Article

Not peer-reviewed version

The Hidden Risks of Using Linux in Aviation Systems

Haoran Lu *

Posted Date: 13 March 2026

doi: 10.20944/preprints202603.0354.v3

Keywords: ARINC 653; DO-178C; DO-330; fault isolation; Linux; partition; real-time determinism; system semantics

 doi:10.20944/preprints202603.0354.v3

Article

The Hidden Risks of Using Linux in Aviation Systems

Haoran Lu

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

Abstract

This paper presents a certification-oriented, system-level argument that Linux is fundamentally 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 and ARINC 653. The framework turns architectural properties into concrete certification risks and provides actionable guidance for OS selection and governance in integrated modular avionics.

Keywords: ARINC 653; DO-178C; DO-330; fault isolation; Linux; partition; real-time determinism; 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 practices 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. These platforms are explicitly designed around ARINC 653 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.

However, a large and rapidly growing group of resource-constrained new entrants faces a fundamentally 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—

 doi:10.20944/preprints202603.0354.v3

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 ARINC 653 or 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.

The situation is further complicated by periodic discussions within international aviation communities—such as exploratory studies by regulators, research institutions, or large OEMs—regarding the potential role of open-source platforms in future avionics architectures. Although these studies typically examine Linux from a research or non-certification perspective, their existence is frequently misinterpreted by inexperienced organizations as evidence that Linux can, with sufficient modification, be transformed into a certifiable avionics-grade operating system. This misconception has become particularly widespread in emerging low-altitude aviation sectors, where safety-critical expertise is limited and cost pressures are high.

This paper addresses that misconception directly. By examining Linux through the lens of ARINC 653 partitioning principles and 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, 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 in integrated modular avionics (IMA) environments.

2. Related Work

Research on operating-system 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.

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.

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. Similarly, the seL4 microkernel demonstrates that a minimal kernel design can be formally verified down to C source code, establishing a level of behavioral predictability that greatly exceeds what can be achieved with large monolithic kernels.

 doi:10.20944/preprints202603.0354.v3

More recently, industry initiatives such as the ELISA project (Enabling Linux in Safety Applications) 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.

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 the interaction between kernel-level architectural properties and the explicit requirements defined in DO-178C, DO-330, and ARINC 653.

The analysis proceeds in three structured steps. First, the architectural semantics of Linux—particularly its monolithic kernel structure, dynamic memory-management behavior, and asynchronous subsystem interactions—are characterized using publicly available documentation and prior research on real-time Linux and kernel execution behavior.

Second, these architectural characteristics are mapped to airworthiness requirements related to temporal determinism, spatial partitioning, fault containment, trusted computing base minimization, and toolchain stability, drawing on the constraints formally defined in ARINC 653 and DO-178C/DO-330.

Finally, the identified mismatches are organized into a unified causal structure that explicates how fundamental kernel semantics propagate into certification-blocking outcomes, thereby offering a systematic explanation for Linux's incompatibility with high-integrity airborne environments.

This methodology does not rely on proprietary engineering processes, internal development practices, or implementation-specific techniques. Instead, it synthesizes openly documented architectural properties with established certification requirements to construct an evidence-based system-level argument about certifiability. Such an approach is widely used in safety-critical systems engineering and avoids the need for domain-specific proprietary methods or experimental data.

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.

 doi:10.20944/preprints202603.0354.v3

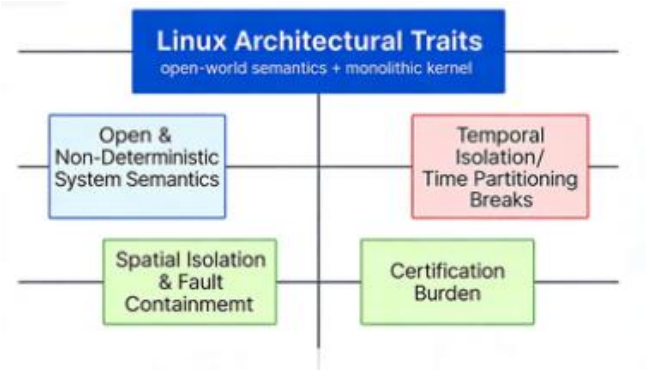

Figure 1. Classification of key Linux architectural traits.

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 core kernel 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 ARINC 653'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 root cause of its subsequent temporal and spatial-isolation failures, as it inherently precludes the static, bounded, and fully verifiable behavior required for high-integrity DAL A/B avionics systems.

