

Article

Not peer-reviewed version

Architectural Unsuitability of Linux for Airborne Avionics OS: A System-Level Certification-Oriented Analysis

[Haoran Lu](#) *

Posted Date: 24 March 2026

doi: 10.20944/preprints202603.0354.v4

Keywords: airworthiness; DO-178C; DO-330; determinism; isolation; Linux; operating system certification; partition; trusted computing base; system semantics



Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a [Creative Commons CC BY 4.0 license](#), which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Architectural Unsuitability of Linux for Airborne Avionics OS: A System-Level Certification-Oriented Analysis

Haoran Lu

Independent Researcher, Shanghai, China; 37183985@qq.com

Abstract

This paper presents a certification-oriented, system-level argument that Linux is unsuitable for safety-critical avionics. Because Linux is a feature-rich, high-performance general-purpose OS, it exhibits open and dynamic execution semantics that cannot be finitely bounded or frozen at integration time. Two consequences follow. First, airworthiness infeasibility: an oversized TCB, prohibitive DO-330 toolchain qualification burden, and continuous patch churn that prevents stable, certifiable baselines. Second, semantic complexity: temporal non-isolation and spatial non-isolation, materializing as mutable logical-to-physical mappings, driver-induced contamination of global kernel state, and lack of fault containment. We consolidate these observations into an avionics-oriented OS evaluation framework that makes certification implications explicit—closed-world timing analysis at the partition level, provable spatial and fault isolation, TCB minimization, and lifecycle-stable evidence under DO-178C/DO-330. The framework turns architectural properties into concrete certification risks and provides actionable guidance for OS selection and governance in integrated modular avionics [3,6].

Keywords: airworthiness; DO-178C; DO-330; determinism; isolation; Linux; operating system certification; partition; trusted computing base; system semantics

1. Introduction

Over the past decade, the global aviation industry has entered a phase of steady and diversified expansion, driven simultaneously by the modernization of traditional commercial fleets and the rapid emergence of low-altitude economic activities. New operational domains—such as unmanned aerial systems, urban air mobility, electric vertical-takeoff-and-landing (eVTOL) vehicles, and low-altitude logistics—have significantly broadened the aviation technology landscape. In China in particular, national and regional policies supporting low-altitude airspace reform have accelerated industrial development and attracted substantial private-sector investment. As a result, the ecosystem now consists not only of established aerospace manufacturers but also of a large number of newly formed or cross-industry entrants whose engineering experiences originate from consumer electronics, robotics, and commercial embedded-systems sectors rather than from safety-critical aviation.

This shift in industry demographics creates a structural divergence in system-architecture choices. Well-resourced organizations—especially those with prior experience in safety-critical projects—typically adopt mature, certifiable real-time operating systems (RTOSs) such as VxWorks 653 or PikeOS. These platforms are explicitly designed around partitioning principles and provide the deterministic behavior, analyzable execution semantics, and long-term configuration stability required under DO-178C. Their high licensing and integration costs are acceptable for organizations whose safety processes, program governance, and airworthiness culture already align with traditional avionics expectations. [3,21,23]

However, a large and rapidly growing group of resource-constrained new entrants faces a different set of incentives. Lacking deep familiarity with aviation certification and seeking to minimize time-to-market and platform cost, these companies increasingly turn to Linux—an open-source, feature-rich, and widely supported operating system. Linux offers obvious practical advantages: extensive driver availability, mature tooling, broad developer familiarity, and zero licensing fees. For commercial embedded markets, these attributes legitimately accelerate prototyping and reduce development cost. For teams unfamiliar with DO-178C, Linux may appear not only economical but also technologically “good enough”, especially when combined with hardening efforts or real-time extensions such as PREEMPT_RT. [3,30]

Furthermore, some international Linux institutions are studying the potential role of open-source platforms in future avionics architectures. This has contributed to a widespread misconception, particularly in emerging low-altitude aviation sectors.

This paper addresses that misconception directly. By examining Linux through the lens of DO-178C/DO-330 lifecycle assurance requirements, we demonstrate that Linux’s open-world execution semantics, nondeterministic timing behavior, mutable memory mappings, monolithic trusted computing base(TCB), and continuously evolving toolchain fundamentally contradict the architectural foundations of certifiable airborne systems. To clarify these issues, we present a certification-oriented, system-level analysis that links Linux’s architectural properties to concrete airworthiness impacts and synthesizes them into a unified causal framework. This framework provides a structured basis for operating-system selection, technology governance, and lifecycle planning.

2. Related Work

Research on operating system (OS) suitability for safety-critical avionics has long emphasized the need for deterministic execution, strong spatial and temporal isolation, and a verifiable and stable software baseline. The ARINC 653 family of standards formalizes these requirements through a partitioned execution model with fixed time windows, memory protection regions, and a minimal, analyzable separation kernel architecture. Meanwhile, airworthiness guidance such as RTCA DO-178C defines lifecycle-oriented objectives for requirements traceability, verification rigor, configuration control, and software tool qualification, forming the dominant assurance framework for civil airborne software development. These standards collectively assume that the underlying platform exhibits closed-world, finitely analyzable execution semantics—an assumption that general-purpose operating systems typically cannot satisfy. [1–3]

In contrast to separation kernels, monolithic general-purpose kernels such as Linux implement rich functionality and dynamically evolving subsystems whose behavior depends heavily on runtime conditions. Prior work on Linux real-time extensions, particularly PREEMPT_RT, has focused on reducing kernel latency through techniques such as threaded interrupt handlers and priority-inheritance locks, and these developments have recently culminated in upstream integration of PREEMPT_RT into mainline kernels. However, the PREEMPT_RT project itself acknowledges ongoing challenges with non-preemptible code paths, memory-management latency, and concurrent subsystem interactions that can complicate deterministic timing guarantees. [30]

In parallel, the avionics research community has developed partitioning hypervisors and separation kernels as a more certifiable alternative to monolithic kernels. XtratuM, for example, implements strong temporal and spatial partitioning through a minimal hypervisor architecture designed for real-time embedded systems, enabling deterministic scheduling and strict fault containment boundaries consistent with ARINC 653 principles. [1,13]

More recently, industry initiatives such as the Enabling Linux in Safety Applications project (ELISA) have sought to define processes, tools, and artifacts supporting the evaluation of Linux-based systems in safety-related domains. ELISA collaborates with certification bodies and industrial stakeholders and maintains working groups for sectors including aerospace. However, ELISA explicitly states that it cannot produce a certifiable Linux kernel or guarantee compliance with

aviation standards, positioning its outputs as exploratory guidance rather than certification-oriented evidence. Accordingly, while ELISA demonstrates increasing industrial interest in adapting Linux to safety-related use cases, it does not alter the fundamental architectural considerations underlying DO-178C and ARINC 653 compliance. [36]

Collectively, existing work highlights a clear divide between certifiable partitioned architectures and the open-world semantics characteristic of general-purpose kernels like Linux. While real-time patches and safety-focused initiatives improve aspects of the Linux ecosystem, they do not eliminate the architectural mismatches that underpin airworthiness challenges for DAL A/B avionics systems.

3. Methodology

This work adopts a system-level analytical methodology grounded in standards-based evaluation and architectural reasoning rather than empirical measurement or prototype construction. The objective is to assess Linux's suitability for use in DAL A/B airborne systems by analyzing how its kernel-level architectural properties interact with the explicit requirements defined in DO-178C, DO-330. [3,6]

The analysis proceeds in three structured and mutually reinforcing stages:

3.1. Characterization of Linux Architectural Semantics

The first stage systematically characterizes Linux's architectural semantics—its monolithic kernel structure, dynamic memory-management behavior, asynchronous subsystem interactions, and other properties documented in the real-time Linux and kernel-execution literature.

