/10.1109/DCIS53048.2021.9666157.

[52] S. Karthik, K. Ramanan, N. Devshatwar, S. Paul, V. Mahaveer, S. Zhao, M. Vishwanathan, C. Matad, Hypervisor based approach for integrated cockpit solutions, in: 2018 IEEE 8th International Conference on Consumer Electronics - Berlin, (ICCE-Berlin), 2018, pp. 1–6, http://dx.doi.org/10.1109/ICCE-Berlin.2018.8576222.

[53] K.-B. Gemlau, N. Sperling, R. Ernst, Efficient timing isolation for mixed-criticality communication stacks in performance architectures, in: 2022 IEEE 27th International Conference on Emerging Technologies and Factory Automation, ETFA, 2022, pp. 1–8, http://dx.doi.org/10.1109/ETFA52439.2022.9921445.

[54] T. Van Eyck, H. Trimech, S. Michiels, D. Hughes, M. Salehi, H. Janjuaa, T.-L. Ta, Mr-TEE: Practical trusted execution of mixed-criticality code, in: Proceedings of the 24th International Middleware Conference: Industrial Track, Middleware '23, Association for Computing Machinery, New York, NY, USA, 2023, pp. 22–28, http://dx.doi.org/10.1145/3626562.3626831.

[55] D.B. de Oliveira, T. Cucinotta, R.S. de Oliveira, Efficient formal verification for the Linux kernel, in: P.C. Ölveczky, G. Salaün (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2019, pp. 315–332.

[56] A. Kornecki, J. Zalewski, Certification of software for real-time safety-critical systems: state of the art, Innov. Syst. Softw. Eng. 5 (2) (2009) 149–161, http://dx.doi.org/10.1007/s11334-009-0088-1.

[57] A. Andryushin, V. Durnev, A. Chernyaev, Issues of operating systems usage for nuclear power plants, Ann. Nucl. Energy 70 (2014) 87–89, http://dx.doi.org/10.1016/j.anucene.2014.03.009, URL https://www.sciencedirect.com/science/article/pii/S0306454914001212.

[58] N. Mc Guire, Linux for safety critical systems in IEC 61508 context, in: Proceedings of the Ninth Real-Time Linux Workshop in Linz, vol. 5, 2007.

[59] N. Mc Guire, I. Allende, Approaching certification of complex systems, in: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN-W, 2020, pp. 70–71, http://dx.doi.org/10.1109/DSN-W50199.2020.00022.

[60] N. Baek, A graphics device driver implementation compliant with opengl SC 2.0.1 standard specification, Int. J. Appl. Eng. Technology (London) 5 (3) (2023) 143–149, http://dx.doi.org/10.61485/IJAeT/v5-3-2023-18, Publisher Copyright: © 2023, Roman Science Publications and Distributions. All rights reserved..

[61] S.H. VanderLeest, Avionics Linux, in: 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference, DASC, 2023, pp. 1–6, http://dx.doi.org/10.1109/DASC58513.2023.10311247.

[62] S.H. VanderLeest, K. Stewart, Enabling Linux in aerospace applications, in: 2023 IEEE/AIAA 42nd Digital Avionics Systems Conference, DASC, 2023, pp. 1–6, http://dx.doi.org/10.1109/DASC58513.2023.10311338.

[63] I. Allende, N.M. Guire, J. Perez, L.G. Monsalve, J. Fernandez, R. Obermaisser, Estimation of Linux kernel execution path uncertainty for safety software test coverage, in: 2021 Design, Automation & Test in Europe Conference & Exhibition, DATE, 2021, pp. 1446–1451, http://dx.doi.org/10.23919/DATE51398.2021.9473951.

[64] I. Allende, N. Mc Guire, J. Perez, L.G. Monsalve, R. Obermaisser, Towards Linux based safety systems—A statistical approach for software execution path coverage, J. Syst. Archit. 116 (C) (2021) http://dx.doi.org/10.1016/j.sysarc.2021.102047.

[65] Y. Chen, X. Tang, S. Xu, F. Zhu, Q. Zhou, T.-H. Weng, Analyzing execution path non-determinism of the linux kernel in different scenarios, Connect. Sci. 35 (1) (2023) 2192442, http://dx.doi.org/10.1080/09540091.2023.2192442.