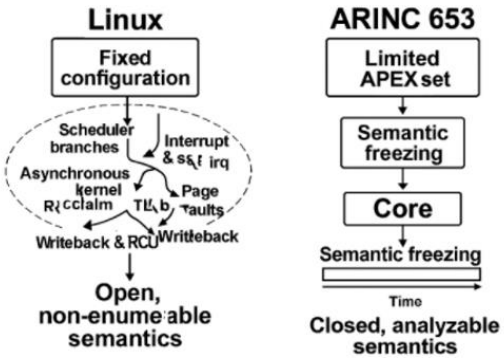

Figure 2. Semantic unfreezing.

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

 doi:10.20944/preprints202603.0354.v3

additional environment-dependent execution paths that cannot be captured within a closed-world model.

This open-world execution model directly violates the deterministic and predictable-behavior requirements for safety-critical aerospace functions stipulated in SAE ARP4754A (2010) Clause 6.4.3—the top-level system-engineering guideline underpinning DO-178C software-certification criteria.

4.2. Lack of Temporal Determinism

Deterministic execution requires that the system provide 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 provide 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.

In Linux, temporal nondeterminism arises from two architecturally distinct mechanisms whose effects on execution timing must be analyzed separately. Although both may ultimately manifest as preemption or execution delay, their underlying causes, triggering conditions, and analyzability properties are fundamentally different.

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.

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 worst-case execution time analytically unbounded.

Thus, synchronous nondeterminism originates from Linux's time-sharing scheduling model combined with variable-length non-preemptible kernel paths, independent of any asynchronous device or kernel events.

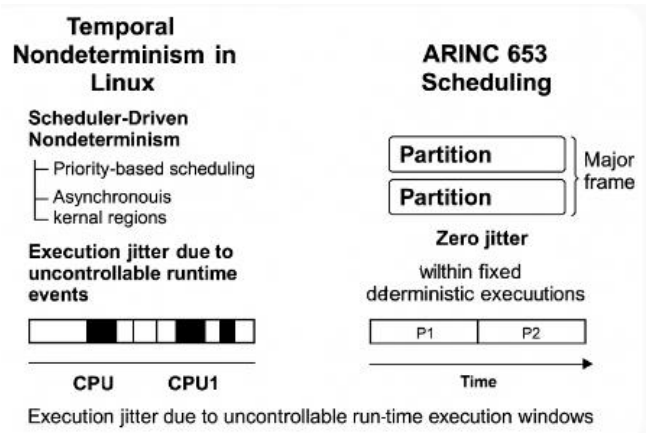

Figure 3. Scheduler-driven execution jitter in Linux versus deterministic ARINC 653 partition.

 doi:10.20944/preprints202603.0354.v3

4.2.2. Event-Driven (Asynchronous) Nondeterminism

Separate from scheduler-driven effects, Linux also contains multiple asynchronous execution sources—including hardware interrupts, 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 (e.g., softirq/ksoftirqd), 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.

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 fundamentally incompatible with avionics software.

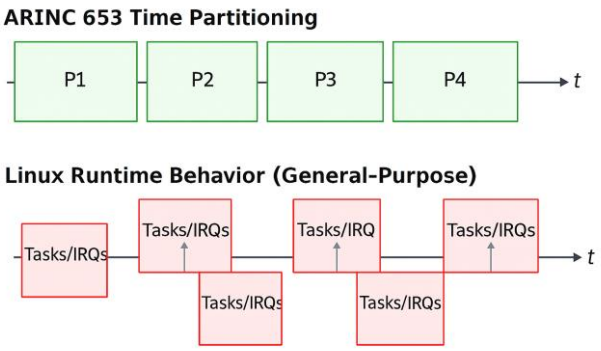

Figure 4. Event-driven execution jitter in Linux versus ARINC 653 fixed time partitions.

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 and partitioned-kernel architectures (e.g., ARINC 653) assume 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.

On the other hand, Linux relies on a dynamic and mutable virtual-memory architecture that violates these assumptions. 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 TLB shootdowns. Linux frequently updates page attributes, permission bits, and mapping structures, triggering cross-CPU TLB invalidations and modifying the effective physical-memory layout during operation.

 doi:10.20944/preprints202603.0354.v3

- Shared kernel-memory structures. The kernel's slab allocators, per-CPU buffers, and driver subsystems allocate and free kernel memory dynamically; these regions are globally shared and not partition-scoped.