3.2. Identification of Independently Sufficient Certification Blockers

These architectural characteristics are then rigorously mapped against DAL A/B certification requirements. This standards-based mapping reveals eight distinct architectural deficiencies, each of which is independently sufficient to prevent Linux from satisfying DO-178C/DO-330 and ARINC 653 objectives. In other words, any one of the eight deficiencies alone constitutes a complete and self-standing obstacle to DAL A/B certifiability, and the eight together collectively illuminate the breadth of Linux's systemic misalignment with high-integrity airborne assumptions.

3.3. Unified Causal Framework and Refutation of Misconceptions

To avoid treating these eight deficiencies as isolated observations, the analysis consolidates them into a unified causal structure. This structure traces all eight independently sufficient blockers back to a single architectural root cause—Linux's design as an open-world, general-purpose operating system—and two major consequence categories: airworthiness infeasibility and semantic complexity. To ensure the argument is not susceptible to circumvention, the analysis further evaluates common misconceptions (e.g., PREEMPT_RT, containers, static configuration, etc.) and demonstrates how each misconception fails when examined against the causal structure. This step ensures that alternative interpretations or mitigation strategies do not invalidate the core conclusion.

This methodology relies exclusively on openly documented architectural behaviors and publicly available certification requirements. It constructs an evidence-based, standards-aligned, system-level certifiability argument, consistent with established analytical practices in safety-critical systems engineering.

4. Core Limitations of Linux in Safety-Critical Avionics

Linux's incompatibility with safety-critical avionics stems from eight fundamental architectural deficiencies, every of which directly conflicts with avionics certification standards (e.g., DO-178C), especially for the level DAL A/B airborne systems. Each deficiency is analyzed below, with direct comparisons to avionics-grade platform design requirements.

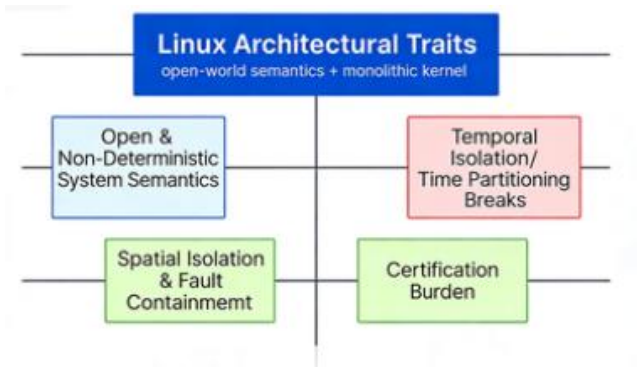


Figure 1. Classification of key Linux architectural traits.

4.1. Open Execution Semantics with Unpredictable Behavior

Linux exhibits inherent open-world system semantics, characterized by a vast, dynamically evolving set of execution paths whose behavior is shaped by runtime operating conditions, workload characteristics, and autonomous interactions among subsystems. These execution paths arise from diverse kernel-internal mechanisms—including background kernel threads, dynamic memory-management activities, interrupt and deferred-work handling, and subsystem-specific state transitions—which evolve independently of application logic and cannot be enumerated or frozen at integration time. The combined effect of these mechanisms results in an unbounded, non-enumerable system state space, driven by hardware events, concurrent execution, dynamic memory pressure, and load-dependent scheduling, which defies closed-form formal analysis and static verification. This core design attribute stands in direct opposition to ARINC653’s architectural philosophy, which mandates closed, finitely analyzable system semantics at integration time: all partition schedules are fixed, memory regions are statically allocated, inter-partition communication is deterministic, and dynamic execution-path creation is prohibited, constraining the separation kernel to a small, finite, and fully analyzable state machine. Linux’s semantic openness is therefore the primary cause of its subsequent temporal and spatial-isolation failures, as it inherently precludes the static, bounded, and fully verifiable behavior required for high-integrity DALA/B avionics systems. [1]

In addition to these dynamically evolving kernel subsystems, Linux further expands its execution state space through dynamic linking mechanisms. Even the linking process contributes to Linux’s open-world execution semantics: because part of the linking and symbol-resolution behavior occurs at runtime, the identity of the executing code is not fully fixed at integration time. Unlike ARINC 653 systems—in which partition images are fully statically linked and all executable content is defined at integration—the Linux execution model allows library composition, symbol resolution, and executable identity to vary with runtime environment and loader behavior. This introduces additional environment-dependent execution paths that cannot be captured within a closed-world model.

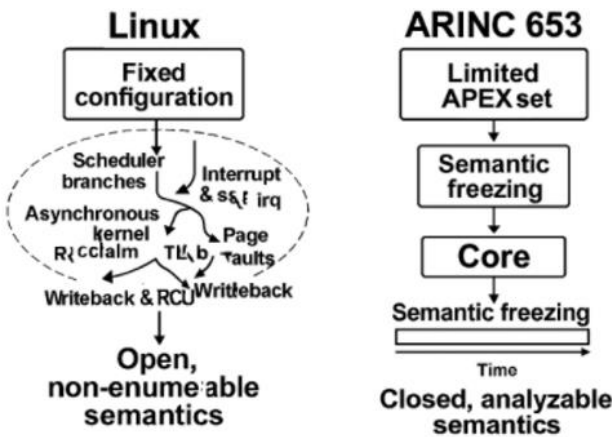


Figure 2. Linux Semantic unfreezing vs Arinc653 Semantic freezing.

This open-world execution model directly violates the deterministic and predictable-behavior requirements for safety-critical aerospace functions stipulated in SAEARP4754A (2010) Clause6.4.3 – the top-level system-engineering guideline underpinning DO-178C software-certification criteria. [3,7]

4.2. Lack of Temporal Determinism

ARINC 653 enforces strict temporal isolation to satisfy DO-178C’s requirements for partitioning and freedom from adverse interactions, which requires the system providing analyzable upper bounds on component release latency, dispatch delay, kernel service time, and interference from other software components. The Linux process–thread execution model cannot supply such guarantees because scheduling decisions, interrupt handling, and kernel activity are driven by dynamically evolving workload, subsystem state, and asynchronous events. As a result, neither processes nor threads can be associated with a statically provable determinism time. [1,3]

In Linux, temporal nondeterminism arises from two architecturally distinct mechanisms whose effects on execution timing must be analyzed separately.

4.2.1. Scheduler-Driven Nondeterminism

Priority-based scheduling in Linux does not guarantee immediate execution for a runnable high-priority task because scheduling behavior depends on the dynamic state of the system, including run-queue composition, wakeup patterns, migration decisions, and internal kernel bookkeeping. These behaviors occur even in the absence of external events.

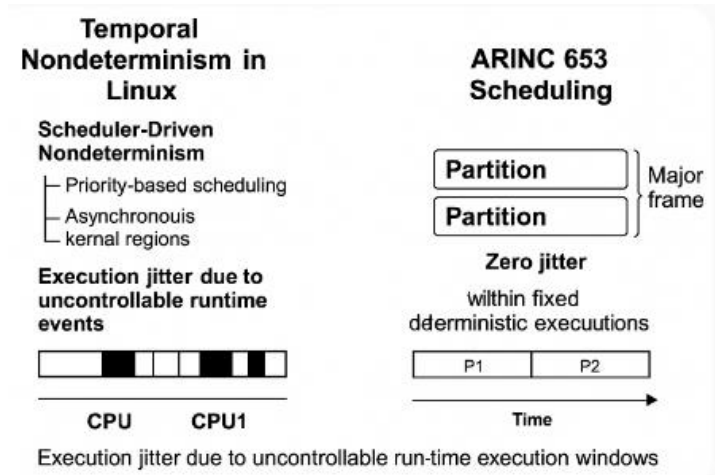


Figure 3. Scheduler-driven execution jitter in Linux vs deterministic ARINC 653 partition.