[66] R. Shao, Y. Wu, L. Liu, Y. Chen, R. Zhou, Q. Zhou, Dynamic execution path analysis of the Linux kernel on ACRN, in: 2024 China Automation Congress, CAC, 2024, pp. 4433–4438, http://dx.doi.org/10.1109/CAC63892.2024.10865482.

[67] M. Galarraga, C.-A. Lefebvre, J. Perez-Cerrolaza, J.A. Pascual, Toward Linux-based safety-critical systems—Execution time variability analysis of Linux system calls, J. Syst. Archit. 156 (2024) 103266, http://dx.doi.org/10.1016/j.sysarc.2024.103266, URL https://www.sciencedirect.com/science/article/pii/S1383762124002030.

[68] A. Christmann, R. Hapka, R. Ernst, Formal analysis of timing diversity for autonomous systems, in: 2023 Design, Automation & Test in Europe Conference & Exhibition, DATE, 2023, pp. 1–6, http://dx.doi.org/10.23919/DATE56975.2023.10137030.

[69] G.K. Adam, Real-time performance and response latency measurements of linux kernels on single-board computers, Computers 10 (5) (2021) http://dx.doi.org/10.3390/computers10050064, URL https://www.mdpi.com/2073-431X/10/5/64.

[70] R. Hapka, A. Christmann, R. Ernst, Controlling high-performance platform uncertainties with timing diversity, in: 2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2022, pp. 212–219, http://dx.doi.org/10.1109/RTCSA55878.2022.00029.

[71] L. Thomeczek, A. Attenberger, J. Kolb, V. Matousek, J. Mottok, Measuring safety critical latency sources using Linux kernel eBPF tracing, in: ARCS Workshop 2019; 32nd International Conference on Architecture of Computing Systems, 2019, pp. 1–8.

[72] P. Houdek, M. Sojka, Z. Hanzálek, Towards predictable execution model on ARM-based heterogeneous platforms, in: 2017 IEEE 26th International Symposium on Industrial Electronics, ISIE, 2017, pp. 1297–1302, http://dx.doi.org/10.1109/ISIE.2017.8001432.

[73] A. Zuepke, R. Kaiser, Deterministic futexes: Addressing WCET and bounded interference concerns, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2019, pp. 65–76, http://dx.doi.org/10.1109/RTAS.2019.00014.

[74] A. Alonso, E. Salazar, J. López, Resource management for enhancing predictability in systems with limited processing capabilities, in: 2010 IEEE 15th Conference on Emerging Technologies & Factory Automation, ETFA 2010, 2010, pp. 1–7, http://dx.doi.org/10.1109/ETFA.2010.5641339.

[75] J. Kim, P. Shin, S. Noh, D. Ham, S. Hong, Reducing memory interference latency of safety-critical applications via memory request throttling and Linux Cgroup, in: 2018 31st IEEE International System-on-Chip Conference, SOCC, 2018, pp. 215–220, http://dx.doi.org/10.1109/SOCC.2018.8618555.

[76] V. Struhár, S.S. Craciunas, M. Ashjaei, M. Behnam, A.V. Papadopoulos, REACT: Enabling real-time container orchestration, in: 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2021, pp. 1–8, http://dx.doi.org/10.1109/ETFA45728.2021.9613685.

[77] W. Dong, C. Zhao, S. Shu, M. Leucker, Anticipatory active monitoring for safety- and security-critical software, Sci. China Inf. Sci. 55 (12) (2012) 2723–2737, http://dx.doi.org/10.1007/s11432-012-4739-8.

[78] S. Barut, M. Boneberger, P. Mohammadi, J.J. Steil, Benchmarking real-time capabilities of ROS 2 and OROCOS for robotics applications, in: 2021 IEEE International Conference on Robotics and Automation, ICRA, 2021, pp. 708–714, http://dx.doi.org/10.1109/ICRA48506.2021.9561026.

[79] P. Masek, M. Thulin, H. Andrade, C. Berger, O. Benderius, Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle, in: 2016 IEEE 19th International Conference on Intelligent Transportation Systems, ITSC, 2016, pp. 2398–2403, http://dx.doi.org/10.1109/ITSC.2016.7795942.

[80] G. Albanese, R. Birke, G. Giannopoulou, S. Schönborn, T. Sivanthi, Evaluation of networking options for containerized deployment of real-time applications, in: 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2021, pp. 1–8, http://dx.doi.org/10.1109/ETFA45728.2021.9613320.