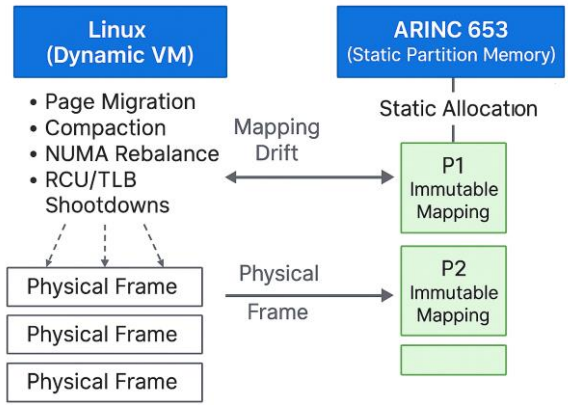

Figure 5. Memory mapping drift.

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.

Consequently, Linux cannot establish the immutable physical-memory boundaries required for DO-178C DAL A/B and cannot meet the spatial-isolation guarantees expected of ARINC 653-style partitioning systems. The kernel's dynamic memory-mapping semantics inherently preclude the formation of independently verifiable, physically isolated memory partitions.

Thus, Linux's dynamic page-table management fundamentally conflicts with the DO-178C requirement that a partition's physical memory be statically allocated, hardware-isolated, and invariant throughout system operation.

4.4. Driver Contamination of Kernel Global State

Linux device drivers execute with full kernel privilege, sharing the monolithic kernel's global address space and core data structures. As a result, a single defective driver can inadvertently corrupt system-wide shared state, including:

- scheduler queues, such as per-CPU run queues and scheduling structures, whose corruption impacts all tasks sharing the CPU;
- memory-management metadata , including allocator structures and slab lists, leading to cross-subsystem memory corruption;
- timing-critical kernel variables, such as jiffies or timer-wheel structures, whose misuse destabilizes system-wide timing behaviour.

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 contain or isolate erroneous behaviour. This propagation pathway is fundamentally incompatible with avionics fault-containment principles, where failures within one component must not compromise the integrity or availability of others.

 doi:10.20944/preprints202603.0354.v3

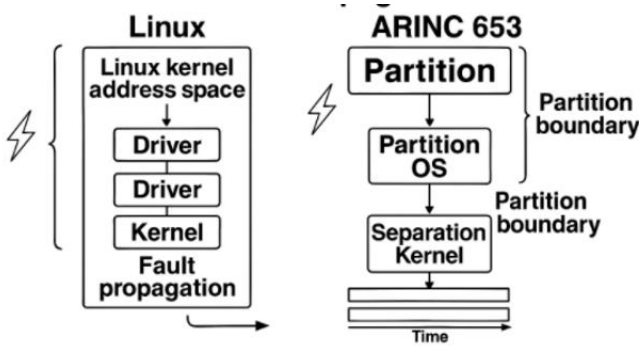

Figure 6. Kernel containing driver.

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, interrupt, memory-management action, driver interaction, and scheduling decision. 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.

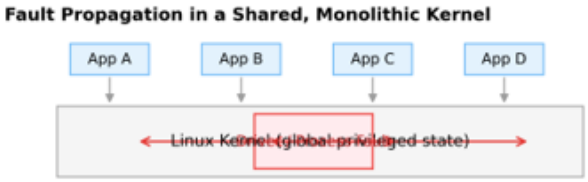

Figure 7. Fault Propagation.

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 hardware-enforced spatial isolation and partition-level fault containment required for DAL A/B airborne software under DO178C.

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;

 doi:10.20944/preprints202603.0354.v3

- 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;
- 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/DO-297 fault-propagation analysis for DAL A/B functions.

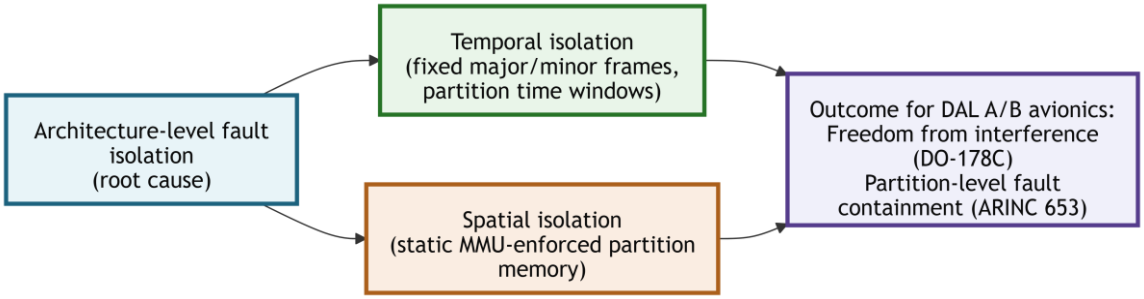

Figure 8. Relationship among architecture-level fault isolation and temporal/spatial isolation for DO-178C DAL A/B.