Furthermore, the Linux kernel contains numerous non-preemptible regions, such as:

- spinlock-protected critical sections,
- per-CPU data updates,
- scheduler state transitions,
- low-level exception-handling paths.

While these sections execute, preemption is explicitly disabled, preventing the scheduler from dispatching a higher-priority process or thread until the critical section completes. The duration of these regions is runtime-dependent and influenced by cache state, lock contention, and microarchitectural factors, making their execution time analytically unbounded.

4.2.2. Event-Driven (Asynchronous) Nondeterminism

Separate from scheduler-driven effects, Linux also contains multiple asynchronous execution sources—including hardware interrupts, software interrupt request(softirq) processing, network-stack activity, timer callbacks, memory-reclaim operations, and background kernel threads. These activities are triggered by external stimuli or internal system conditions and may occur at arbitrary times. As such, they can preempt a running task immediately (e.g., an interrupt) or occupy CPU time through deferred work, introducing additional timing variability that cannot be bounded statically.

Asynchronous nondeterminism therefore reflects the open-world, event-driven nature of the kernel’s interaction with devices, resource pressure, and subsystem events—factors inherently outside the scheduler’s deterministic control.

Conversely, ARINC 653-compliant platforms use a predefined major/minor-frame scheduling model with fixed, deterministic execution windows for all subsystems, eliminating runtime jitter from unconstrained events.

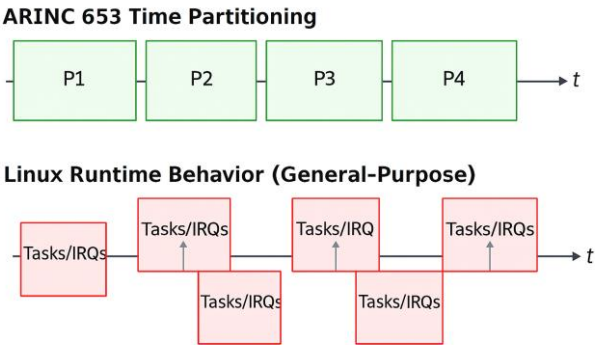


Figure 4. Event-driven execution jitter in Linux versus ARINC 653 fixed time partitions.

ARINC 653 Part 1 explicitly defines fixed time partitioning and temporal isolation to ensure deterministic execution for safety-critical avionics. Without static temporal partitioning, Linux is incompatible with avionics software.

4.3. No Physical Memory Isolation

A certifiable airborne system must ensure strict physical memory isolation so that the memory assigned to a safety-critical partition cannot be altered, accessed, or affected by other software components at runtime. DO-178C requires that the integrity of a partition’s memory region is preserved by static, hardware-enforced address mappings and that these mappings remain stable across the system’s operational life.

However, Linux relies on a dynamic and mutable virtual-memory architecture that violates these requirements. The kernel continuously modifies page-table entries, memory mappings, and physical-page ownership as part of normal operation. Several core mechanisms illustrate this behavior:

- Demand paging and on-demand allocation. Page tables are populated lazily, and physical pages may be allocated, remapped, or reclaimed during runtime based on memory pressure and process behavior.
- Page reclaim and compaction. Under memory pressure, the kernel evicts or relocates physical pages, invoking reclaim, compaction, or write-back paths that modify page-table mappings without application involvement.

- Dynamic page-table updates and Translation Lookaside Buffer (TLB) shutdowns. Linux frequently updates page attributes, permission bits, and mapping structures, triggering cross-CPU TLB invalidations and modifying the effective physical-memory layout during operation. Because Linux performs these modifications autonomously, a process or thread cannot be associated with a fixed, statically provable physical-memory region. No mechanism prevents the kernel from reassigning or altering physical pages that lie within or adjacent to the memory range used by a safety-critical application, nor does Linux provide hardware-enforced barriers preventing other components from accessing or corrupting those regions.
- Thus, Linux’s dynamic page-table management conflicts with the DO-178C requirement that a partition’s physical memory be statically allocated, hardware-isolated, and invariant throughout system operation.

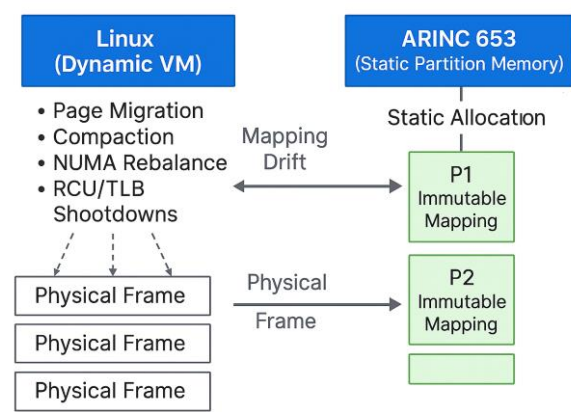


Figure 5. Linux memory mapping drift vs Arinc653 Fixed memory mapping.

4.4. Driver Contamination of Kernel Global State

Linux device drivers execute with full kernel privilege, sharing the monolithic kernel’s global address. As a result, a single defective driver can inadvertently corrupt system-wide shared state including the structures mentioned in above sections.

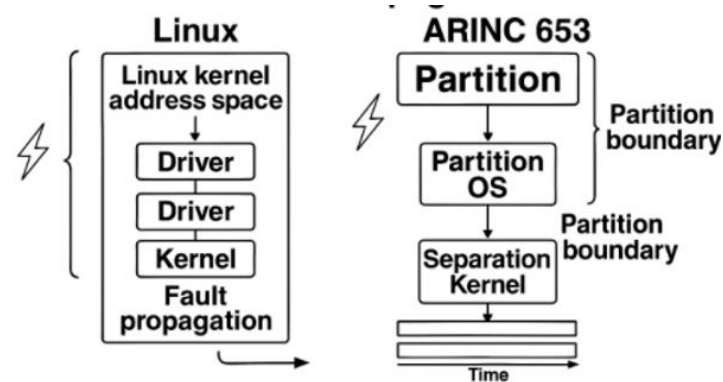


Figure 6. Linux monolithic kernel with drivers vs ARINC 653 separation kernel.

Because these structures are global and not partition-scoped, faults originating in one driver can propagate across unrelated components, undermining the system’s ability to isolate erroneous behavior. This propagation pathway is incompatible with avionics fault-containment principles, where failures within one component must not compromise the integrity or availability of others.

By contrast, the seL4 microkernel provides a minimal trusted kernel and executes device drivers in user-level components with strictly defined authority. Because the kernel exposes no writable

global state to drivers, faults cannot corrupt kernel-internal structures or propagate through shared kernel data.

4.5. Lack of Fault Isolation

Linux does not provide certifiable inter-process fault containment. All user processes ultimately depend on a single, shared, monolithic kernel, and this kernel is responsible for handling every system call. Because all processes share the same privileged kernel address space and kernel-resident global structures, a fault in any process can corrupt kernel state that is globally visible and globally trusted.