[81] T.F. De Barrena, A. Garcia, J. Franco, J.L. Ferrando, Impact of real-time Linux for industrial edge AI, in: M. Dassisti, K. Madani, H. Panetto (Eds.), Innovative Intelligent Industrial Production and Logistics, Springer Nature Switzerland, Cham, 2025, pp. 486–504.

[82] S. Liu, R. Wagle, J.H. Anderson, M. Yang, C. Zhang, Y. Li, Autonomy Today: Many Delay-Prone Black Boxes, in: R. Pellizzoni (Ed.), 36th Euromicro Conference on Real-Time Systems (ECRTS 2024), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 298, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2024, pp. 12:1–12:27, http://dx.doi.org/10.4230/LIPIcs.ECRTS.2024.12, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2024.12.

[83] R. Wagle, Z. Tong, R.L. Sites, J.H. Anderson, Want predictable GPU execution? Beware SMIs!, in: 2023 IEEE 29th International Conference on Parallel and Distributed Systems, ICPADS, 2023, pp. 2100–2109, http://dx.doi.org/10.1109/ICPADS60453.2023.00285.

[84] E. Seals, M. Bechtel, H. Yun, BandWatch: A system-wide memory bandwidth regulation system for heterogeneous multicore, in: 2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2023, pp. 38–46, http://dx.doi.org/10.1109/RTCSA58653.2023.00014.

[85] O. Benedikt, M. Sojka, P. Zaykov, D. Hornof, M. Kafka, P. Š° ucha, Z. Hanzálek, Thermal-aware scheduling for MPSoC in the avionics domain: Tooling and initial results, in: 2021 IEEE 27th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, 2021, pp. 159–168, http://dx.doi.org/10.1109/RTCSA52859.2021.00026.

[86] D. de Niz, L.T. Phan, Partitioned scheduling of multi-modal mixed-criticality real-time systems on multiprocessor platforms, in: 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2014, pp. 111–122, http://dx.doi.org/10.1109/RTAS.2014.6925995.

[87] G. Gala, I. Kadusale, G. Fohler, Joint time-and event-triggered scheduling in the Linux kernel, in: The 17th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications, OSPERT 2023, 2023, http://dx.doi.org/10.48550/arXiv.2306.16271.

[88] J.-J. Han, Z. Wang, S. Gong, T. Miao, L.T. Yang, Resource-aware scheduling for dependable multicore real-time systems: Utilization bound and partitioning algorithm, IEEE Trans. Parallel Distrib. Syst. 30 (12) (2019) 2806–2819, http://dx.doi.org/10.1109/TPDS.2019.2926455.

[89] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, C. Lu, Mixed-criticality federated scheduling for parallel real-time tasks, in: 2016 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2016, pp. 1–12, http://dx.doi.org/10.1109/RTAS.2016.7461340.

[90] K. Lakshmanan, D. De Niz, R.R. Rajkumar, G. Moreno, Overload provisioning in mixed-criticality cyber-physical systems, ACM Trans. Embed. Comput. Syst. 11 (4) (2013) http://dx.doi.org/10.1145/2362336.2362350.

[91] W. Ali, H. Yun, RT-Gang: Real-time gang scheduling framework for safety-critical systems, in: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, IEEE Computer Society, Los Alamitos, CA, USA, 2019, pp. 143–155, http://dx.doi.org/10.1109/RTAS.2019.00020, URL https://doi.ieeecomputersociety.org/10.1109/RTAS.2019.00020.

[92] A.H. Sifat, X. Deng, B. Bharmal, S. Wang, S. Huang, J. Huang, C. Jung, H. Zeng, R. Williams, A safety-performance metric enabling computational awareness in autonomous robots, IEEE Robot. Autom. Lett. 8 (9) (2023) 5727–5734, http://dx.doi.org/10.1109/LRA.2023.3300251.

[93] R. Amarnath, S.N. Bhat, P. Munk, E. Thaden, A fault injection approach to evaluate soft-error dependability of system calls, in: 2018 IEEE International Symposium on Software Reliability Engineering Workshops, ISSREW, 2018, pp. 71–76, http://dx.doi.org/10.1109/ISSREW.2018.00-28.

[94] G. Cabodi, M. Murciano, M. Violante, Boosting software fault injection for dependability analysis of real-time embedded applications, ACM Trans. Embed. Comput. Syst. 10 (2) (2011) http://dx.doi.org/10.1145/1880050.1880060.