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. Because Linux operates as a monolithic privileged kernel, all resident components form part of the Trusted Computing Base (TCB). Under avionics standards such as DO-178C and DO-297, 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 fundamental 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, and toolchain behaviors.

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.

 doi:10.20944/preprints202603.0354.v3

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 fundamentally conflicts with this expectation:

- The kernel contains vast quantities of implementation-driven code, developed incrementally without DAL-style requirements decomposition,
- Core subsystems (scheduler, MM, VFS, network stack, timers, softirq) lack formalized HLR/LLR /HLD/LLDspecifications, making traceability impossible,
- Architectural behavior depends on dynamic global state, hardware-dependent heuristics, and runtime conditions, violating DO-178C expectations of predictable and reviewable design behavior,
- Many kernel paths have implicit behavior (e.g., locking, 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.

Linux's scale makes these obligations infeasible:

- The kernel's millions of lines of code require astronomical verification effort,
- Many kernel paths depend on hardware behavior, timing, interrupts, speculative execution, and concurrency, making complete test coverage impossible,
- Dynamic subsystems (e.g., memory reclaim, workqueues, softirq, RCU) make it impractical to achieve deterministic coverage closure, because behavior varies with load, timing, and configuration,
- MC/DC on the kernel would require analyzing tens of thousands of complex decision points, many interacting across subsystems.

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

At the same time, DO-178C requires that all DAL A/B code be traceable to requirements and fully exercised by verification, leaving no dead code, no unintended functionality, and no unverified execution paths. Structural-coverage objectives—up to MC/DC—require that every implemented path be reachable and justified.

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 these paths cannot be deterministically exercised or fully covered, and often have no requirement-level justification.

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

 doi:10.20944/preprints202603.0354.v3

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, 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; however, 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.

Linux violates these expectations because:

- Its development is distributed across thousands of contributors with no DAL-style independence.
- 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.

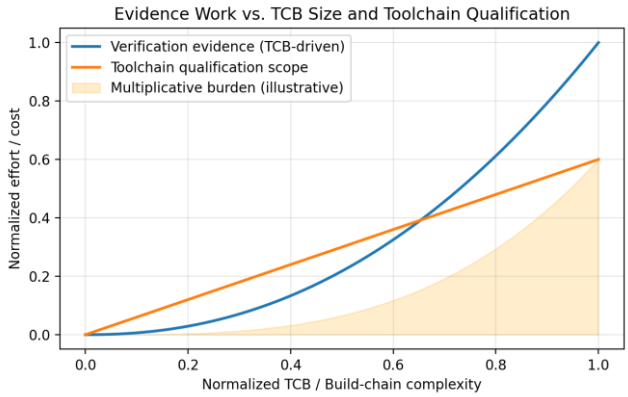

Figure 9. TCB & Toolchain: The Certification Cost Curve.

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 certification explosion that characterizes monolithic kernels like Linux.

4.7. Continuous Patch Stream Destabilizes Certified Baselines

Linux evolves at a rapid pace, and maintaining a stable, certifiable baseline is fundamentally 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, including:

- locking primitives

 doi:10.20944/preprints202603.0354.v3

- interrupt and softirq threading
- scheduling heuristics
- preemption and concurrency models

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 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—conditions that Linux's continuous patch stream cannot meet for DAL A/B certification.

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.

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. No meta-build layers, no automatic generators, and no distribution-level tool dependencies exist.

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 leads to open-world and highly dynamic system semantics, which cannot be finitely bounded, frozen, or formally restricted at integration time.

 doi:10.20944/preprints202603.0354.v3

Two Primary Consequences From this root cause:

5.1. Airworthiness Infeasibility

The system cannot satisfy the determinism, stability, and evidence requirements demanded by DO-178C and ARINC 653.