In practice, this means that a defect in one component may propagate through:

- shared kernel memory regions used by subsystems such as the scheduler, VFS, networking, and memory management;
- global locks and synchronization primitives that serialize access across unrelated components;
- reference counters and object life-cycle structures (e.g., file descriptors, network buffers, slab objects);
- interrupt-handling and softirq pathways, whose execution contexts are shared across all processes regardless of their criticality.

Fault Propagation in a Shared, Monolithic Kernel



Figure 7. Fault Propagation in Linux.

Since these structures are not partition-scoped—and cannot be restricted or made private to individual processes—Linux cannot prevent a fault originating in one process from affecting the execution correctness, timing, or stability of others. This propagation mechanism is inherent to monolithic-kernel design and directly contradicts the fault isolation defined for DAL A/B airborne software under DO-178C.

Commercial avionics RTOS products implementing ARINC 653 adopt the opposite model. They enforce strict partition boundaries using:

- hardware Memory Management Unit (MMU) isolation with statically defined, non-overlapping physical memory regions;
- a minimal, rigorously verified separation kernel responsible only for scheduling partitions and mediating controlled IPC;
- complete disallowance of shared kernel-writable global state between partitions;
- static temporal partitioning with fixed, predefined time windows to guarantee CPU execution isolation between partitions;
- hierarchical health monitoring (HM) mechanisms supporting fault detection, containment, and recovery at process, partition, and system levels;
- fault-containment boundaries that ensure a failure inside one partition cannot corrupt the separation kernel or any other partition.

As a result, ARINC 653 systems achieve true inter-partition fault isolation, with failures strictly contained to the originating partition. This property is fundamental to satisfying DO-178C fault-propagation analysis for DAL A/B functions.

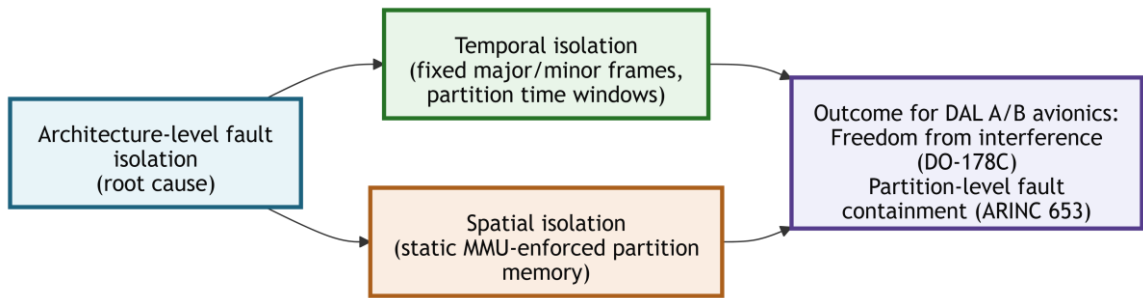


Figure 8. Relationship among architecture-level fault isolation and temporal/spatial isolation for DO-178C DAL A/B.

4.6. *Overly Large Trusted Computing Base (TCB)*

Linux integrates millions of lines of kernel code spanning diverse and continuously evolving subsystems, including device drivers, networking stacks, filesystems, IPC frameworks, tracing and debugging infrastructures, and numerous optional kernel features, all resident components form part of the Trusted Computing Base (TCB). Under avionics standards such as DO-178C, every TCB-resident element contributes directly to certification scope and must undergo full lifecycle assurance activities. For a codebase of this scale and complexity, the resulting verification burden becomes economically prohibitive and technically impractical.

A breakdown of this burden along the five major DO-178C process domains illustrates the mismatch.

4.6.1. Planning Process Impact (PSAC, Standards, Objectives Allocation)

Certification begins with development of the Plan for Software Aspects of Certification (PSAC), where every software component contributing to safety objectives must be identified, characterized, and mapped to DAL-specific objectives.

For Linux, this would require:

- Declaring all kernel subsystems, all bundled drivers, and all architecture-specific code paths as part of the certifiable airborne software item.
- Producing planning artifacts (PSAC, SDP, SVP, SCMP, SQAP) that must describe how each subsystem achieves determinism, verifiability, testability, and configuration stability.
- Defining an assurance strategy for kernel-wide concurrency, memory sharing, interrupt handling, scheduling, dynamic allocation.

Because Linux includes thousands of modules and hundreds of interdependent subsystems, planning artifacts would become unmanageably large, and many required DAL A/B planning commitments (e.g., complete design traceability or determinism justification) simply cannot be demonstrated.

4.6.2. Development Process Impact (Requirements, Design, Code)

DO-178C requires a requirements-driven, top-down software lifecycle with traceable and reviewable transitions from high-level requirements (HLR) to low-level requirements (LLR), and from the software high-level design (HLD) to the software low-level design (LLD), and finally to the source code.

Linux conflicts with this expectation:

- The kernel contains vast quantities of implementation-driven code, developed incrementally without DAL-style requirements decomposition,

- Core subsystems (scheduler, memory management(MM), virtual file system(VFS), network stack, timers, softirq) lack formalized HLR/LLR specifications, making traceability impossible,
- Many kernel paths have implicit behavior (e.g., locking, Read-Copy Update(RCU) semantics, memory reclaim conditions) that cannot be fully captured in DAL-style requirements.

To bring Linux into DAL A/B development conformance would require rewriting vast portions of the kernel under DO-178C processes—defeating the purpose of adopting Linux in the first place.

4.6.3. Verification Process Impact (Reviews, Test, MC/DC, Robustness)

Every TCB line of code must be verified to satisfy DO-178C objectives. For DAL A/B this includes:

- Structural coverage analysis up to MC/DC at the source level,
- Verification of all exception paths, error handlers, corner cases, and architecture-specific branches,
- Robustness testing against abnormal inputs and worst-case conditions,
- Review of all interfaces, data flows, and shared states.
- Requirement that all DAL A/B code be traceable to requirements, fully exercised by verification, leaving no dead code or unintended functionality.

Linux's scale makes these obligations infeasible:

- The kernel's millions of lines of code require astronomical verification effort,
- Dynamic kernel subsystems—such as memory reclaim, workqueues, softirq, and RCU—are driven by system-level conditions including variable load, interrupt patterns, scheduler decisions, and configuration parameters. Because these external stimuli cannot be deterministically bounded, execution timing exhibits non-deterministic variation. Consequently, neither complete coverage closure nor a rigorously bounded WCET can be guaranteed under all operating conditions.
- MC/DC on the kernel would require analyzing tens of thousands of complex decision points, many interacting across subsystems.
- The Linux kernel contains a large volume of configuration dependent, architecture specific, or rarely executed code paths (e.g., unused error paths, debug logic, fallback handlers, and deep #ifdef branches), many of which cannot be deterministically exercised or fully covered, and often have no requirement level justification.

The verification burden alone exceeds the cost and feasibility envelope of civil certification.

As a result, substantial portions of Linux remain unverifiable under DO-178C's code-level objectives, making DAL A/B compliance infeasible.

4.6.4. Configuration Management (SCMP, Baseline Control, Change Records)

DO-178C requires that configuration management ensure consistent control over software artifacts—including requirements, design data, source code, parameter document, and verification evidence—so that traceability and reproducibility are maintained throughout the lifecycle. For a monolithic kernel of Linux's scale, the diversity of configuration options and the volume of generated artifacts complicate basic configuration-control activities; whereas, the deeper issue is the instability of the software baseline caused by Linux's continuous patch stream.

Because baseline stability is a lifecycle-level property rather than an artifact-level one, a detailed examination of this issue is provided in Section 4.7.