[95] M. Cinque, A. Pecchia, On the injection of hardware faults in virtualized multicore systems, J. Parallel Distrib. Comput. 106 (2017) 50–61, http://dx.doi.org/10.1016/j.jpdc.2017.03.004, URL https://www.sciencedirect.com/science/article/pii/S0743731517300849.

[96] H. Alemzadeh, D. Chen, A. Lewis, Z. Kalbarczyk, J. Raman, N. Leveson, R. Iyer, Systems-theoretic safety assessment of robotic telesurgical systems, in: F. Koornneef, C. van Gulijk (Eds.), Computer Safety, Reliability, and Security, Springer International Publishing, Cham, 2015, pp. 213–227.

[97] L.G. Casagrande, F.L. Kastensmidt, Soft error analysis in embedded software developed with & without operating system, in: 2016 17th Latin-American Test Symposium, LATS, 2016, pp. 147–152, http://dx.doi.org/10.1109/LATW.2016.7483355.

[98] A. Lanzaro, A. Pecchia, M. Cinque, D. Cotroneo, R. Barbosa, N. Silva, A preliminary fault injection framework for evaluating multicore systems, in: F. Ortmeier, P. Daniel (Eds.), Computer Safety, Reliability, and Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 106–116.

[99] A.D. Velasco, B. Montrucchio, M. Rebaudengo, KITO tool: A fault injection environment in Linux kernel data structures, Microelectron. Reliab. 60 (2016) 153–162, http://dx.doi.org/10.1016/j.microrel.2016.02.011, URL https://www.sciencedirect.com/science/article/pii/S0026271416300300.

[100] A. Bovenzi, S. Russo, F. Brancati, A. Bondavalli, Towards identifying OS-level anomalies to detect application software failures, in: 2011 IEEE International Workshop on Measurements and Networking Proceedings (M&N), 2011, pp. 71–76, http://dx.doi.org/10.1109/IWMN.2011.6088494.

[101] A.D. Velasco, B. Montrucchio, M. Rebaudengo, Hardening approach for the scheduler's kernel data structures, in: ARCS 2017; 30th International Conference on Architecture of Computing Systems, 2017, pp. 1–4.

[102] A.D. Velasco, B. Montrucchio, M. Rebaudengo, TMR technique for mutex kernel data structures, in: 2017 18th IEEE Latin American Test Symposium, LATS, 2017, pp. 1–6, http://dx.doi.org/10.1109/LATW.2017.7906745.

[103] M.Y. Jung, P. Kazanzides, Fault detection and diagnosis for component-based robotic systems, in: 2012 IEEE International Conference on Technologies for Practical Robot Applications, TePRA, 2012, pp. 1–6, http://dx.doi.org/10.1109/TePRA.2012.6269387.

[104] D. Cotroneo, D. Di Leo, R. Natella, R. Pietrantuono, A case study on state-based robustness testing of an operating system for the avionic domain, in: F. Flammini, S. Bologna, V. Vittorini (Eds.), Computer Safety, Reliability, and Security, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 213–227.

[105] R. Shahpasand, Y. Sedaghat, S. Paydar, Improving the stateful robustness testing of embedded real-time operating systems, in: 2016 6th International Conference on Computer and Knowledge Engineering, ICCKE, 2016, pp. 159–164, http://dx.doi.org/10.1109/ICCKE.2016.7802133.

[106] G.S. Rodrigues, F. Rosa, Á.B. de Oliveira, F.L. Kastensmidt, L. Ost, R. Reis, Analyzing the impact of fault-tolerance methods in ARM processors under soft errors running Linux and parallelization APIs, IEEE Trans. Nucl. Sci. 64 (8) (2017) 2196–2203, http://dx.doi.org/10.1109/TNS.2017.2706519.

[107] J. Gava, R. Reis, L. Ost, RAT: A lightweight architecture independent system-level soft error mitigation technique, in: A. Calimera, P.-E. Gaillardon, K. Korgaonkar, S. Kvatinsky, R. Reis (Eds.), VLSI-SoC: Design Trends, Springer International Publishing, Cham, 2021, pp. 235–253.

[108] J. Gava, R. Reis, L. Ost, RAT: A lightweight system-level soft error mitigation technique, in: 2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration, VLSI-SOC, 2020, pp. 165–170, http://dx.doi.org/10.1109/VLSI-SOC46417.2020.9344080.