Consequences Derived from Airworthiness Infeasibility. The first primary consequence are 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:

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.

Together, 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.

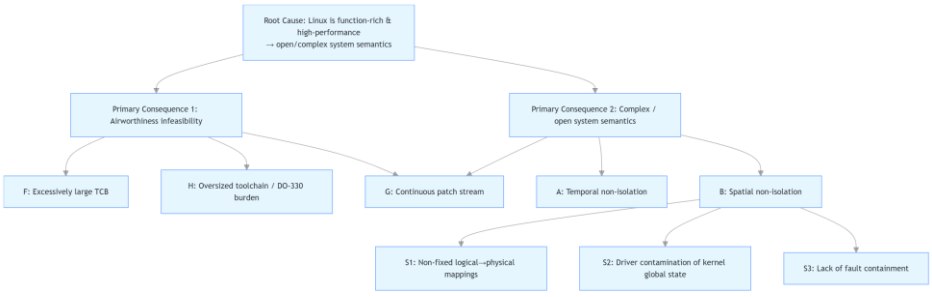

Figure 10. Together, these results show that all eight observations (A–H) are not isolated weaknesses but systemic consequences of a single architectural root cause. The causal chain closes the loop from Linux's open-world, dynamic semantics to certification-blocking outcomes: unprovable temporal bounds, unassurable spatial and fault isolation, TCB explosion, unstable baselines, and prohibitive DO-330 obligations. This unified explanation converts architectural properties into concrete certification risks under DO-178C, and provides a replicable decision aid for avionics OS selection, governance, and lifecycle assurance.

 doi:10.20944/preprints202603.0354.v3

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 eliminate the asynchronous, workload-dependent kernel pathways identified in §4.1. Softirq cascades, unbounded memory-management events (page faults, reclaim, compaction, NUMA migrations, TLB shootdowns), dynamic scheduler behavior, and driver-induced jitter all remain present and unbounded.

6.1.3. Why This Matters for Avionics

ARINC 653 requires a closed-world, finitely analyzable timing model. Because PREEMPT_RT cannot bound worst-case latency for kernel subsystems, Linux cannot provide partition-level temporal guarantees or static schedulability proofs for DAL A/B functions.

6.1.4. Technical Explanation

- Softirqs may re-arm and cascade, producing unbounded execution chains.
- MM events contain non-preemptible critical sections and global synchronization.
- IRQ-exit softirqs and TLB shootdowns outrank RT tasks and cannot be preempted.
- Driver execution time depends on firmware, DMA completion, and lock contention.
- DVFS transitions and CPU C-state exits introduce microarchitectural stalls with unbounded latency, cannot be preempted by RT tasks, and occur outside the scheduler's control.
- Linux's dynamic scheduler behavior—including load balancing, task migration, wakeup-preemption rules, and kernel housekeeping threads—continues to introduce runtime-dependent interference that cgroups cannot constrain into fixed, partition-like execution windows.
- Real-time scheduling in Linux follows an ASAP execution model—tasks run as soon as they become runnable rather than within fixed, predetermined time windows—making Linux's real-time behavior fundamentally incompatible with DO-178C timing-analysis requirements for determinism and worst-case execution boundability.

6.2. Containers/VMs/cgroups = Partitioning

6.2.1. Misconception

Namespaces/containers/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.

6.2.3. Why This Matters for Avionics

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:

 doi:10.20944/preprints202603.0354.v3

- **Temporal isolation fails** because Linux's dynamic scheduler (load balancing, migration, wakeup-preemption, kthreads, IRQ activity) introduces runtime-dependent jitter that no cgroup controller can bound.
- **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.

6.2.4. Technical Explanation

- Cgroups control CPU share over long time intervals but cannot enforce fixed, exclusive execution windows.
- Linux's dynamic scheduler (load balancing, migrations, wakeup-preemptions, kernel housekeeping threads) continues to introduce unbounded interference beyond cgroup control.
- Interrupts, softirqs, kernel threads, MM events, and driver execution all run outside cgroup boundaries and freely preempt or delay tasks inside the cgroup.
- Cgroups limit memory usage but do not reserve immutable physical memory; pages can still be migrated, compacted, reclaimed, or remapped by the kernel.
- All cgroups share the same kernel global state and driver domain, preventing DAL A/B-grade fault containment.

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

6.3.1. Misconception

Locking pages with mlock() or disabling swap is sufficient to guarantee deterministic memory behavior and physical isolation for safety-critical tasks.