4.6.5. Quality Assurance (SQAP, Process Audits, Independence Requirements)

QA must demonstrate that every process objective is followed and that independence is maintained for verification activities for DAL A/B.

Linux violates these expectations because:

- The development is performed by a large, open community, no QA organization can audit or ensure compliance of upstream kernel changes.
- The kernel’s enormous TCB size makes independent reviews impractical, as QA must assess the entire lifecycle—from requirements to design to code—for millions of lines and hundreds of contributors.

ARINC 653-based systems take the opposite approach:

- The separation kernel is purpose-designed to be extremely small, static, and analyzable.
- Device drivers and applications run outside the certified kernel, in isolated partitions.
- The separation kernel’s TCB is on the order of tens of thousands of lines, not millions, making DO-178C planning, verification, configuration control, and QA processes achievable at DAL A.

This minimal-TCB architecture exists specifically to avoid the audit explosion that characterizes monolithic kernels like Linux.

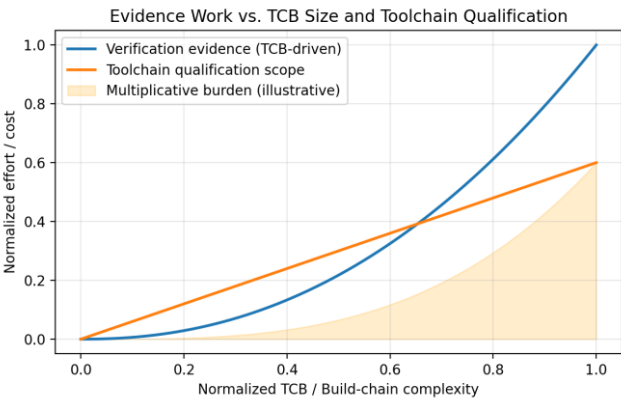


Figure 9. TCB & Toolchain: The Certification Cost Curve.

4.7. Continuous Patch Stream Destabilizes Certified Baselines

Linux evolves at a rapid pace, and maintaining a stable, certifiable baseline is incompatible with this development model. Kernel updates are continuous and unavoidable because Linux must support a vast hardware ecosystem, address frequent security vulnerabilities, and resolve behavioral regressions across numerous subsystems. These patches routinely modify core kernel semantics.

For safety-critical software, DO-178C requires that once a baseline is selected, its behavior must remain frozen, repeatable, and fully traceable throughout certification. Any change to that baseline — no matter how small — invalidates prior verification evidence and triggers the DO-178C change-management process: impact analysis, regression testing, artifact updates, and re-establishment of linkage between requirements, design data, and test results. A kernel that changes frequently cannot satisfy this expectation.

An additional complication is that PREEMPT_RT, the component most relevant for deterministic behavior, is itself still under active refinement. Because its design continues to evolve — in areas such as sleeping spinlocks, threaded interrupts, low-latency code paths, and real-time scheduler behavior — maintainers must regularly modify the underlying kernel infrastructure. As upstream Real Time(RT) support matures, new patches inevitably alter timing behavior, locking rules, and execution semantics. This creates semantic churn directly within the parts of the kernel that would be most critical to a certifiable real time baseline. In practice, this means any attempt to “freeze” a PREEMPT_RT-based Linux kernel will quickly diverge from upstream and will require ongoing, high-cost revalidation.

By contrast, commercial avionics kernels intentionally maintain frozen, long-lived baselines, applying only narrow, tightly controlled corrective patches under strict configuration control. Their architectures and processes are specifically optimized to meet DO-178C requirements for version stability, reproducibility, and long-term maintainability.

4.8. Complex Toolchain Imposes Prohibitive DO-330 Qualification Burden

Linux relies on a broad and deeply layered toolchain ecosystem that includes compilers, linkers, meta-build systems, configuration generators, device-tree compilers, package utilities, and numerous scripting tools. This ecosystem is structurally different from avionics development environments, both in scale and in the number of transformations that occur between source and final binaries.

Even if all tool versions were frozen, DO-330 requires qualification for every tool that can affect airborne software, as well as for each transformation stage that generates derived artifacts. Linux's build process depends on dozens of such tools—GCC/LLVM, binutils, kbuild, Kconfig, autotools, CMake, Yocto, Device Tree(DT) compilers, Python and shell-based code generators, pkg-config, ninja, and many others. Each participates in multiple stages of the build pipeline, producing intermediate files, scripts, headers, configuration databases, or hardware description blobs consumed by the kernel.

Under DO-330, each tool and each interaction between tools must be justified or qualified, and the scope scales combinatorially. This creates an immense qualification envelope that is economically infeasible for DAL A/B programs — even before considering version churn.

Except for the tool-qualification obligations defined in DO-330, DO-178C itself requires verification of source-to-object code consistency and traceability. The standard Linux toolchain provides no mechanism to ensure this. Modern compilers introduce aggressive optimizations, implicit code transformations, architecture-dependent instruction selection, and complex multi-stage build flows, all of which may generate binary artifacts that cannot be deterministically or completely traced back to the originating source code. As a result, the toolchain cannot guarantee strict source-object semantic consistency as required by DO-178C.

Whereas, avionics RTOS environments deliberately employ minimal, deterministic, and long-term-stable toolchains. A single vendor-qualified compiler, along with a simple assembler and linker, constitutes the entire TQL surface. Build orchestration may still use a meta-build system (e.g., CMake)—including in large aerospace programs—but in avionics it is typically frozen, project-scoped, and configuration-controlled; importantly, such tooling must still be assessed under DO-330 based on whether it can introduce or mask errors in airborne software. The critical distinction is the absence of a continuously evolving distribution build stack (e.g., Yocto/Buildroot) and large transitive host-tool dependency chains. As a result, automatic code generation and distribution-level tool dependencies are minimized or eliminated.

5. Unified Causal Structure of Linux's Architectural Incompatibility

This work can be summarized through a unified and internally consistent causal chain that explains why Linux is fundamentally incompatible with safety-critical avionics.

Root Cause: Linux is a function-rich, high-performance general-purpose operating system. This design choice inherently triggers two downstream consequences, forming a coherent causal chain that explains Linux's incompatibility with safety-critical avionics:

5.1. Airworthiness Infeasibility

Consequences Derived from Airworthiness Infeasibility. The first consequence is attributable to two additional certifiability-blocking properties:

5.1.1. An Excessively Large Trusted Computing Base (TCB)

5.1.2. A Highly Complex Toolchain That Imposes Prohibitive DO-330 Qualification Burdens

Jointly Caused Consequence. Finally, both primary branches—airworthiness infeasibility and semantic complexity—jointly produce:

5.1.3. Continuous Patch-Stream Evolution, Which Prevents the Formation of Any Stable, Certifiable Baseline and Invalidates Prior Verification Evidence

5.2. Complex and Open System Semantics

The execution state space becomes non-finite, workload-dependent, and continuously evolving. Of course, the complicated system semantics is also partial of reason for above Continuous patch-stream evolution.