[109] J. Gava, V. Bandeira, F. Rosa, R. Garibotti, R. Reis, L. Ost, SOFIA: An automated framework for early soft error assessment, identification, and mitigation, J. Syst. Archit. 131 (2022) 102710, http://dx.doi.org/10.1016/j.sysarc.2022.102710, URL https://www.sciencedirect.com/science/article/pii/S1383762122002028.

[110] G.S. Rodrigues, F.L. Kastensmidt, R. Reis, F. Rosa, L. Ost, Analyzing the impact of using pthreads versus openmp under fault injection in ARM cortex-A9 dual-core, in: 2016 16th European Conference on Radiation and Its Effects on Components and Systems, RADECS, 2016, pp. 1–6, http://dx.doi.org/10.1109/RADECS.2016.8093180.

[111] F. Mazzocchetti, S. Alcaide, F. Bas, P. Benedicte, G. Cabo, F. Chang, F. Fuentes, J. Abella, SafeSoftDR: A library to enable software-based diverse redundancy for safety-critical tasks, in: FORECAST Workshop (with HiPEAC Conference), 2022.

[112] H. Joe, H. Jeong, Y. Yoon, H. Kim, S. Han, H.-W. Jin, Full virtualizing micro hypervisor for spacecraft flight computer, in: 2012 IEEE/AIAA 31st Digital Avionics Systems Conference, DASC, 2012, pp. 6C5–1–6C5–9, http://dx.doi.org/10.1109/DASC.2012.6382393.

[113] A. Prakash, L. Krawczyk, C. Wolff, APP4MC RaceCar: A practical ADAS demonstrator for evaluating and verifying timing behavior, in: Proceedings of the 2nd Eclipse Research International Conference on Security, Artificial Intelligence, Architecture and Modelling for Next Generation Mobility, 2021.

[114] R. Messnarz, G. Macher, J. Stolfa, S. Stolfa, Highly autonomous vehicle (system) design patterns – achieving fail operational and high level of safety and security, in: A. Walker, R.V. O'Connor, R. Messnarz (Eds.), Systems, Software and Services Process Improvement, Springer International Publishing, Cham, 2019, pp. 465–477.

[115] S. Mouelhi, D. Cancila, A. Ramdane-Cherif, Distributed object-oriented design of autonomous control systems for connected vehicle platoons, in: 2017 22nd International Conference on Engineering of Complex Computer Systems, ICECCS, 2017, pp. 40–49, http://dx.doi.org/10.1109/ICECCS.2017.32.

[116] Z. Gu, Z. Wang, S. Li, H. Cai, Design and implementation of an automotive telematics gateway based on virtualization, in: 2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, 2012, pp. 53–58, http://dx.doi.org/10.1109/ISORCW.2012.20.

[117] I. Allende, N. Mc Guire, J. Perez, L.G. Monsalve, N. Uriarte, R. Obermaisser, Towards Linux for the development of mixed-criticality embedded systems based on multi-core devices, in: 2019 15th European Dependable Computing Conference, EDCC, 2019, pp. 47–54, http://dx.doi.org/10.1109/EDCC.2019.00020.

[118] M. Kotur, M. Dragojević, G. Velikić, I. Bašičević, Digital cockpit in AUTOSAR adaptive context, in: 2018 IEEE 8th International Conference on Consumer Electronics - Berlin, ICCE-Berlin, 2018, pp. 1–4, http://dx.doi.org/10.1109/ICCE-Berlin.2018.8576209.

[119] C. Hernàndez, J. Flieh, R. Paredes, C.-A. Lefebvre, I. Allende, J. Abella, D. Trilla, M. Matschnig, B. Fischer, K. Schwarz, J. Kiszka, M. Rönnbäck, J. Klockars, N. McGuire, F. Rammerstorfer, C. Schwarzl, F. Wartet, D. Lüdemann, M. Labayen, SELENE: Self-monitored dependable platform for high-performance safety-critical systems, in: 2020 23rd Euromicro Conference on Digital System Design, DSD, 2020, pp. 370–377, http://dx.doi.org/10.1109/DSD51259.2020.00066.

[120] M. Wahler, S. Richter, S. Kumar, M. Oriol, Non-disruptive large-scale component updates for real-time controllers, in: 2011 IEEE 27th International Conference on Data Engineering Workshops, 2011, pp. 174–178, http://dx.doi.org/10.1109/ICDEW.2011.5767631.