6.3.2. Reality

Disabling swap only prevents paging to disk; it does not provide fixed physical memory allocation or prevent the kernel from reclaiming, migrating, compacting, or remapping physical pages. Similarly, mlock() keeps pages resident in RAM but does not guarantee physical placement, prevent page-table updates, or isolate the locked pages from global memory-management activity.

 doi:10.20944/preprints202603.0354.v3

Linux continues to treat physical memory as a shared global pool, and the kernel may transparently migrate, compact, or invalidate pages—even pages that are locked in memory. None of these mechanisms can be bounded, controlled, or associated with ARINC 653-style hardware-enforced memory partitions.

6.3.3. Why This Matters for Avionics

ARINC 653 requires immutable physical memory regions per partition, enforced by hardware (MMU/MPU), and no dynamic page-table changes during execution. Because Linux's memory subsystem performs demand paging, reclaim, compaction, NUMA balancing, TLB shootdowns, and page-table updates at runtime—even with swap disabled and pages locked—memory isolation and WCET analysis remain impossible under DO-178C DAL A/B.

Thus, mlock() and disabling swap do not create deterministic memory behavior and cannot satisfy aviation-grade spatial-isolation requirements.

6.3.4. Technical Explanation

- Disabling swap prevents paging to disk but does not freeze physical page allocation; the kernel may still reclaim, migrate, or remap pages under memory pressure.
- mlock() preserves page residency but does not ensure fixed physical placement, preventing page migration or compaction.
- Linux may modify PTEs for locked pages, triggering TLB shootdowns that preempt even the highest-priority real-time tasks.
- NUMA balancing and memory compaction remain active, producing unbounded, workload-dependent memory-access latency.
- ARINC 653 requires statically defined, hardware-enforced physical memory regions, which cannot be emulated by logical memory limits or mlock()-based residency controls.

6.4. Static Configuration Can 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 layouts, 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 schedule softirqs, initiate memory-management actions, activate timers and RCU callbacks, respond to driver or firmware events, and reorder internal operations in ways 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.

 doi:10.20944/preprints202603.0354.v3

6.4.3. Why This Matters for Avionics

ARINC 653 and DO-178C assume that system behavior follows a closed-world, finitely analyzable semantics: all execution paths must be known at integration time, and no new behaviors may emerge at runtime. Linux violates this assumption architecturally.

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.

its execution semantics remain incompatible with WCET analysis, partition-level determinism, and DAL A/B timing isolation. Static application-layer restrictions cannot remove these semantic properties.

6.4.4. Technical Explanation

- Fixing the process/thread set does not prevent the kernel from generating new execution paths through softirq activation, timer events, RCU quiescence cycles, or driver callbacks.
- Static allocation of stacks or memory does not eliminate dynamic memory-management behavior; page migration, compaction, PTE updates, and TLB shootdowns still occur based on runtime state, not static declarations.
- Masking interrupts or restricting user-visible interrupt sources does not suppress IRQ-exit softirqs, DMA completions, firmware-initiated events, or driver-level asynchronous activity.
- Removing signals or freezing IPC patterns limits application-level nondeterminism but leaves kernel-level nondeterminism untouched: kernel threads, scheduler housekeeping, NUMA balancing, and deferred work continue to run according to dynamic internal conditions.
- 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.

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, not harder, 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.

Unlike certifiable ARINC 653 RTOS platforms—which deliberately minimize the toolchain to a small, stable, vendor-controlled set—Linux's ecosystem evolves continuously and lacks the tight configuration control required for certification. The ecosystem's richness improves productivity but degrades certifiability.

6.6.3. Why This Matters for Avionics

DO-178C and DO-330 require that all tools involved in producing airborne software be qualified, justified, or placed under strict configuration control. Linux systems typically involve:

- multiple interacting toolchains (GCC/LLVM, Clang, binutils),

 doi:10.20944/preprints202603.0354.v3

- meta-build frameworks (Yocto, Buildroot, CMake),
- code generators and device-tree compilers,
- Python/shell-based build scripts,
- subsystem-specific build steps (e.g., kbuild, kernel modules),
- and fast-moving upstream dependencies.

This results in a combinatorial explosion of tool interactions that cannot be economically qualified for DAL A/B programs. In contrast, ARINC 653 systems rely on frozen, single-vendor, certifiable toolchains with minimal transformation steps and long-term stability.