The second primary consequence Derived from two major technical deficiencies:

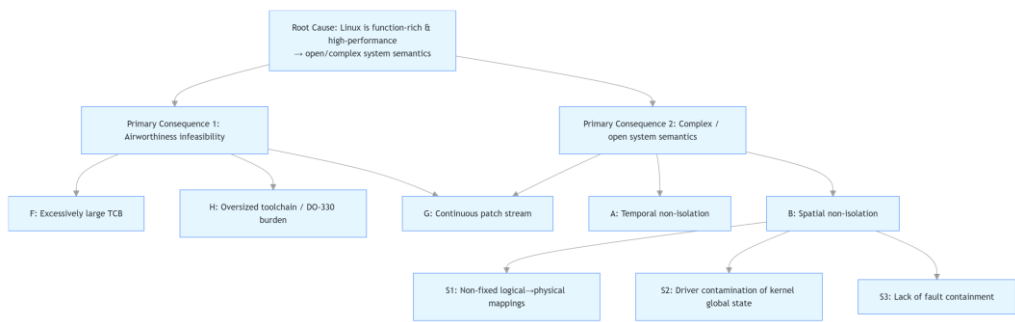


Figure 10. Two primary consequences and eight derived architectural deficiencies forming the Linux certification risk structure.

5.2.1. Temporal Non-Isolation–Arising from Asynchronous Kernel Activity, Dynamic Scheduling Behavior, and Non-Preemptible Regions

5.2.2. Spatial Non-Isolation–Further Decomposing into Three Architectural Mechanisms

- Mutable logical-to-physical memory mappings (dynamic page tables, reclaim, compaction),
 - Driver-induced contamination of globally shared kernel state,
 - Lack of hardware-enforced fault-containment boundaries.
- Consequently, this causal chain provides a complete and coherent explanation for all eight architectural deficiencies discussed in this paper, showing that they are not isolated issues but systemic consequences of Linux’s foundational design philosophy.

6. Common Misunderstandings and Why They Fail

6.1. PREEMPT_RT = Determinism

6.1.1. Misconception

Applying PREEMPT_RT converts Linux into a deterministic, certifiable real-time system.

6.1.2. Reality

PREEMPT_RT improves average-case latency by threading interrupts and shrinking non-preemptible regions, but it does not turn Linux into a temporally deterministic system.

Unbounded interference sources remain:

- IRQ exit softirq cascades;
- memory management events (page faults, reclaim, compaction) that include non-preemptible critical sections and global synchronization;
- cross-CPU TLB shootdowns that can outrank RT tasks;
- driver execution time that depends on firmware/DMA completion/lock contention;
- DVFS and CPU C-state exits that introduce microarchitectural stalls outside scheduler control;

- dynamic scheduler behavior (wakeup rules, load balancing, task migration, housekeeping threads) that produces runtime-dependent jitter.

Because of these, even a high-priority real-time task may fail to preempt and run immediately when it becomes runnable (dispatch/preemption latency), and may be preempted or lose CPU time while executing.

6.1.3. Avionics Impact

DAL A/B avionics needs temporal determinism backed by analyzable upper bounds on interference. ARINC 653 achieves this with fixed, table-driven execution windows (temporal partitioning).Linux (even with PREEMPT_RT) remains an as-soon-as-possible (ASAP) scheduling model, in which asynchronous kernel activities can consume CPU time unpredictably. Therefore, PREEMPT_RT cannot support ARINC 653-style temporal partitioning or provide partition-level schedulability evidence under DO-178C.

6.2. Containers/ Virtual Machines (VMs)/Cgroups = Partitioning

6.2.1. Misconception

Namespaces/Containers/Cgroups/VMs (on a Linux host) provide avionics-grade partitioning equivalent to ARINC 653.

6.2.2. Reality

Containers and Linux hosted VMs ultimately share the same monolithic kernel and its global state. Cgroups offer resource shaping, not hardware enforced temporal or spatial isolation. As long as kernel global data structures, interrupt domains, MM events, and driver paths are shared, faults and timing interference can propagate.

Fundamentally, these mechanisms still amount to application-level constrained or bound processes (or process groups), and therefore retain the composition-level fault-contagion risk described in Sections 4.2, 4.3 and 4.5.

6.2.3. Avionics Impact

ARINC 653 requires strict temporal partitioning (fixed, table driven major/minor frames) and strict spatial isolation (hardware enforced, immutable physical memory regions). Cgroups provide neither:

- Enforcing strict and deterministic execution budgets, the cgroup scheduler provides only proportional CPU time shares. As a result, each group receives an average amount of runtime but with no guarantee of bounded latency or deterministic execution timing.
- Spatial isolation fails because cgroups do not define exclusive physical memory ranges, do not prevent page migration, compaction, NUMA balancing, or TLB shootdowns, and do not protect kernel global state.
- Fault isolation fails because any cgroup can corrupt shared kernel global data structures or destabilize drivers, affecting others. Therefore, cgroups/containers/VMs cannot satisfy DO-178C DAL A/B isolation or ARINC 653 partition requirements.

Table 1. Linux cgroups vs ARINC 653 partitions.

Aspect	Linux cgroups	ARINC 653 Partitions
Isolation model	Resource quotas (CPU/mem/IO); no hard isolation	Hardware enforced spatial isolation (MMU)
Kernel domain	All tasks share one monolithic kernel	Separation kernel enforces strict boundaries
Fault containment	None – faults propagate through shared kernel state	Strong – faults confined to partition
Memory separation	No exclusive physical regions; no immutable mappings	Dedicated address spaces, fixed at integration
Time isolation	No deterministic execution or WCET guarantees	Deterministic major/minor frame scheduling
System semantics	Dynamic, open world, non enumerable	Closed, analyzable, integration time frozen
TCB size	Millions of LOC (grows with every patch)	Small, static, certifiable separation kernel
Certification suitability	Cannot meet DO 178C DAL A/B	Designed specifically for DAL A/B compliance
Purpose	Resource management	Safety critical partitioning architecture
Nature	Policy mechanism inside Linux	Architectural foundation of safety systems

6.3. Using mlock() and Disabling Swap to Fix Partition Memory

6.3.1. Misconception

Locking pages with mlock() or disabling swap provides deterministic memory behavior and partition-like physical isolation.

6.3.2. Reality

Both mechanisms only affect paging, not physical-memory determinism. Even with swap disabled and pages locked, Linux may still reclaim, migrate, compact, or remap physical pages, and continues to update Page Table Entries(PTE) and trigger TLB shootdowns. Memory remains part of a shared, dynamic global pool—not a statically isolated region.

6.3.3. Avionics Impact

ARINC 653 requires immutable, hardware-enforced physical memory regions with no runtime page-table changes. Linux’s dynamic memory subsystem prevents establishing such static boundaries, making spatial isolation infeasible under DO-178C DAL A/B.

6.4. Static Configuration Make Linux Deterministic

6.4.1. Misconception

Fixing the process/thread set, statically allocating stacks and memory regions, constraining communication patterns, masking interrupts, removing signals, or otherwise “freezing” the application-level structure can transform Linux into a closed, deterministic execution environment.

6.4.2. Reality

Static application-level configuration reduces user-space variability, but it does not change Linux’s open-world execution semantics. Even with a fully static thread topology, fixed memory size, and restricted IPC behavior, the kernel continues to generate new execution paths at runtime based on hardware events, memory state, device activity, and subsystem heuristics.

These kernel behaviors are not tied to the static structure of the application, and their activation conditions form a combinatorial space of possible interleavings, depending on instantaneous system conditions. The kernel may interact, so that cannot be enumerated or frozen at integration time.

Thus, static configuration affects what the application does, but not what the kernel may do. Linux remains an open-world semantic system, where the control-flow graph expands dynamically and cannot be reduced to a finite, certifiable model through configuration alone.

More broadly, even if Linux is subjected to minor or moderate kernel modifications (hardening, configuration tightening, or targeted real-time changes), its fundamental architectural blockers for DAL A/B remain. Conversely, the degree of restructuring required to truly satisfy avionics operating-system assumptions (closed-world semantics, analyzable partition-level timing, and provable spatial/fault isolation) would be so extensive that the resulting kernel would no longer be meaningfully “Linux” in the COTS/upstream sense, but rather a different operating system derived from it.