[121] A. Armoush, D. Franke, I. Kalkov, S. Kowalewski, An approach for using mobile devices in industrial safety-critical embedded systems, in: G. Memmi, U. Blanke (Eds.), Mobile Computing, Applications, and Services, Springer International Publishing, Cham, 2014, pp. 294–297.

[122] T. seL4 Foundation, seL4 Proofs & Certification, 2025, https://sel4.systems/Verification/certification.html. (Last Accessed on 15 September 2025).

[123] P. Lucas, K. Chappuis, M. Paolino, N. Dagieu, D. Raho, VOSYSmonitor, a Low Latency Monitor Layer for Mixed-Criticality Systems on ARMv8-A, in: M. Bertogna (Ed.), 29th Euromicro Conference on Real-Time Systems (ECRTS 2017), in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 76, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2017, pp. 6:1–6:18, http://dx.doi.org/10.4230/LIPIcs.ECRTS.2017.6, URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECRTS.2017.6.

[124] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, R. Kegley, A predictable execution model for COTS-based embedded systems, in: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium, 2011, pp. 269–279, http://dx.doi.org/10.1109/RTAS.2011.33.

[125] OSADL, SIL2LinuxMP: OSADL - Open Source Automation Development Lab eG, 2014, https://www.osadl.org/SIL2LinuxMP.sil2-linux-project.0.html. (Last Accessed on 30 November 2023).

[126] ELISA, ELISA - Advancing Open Source Safety-Critical Systems, 2019, https://elisa.tech/. (Last Accessed on 27 March 2025).
[127] Automotive Grade Linux, AGL - Automotive Grade Linux, 2012, https://www.automotivelinux.org/. (Last Accessed on 27 March 2025).
[128] S. McElligott, What OS Does Tesla Use?, 2023, https://cars.usnews.com/cars-trucks/features/what-os-does-tesla-use#:~:text=The%20Tesla%2Dspecific%20operating%20systems,based%20projects%20such%20as%20Ubuntu. (Last Accessed on 27 March 2025).
[129] SAE International, Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles, Tech. Rep. J3016_202104, SAE International, 2021.
[130] Elektrobit, EB corbos Linux for Safety Applications, 2024, https://www.elektrobit.com/products/ecu/eb-corbos/linux-for-safety-applications/. (Last Accessed on 27 March 2025).
[131] Codethink, Codethink Trustable Reproducible Linux (CTRL OS), 2025, https://www.codethink.co.uk/ctrl-os.html. (Last Accessed on 21 May 2025).
[132] ELISA, Space Grade Linux SIG. Building a space ready Linux distribution., 2019, https://elisa.tech/space-grade-linux-sig/. (Last Accessed on 17 June 2025).
[133] G. Emad, How Embedded Linux is used in Spacecrafts !, 2024, https://www.embeddedrelated.com/showarticle/1614.php. (Last Accessed on 27 March 2025).
[134] H. avard 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, A. Jain, A.M.S. Martin, G.B. Merewether, Flight control system for nasa's mars helicopter, in: AIAA Scitech 2019 Forum, 2019, p. 1289, http://dx.doi.org/10.2514/6.2019-1289, arXiv:https://arc.aiaa.org/doi/pdf/10.2514/6.2019-1289. URL https://arc.aiaa.org/doi/abs/10.2514/6.2019-1289.
[135] C. Kown, D. Kim, H. Joe, H. Kim, Linux-based memory efficient ARINC 653 partition scheduler, in: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation, ETFA, 2014, pp. 1–5, http://dx.doi.org/10.1109/ETFA.2014.7005306.
[136] S. Han, H.-W. Jin, Full virtualization based ARINC 653 partitioning, in: 2011 IEEE/AIAA 30th Digital Avionics Systems Conference, 2011, pp. 7E1–1–7E1–11, http://dx.doi.org/10.1109/DASC.2011.6096132.
[137] S.H. VanderLeest, ARINC 653 hypervisor, in: 29th Digital Avionics Systems Conference, 2010, pp. 5.E.2–1–5.E.2–20, http://dx.doi.org/10.1109/DASC.2010.5655298.
[138] Concurrent Real-Time, Redhawk Linux RTOS, 2022, https://concurrent-rt.com/products/software/redhawk-linux/. (Last Accessed on 27 March 2025).