6.4.3. Avionics Impact

DO-178C assumes a closed-world, finitely analyzable execution model via all operational behavior must be fully determined and verifiable at integration time, with no emergent execution paths permitted at runtime. Linux violates this assumption at the architectural level.

Because Linux continues to produce:

- asynchronous kernel execution independent of application structure,
- dynamically generated control-flow paths influenced by instantaneous system state, and
- unbounded permutations of kernel event ordering.

Linux’s open-world semantics inherently allow new execution paths to arise at runtime, producing combinatorial interleavings that cannot be enumerated at integration time, violating DO-178C requirements for deterministic, closed-form timing analysis and make the behavior unpredictable.

6.5. *Abundant Linux Ecosystem*

6.5.1. Misconception

Linux’s extensive ecosystem—its diverse toolchains, build systems, subsystems, and libraries—provides engineering convenience and therefore should make it easier to build safety-critical airborne software.

6.5.2. Reality

The breadth of the Linux ecosystem increases engineering convenience but dramatically amplifies certification complexity. Linux depends on a large and heterogeneous collection of compilers, linkers, meta-build systems, configuration generators, device-tree compilers, packaging tools, scripting environments, and subsystem-specific utilities. Each of these components introduces additional transformations, dependencies, and versioned artifacts that must be assessed under DO-330 when they affect airborne software.

6.5.3. Avionics Impact

For DAL A/B avionics, “more tools and libraries” usually means more certification scope: more tool qualification exposure under DO-330, more configuration items to control, and more change impact to re-verify. As a result, ecosystem richness tends to increase assurance cost and lifecycle risk, even if it reduces prototype effort. Thus, the richness of the Linux ecosystem—an advantage for general engineering—becomes a certification burden under DO-178C/DO-330 and is misaligned with the lifecycle stability expected in airborne systems.

6.6. *Open Source = Reduces Cost*

6.6.1. Misconception

Because Linux is open source and has no licensing fees, adopting it should reduce overall program cost for airborne software.

6.6.2. Reality

Licensing is usually not the dominant cost driver in a DAL A/B DO-178C program. What dominates is the amount of assurance work and evidence that must be planned, traced, verified, qualified, configuration-controlled, and repeatedly re verified over the lifecycle. A general purpose, fast evolving, heterogeneous Linux stack expands the Trusted Computing Base (TCB) and the volume of derived artifacts and dependencies, leading to higher certification cost and schedule risk..

6.6.3. Avionics Impact

Under DO-178C, certification cost is driven not by software licensing fees but by the scale and stability of the assurance evidence required. Cost grows with:

- Evidence volume across the full chain of requirements, design, code, verification, coverage, and change-impact artifacts.
- Independence and quality-assurance activities including Quality Assurance /Independent Verification and Validation(QA/IV&V) needed to satisfy DAL A/B objectives.
- Configuration and baseline stability, since every change triggers impact analysis and regression testing.
- Supplier and version control, including long-term reproducibility, auditability, and traceability of all build inputs.

These factors increase certification burden rather than reducing cost, despite the absence of licensing fees.

Table 2. Common misconceptions about Linux for safety-critical systems.

Misconception	Concise Refutation	Section
PREEMPT_RT ⇒ determinism	Improves latency; cannot bound worst-case timing	§6.1
Cgroups/VMs ⇒ partitions	CPU share only; no fixed windows or isolation	§6.2
mlock()/no-swap ⇒ isolation	Residency ≠ exclusivity; mappings still change	§6.3
Static config ⇒ deterministic	Static layout ≠ static semantics; kernel remains dynamic	§6.4
Rich ecosystem ⇒ cert-friendly	Toolchain heterogeneity breaks DO-330 traceability	§6.5
Open source ⇒ reduces cost	Certification effort scales with verification scope, not license fee	§6.6

As shown above, Linux’s unsuitability for DAL A/B cannot be “engineered away” through incremental hardening, configuration tightening, or other minor-to-moderate kernel modifications; the architectural blockers identified in this paper remain. If changes are made extensive enough to satisfy avionics operating-system assumptions (closed-world semantics, analyzable partition-level timing, and provable spatial/fault isolation), the result ceases to be meaningfully “Linux” in the

COTS/upstream sense and becomes a totally different operating system with a new certification burden.

7. Recommendations

This section provides minimal, certification-driven guidance rather than a comprehensive OS survey.

Building on the architectural analysis and certification-oriented argument developed in this work, this section provides actionable and standards-aligned recommendations for organizations evaluating or selecting operating-system platforms for DAL A/B avionics.

7.1. Prefer Partitioned Architectures over General-Purpose OSes

Safety-critical avionics require closed-world execution semantics, strong spatial separation, and bounded temporal behavior. Organizations should therefore adopt one of the following architectural classes, all of which are engineered to satisfy ARINC 653 and DO-178C principles:

7.1.1. ARINC 653 Partitioning Kernels / Type-1 Hypervisors

Suitable platforms include:

- XtratuM + LithOS
- POK
- JetOS

These systems provide:

- Table-driven major/minor-frame scheduling
- Hardware-enforced spatial isolation
- Minimal separation-kernel TCB
- Deterministic inter-partition IPC

7.1.2. High-Assurance Separation Kernels with Userspace ARINC Services

Preferred when formal verification or ultra-small TCB is needed:

- Muen SK

The architectures offer:

- Minimal trusted computing base (order of 10–20 KLOC)
- Strict capability-based authority
- Provable fault isolation
- Support for deterministic mixed-criticality scheduling

7.1.3. Commercial Certifiable RTOS Platforms

When product maturity, vendor support, and certification packages are required:

- VxWorks 653
- INTEGRITY-178B
- LynxOS-178
- PikeOS
- DeOS

These products include:

- Controlled and frozen baselines
- Vendor-qualified toolchains
- Well-established certification artifacts
- Deterministic partitioning services

7.2. Enforce Minimal Trusted Computing Base (TCB) as a Primary Design Objective

From earlier analysis, Linux's monolithic TCB is a certifiability blocker. Future airborne systems should adopt the following TCB-reduction strategies:

7.2.1. Keep the Separation Kernel Minimal

Only scheduling, memory isolation, and IPC belong in the trusted kernel.

Push device drivers and services to unprivileged partitions:

Use IOMMU/MPU/VT-d mechanisms to sandbox drivers in their own partitions.

- Avoid shared kernel global state and shared writable memory:
The platform architecture must enforce non-transitive fault containment.
- Ensure deterministic and static memory mappings:
No demand paging, no dynamic reclaim, no runtime compaction, no page migration.
A minimal, analyzable TCB is the single most effective way to reduce certification cost and risk.

7.3. Strengthen System-Architecture Governance for Mixed-Criticality Designs

For emerging electric aviation, low-altitude UAS, and UAM platforms—main areas where Linux misuse is common—the following governance measures are recommended:

- Enforce architectural separation of DAL A/B
- Require early airworthiness review for OS/platform selection
- Document OS-selection rationale in the system PSAC artifacts

7.4. Additional Recommendations

To ensure certifiable, deterministic, partition-safe airborne computing:

- Adopt a narrow, stable, vendor-qualified toolchain.
- Freeze baselines early and maintain long-term configuration control.
- Develop organizational capability in standards-based assurance.

This set of recommendations converts the paper's architectural analysis into a clear, practical, and certification-aligned decision framework.

8. Conclusions

As demonstrated in this work, Linux's architecture and life-cycle model cannot satisfy the determinism, isolation, configuration stability, or verification rigor mandated by DO-178C. Architecturally, Linux retains open-world execution semantics (Section 4.1), lacks analyzable temporal isolation and worst-case timing bounds (Section 4.2), cannot guarantee immutable hardware-enforced physical memory partitions due to mutable mappings (Section 4.3), and does not provide strong fault-containment boundaries because faults can propagate through shared kernel state (Sections 4.4–4.5). These technical properties, combined with the absence of fixed baselines, controlled development processes, and certifiable assurance evidence at scale, prevent Linux-based systems from meeting DO-178C objectives.

Although Linux remains an effective platform for prototyping and offers substantial practical advantages, which make it particularly appealing to new commercial entrants in the emerging low-altitude aviation market, it must not be employed as the operating basis for flight-critical DAL A/B systems. As the low-altitude aviation sector continues to expand and attract cross-industry participants, it is essential that system architecture decisions remain aligned with the safety-first principles underlying DAL A/B certification. Long-term vital safety cannot be traded for short-term development convenience, and high-integrity platforms designed for deterministic, partitioned execution remain indispensable for the next generation of certifiable airborne systems.

References

1. ARINC Industry Activities, ARINC Specification 653P1-3: Avionics Application Software Standard Interface, Part 1, Annapolis, MD, USA: ARINC, 2015.
2. ARINC Industry Activities, ARINC Specification 653P3-2: Avionics Application Software Standard Interface, Part 3, Annapolis, MD, USA: ARINC, 2014.
3. RTCA Inc., DO-178C: Software Considerations in Airborne Systems and Equipment Certification, Washington, DC, USA: RTCA, 2011.
4. EUROCAE, ED-12C: Software Considerations in Airborne Systems and Equipment Certification, Paris, France: EUROCAE, 2011.
5. RTCA Inc., DO-297: Integrated Modular Avionics (IMA) Design Guidance and Certification Considerations, Washington, DC, USA: RTCA, 2005.
6. RTCA Inc., DO-330: Software Tool Qualification Considerations, Washington, DC, USA: RTCA, 2011.
7. SAE International, ARP4754A: Guidelines for Development of Civil Aircraft and Systems, Warrendale, PA, USA: SAE, 2010.
8. P. Wang, Q. Li, & H. Xiong, "Time and space partitioning technology for integrated modular avionics systems," *J. Beijing Univ. Aeronaut. Astronaut.*, vol. 38, no. 6, pp. 721–726, 2012 (in Chinese).
9. F. He, H. Xiong, & X. Zhou, "Overview of key technologies for ARINC 653 partitioned operating systems," *Acta Aeronaut. Astronaut. Sin.*, vol. 35, no. 7, pp. 1777–1796, 2014 (in Chinese).
10. Y. Li, T. Zhou, & J. Li, "Research and implementation of airborne ARINC 653 partition operating system," *Comput. Eng. Appl.*, vol. 51, no. 20, pp. 235–240, 2015 (in Chinese).
11. L. Chen, "Research on deterministic scheduling of avionics partition operating systems," Ph.D. dissertation, Coll. Aeronaut. Eng., Nanjing Univ. Aeronaut. Astronaut., Nanjing, China, 2018 (in Chinese).
12. R. Huang, "Research on ARINC 653 partition isolation mechanism for IMA," Ph.D. dissertation, Sch. Electr. Eng., Northwestern Polytech. Univ., Xi'an, China, 2020 (in Chinese).
13. I. Lopez, P. Parra, M. Uruña, et al., "XtratuM: a hypervisor for partitioned embedded real-time systems," in *Proc. 18th Int. Conf. Real-Time Netw. Syst. (RTNS)*, Paris, France: ACM, 2010, pp. 1–6.
14. A. Crespo, P. Metge, & I. Lopez, *LithOS: A Guest OS for ARINC 653 on XtratuM Hypervisor*, Valencia, Spain: Univ. Politèc. Valencia, 2012.
15. J. Delange, L. Pautet, & S. Faucou, "POK: an ARINC 653 compliant operating system for high-integrity systems," in *Reliable Software Technologies – Ada-Europe 2010*, Berlin, Germany: Springer, 2010, pp. 172–185.
16. B. Huber, A. Lackorzynski, A. Warg, et al., "seL4: formal verification of a high-assurance microkernel," *Commun. ACM*, vol. 57, no. 3, pp. 107–115, 2014.
17. I. Kuz, K. Elphinstone, G. Heiser, et al., "MCS: temporal isolation in the seL4 microkernel," in *Proc. 11th Oper. Syst. Platforms Embedded Real-Time Appl. (OSPERT)*, New York, NY, USA: IEEE, 2015, pp. 1–6.
18. H. Härtig, A. Lackorzynski, & A. Warg, *The Muen Separation Kernel: Design and Formal Verification*, Dresden, Germany: Tech. Univ. Dresden, 2018.
19. J. Rushby, *Design and Verification of Secure Systems*, Menlo Park, CA, USA: SRI Int., 1981.
20. J. Rushby, "A kernelized architecture for safety-critical systems," in *Proc. IFIP Congr.*, Vienna, Austria, 1999, pp. 1–6.
21. Wind River Systems Inc., *VxWorks 653 Platform Datasheet*, [Online]. Available: <https://www.windriver.com>, 2022.
22. Green Hills Software Inc., *INTEGRITY-178B RTOS for Avionics*, [Online]. Available: <https://www.ghs.com>, 2021.
23. SYSGO AG, *PikeOS Safety-Certifiable RTOS and Hypervisor*, [Online]. Available: <https://www.sysgo.com>, 2024.
24. DDC-I Inc., *DeOS Safety-Critical RTOS*, [Online]. Available: <https://www.ddci.com>, 2024.
25. D. Bovet, & M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA, USA: O'Reilly Media, 2005.
26. R. Love, *Linux Kernel Development*, Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
27. M. Gorman, *Understanding the Linux Virtual Memory Manager*, Upper Saddle River, NJ, USA: Prentice Hall, 2004.

28. The Linux Kernel Organization, *Linux Scheduler Documentation*, [Online]. Available: <https://docs.kernel.org/scheduler/>, 2024.
29. The Linux Kernel Organization, *Linux Memory Management Documentation*, [Online]. Available: <https://docs.kernel.org/mm/>, 2024.
30. T. Gleixner, *PREEMPT_RT Patch Overview and Design Philosophy*, San Francisco, CA, USA: Linux Foundation, 2019, [Online]. Available: <https://wiki.linuxfoundation.org/realtime/start>.
31. The Linux Kernel Organization, *kbuild: The Linux Kernel Build System*, [Online]. Available: <https://docs.kernel.org/kbuild/>, 2024.
32. The Yocto Project, *Yocto Project Mega-Manual*, [Online]. Available: <https://www.yoctoproject.org>, 2024.
33. Device Tree Working Group, *Device Tree Specification*, [Online]. Available: <https://www.devicetree.org>, 2024.
34. H. Zhao, S. Gao, & Y. Yang, "Applicability analysis of airborne software based on Linux real-time extension," *Comput. Eng.*, vol. 43, no. S1, pp. 311–315, 2017.
35. a653rs Contributors, *a653rs-linux: ARINC 653 Emulation on Linux*, [Online]. Available: <https://github.com/a653rs>, 2024.
36. ELISA Project, *FAQs – ELISA: Enabling Linux in Safety Applications*, [Online]. Available: <https://elisa.tech/about/faqs/>, Accessed: 2026-03-23.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.