Thus, the richness of the Linux ecosystem—an advantage for general engineering—becomes a certification burden under DO-178C/DO-330 and is fundamentally misaligned with the lifecycle stability expected in airborne systems.

6.6.4. Technical Explanation

- Linux relies on numerous toolchains and generators—GCC/LLVM, binutils, kbuild, Kconfig, CMake, Yocto, Debian, device-tree compilers, and scripting frameworks—each of which increases the DO-330 qualification footprint.
- Frequent upstream changes produce version churn, preventing a stable, certifiable baseline and forcing repeated tool qualification and regression analysis.
- Many build steps involve multi-stage code generation and scripts that are not amenable to DO-178C traceability or source-to-object code consistency requirements.
- ARINC 653 RTOS platforms deliberately use minimal, vendor-qualified toolchains to keep configuration control tractable and adapt to w, while Linux's ecosystem expands the certification scope beyond feasibility.

6.7. Open Source = Reduces Cost

6.7.1. Misconception

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

6.7.2. Reality

Licensing is a small fraction of DO-178C/DO-330 program cost. What dominates cost is the amount of software and tools that must be planned, traced, verified, qualified, controlled, and re-verified over the lifecycle. A general-purpose, fast-evolving, heterogeneous Linux stack expands the Trusted Computing Base (TCB) and the volume of generated artifacts. The result is higher certification cost and schedule risk.

6.7.3. Why This Matters for Avionics

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, tests, 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 and unpredictability rather than reducing cost, despite the absence of licensing fees.

 doi:10.20944/preprints202603.0354.v3

6.7.4. Technical Explanation

- TCB amplification: A monolithic, feature-rich kernel (plus drivers, subsystems, tracing, net/FS stacks) turns millions of lines of code(LOC) into certification scope; evidence and coverage scale with TCB size, not license price.
- Tool-qualification footprint: Multi-stage generators (kbuild, Kconfig, DT compilers), scripting pipelines, and multiple toolchains (GCC/LLVM/binutils) enlarge DO-330 obligations; each transformation must be justified or qualified.
- Evidence churn: Continuous upstream patching (including security backports) invalidates prior results; every change requires DO-178C change-impact analysis and regression, driving recurring cost.
- Machine-code variance: Configuration-dependent code paths and auto-generated artifacts produce binary differences that are hard to trace back to requirements, complicating MC/DC closure and repeatable builds.
- Baseline stability: Open, rapidly evolving ecosystems hinder formation of a frozen, audit-ready certifiable baseline; schedule risk increases as re-verification becomes open-ended.
- Supplier control and independence: Distributed community development lacks DAL-style process independence and supplier QA controls; programs must recreate that governance themselves—another real cost center.
- Net effect: License-free does not mean certification-cheap; for DAL A/B, open-source breadth typically raises total cost of assurance (TCoA) despite zero 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**

7. Recommendations

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. The recommendations address platform architecture, development governance, toolchain control, and lifecycle assurance—the four domains most affected by Linux's structural incompatibilities.

 doi:10.20944/preprints202603.0354.v3

7.1. Adopt Architectures Explicitly Designed for Determinism and Isolation

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/DO-297 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:

- seL4 (MCS)
- Muen SK

These 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:

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.

 doi:10.20944/preprints202603.0354.v3

- 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/ARP4754A artifacts

8. Conclusions

Although Linux offers substantial practical advantages—rich functionality, extensive hardware enablement, mature toolchains, and rapid development turnaround—these characteristics make it particularly appealing to new commercial entrants in the emerging low-altitude aviation market. For organizations without prior experience in safety-critical development, Linux can appear to provide a short and efficient path to early prototypes, enabling rapid demonstrations and fast iteration. However, for DAL A/B avionics, such short-term benefits cannot override the fundamental requirement that safety is the primary design objective.

As demonstrated in this work, Linux's architecture and life-cycle model cannot satisfy the determinism, isolation, configuration stability, or verification rigor mandated by ARINC 653 and DO-178C. The absence of fixed baselines, controlled development processes, and certifiable assurance evidence means that systems built on Linux cannot meet DAL A/B objectives, regardless of additional testing or late-stage mitigation. Airworthiness is cumulative and process-driven; it cannot be recovered after an unsuitable foundation has been chosen.

Therefore, while Linux remains an effective platform for prototyping and non-critical mission functions, it must not be employed as the operating basis for flight-critical 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 airworthiness 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.

The safety principles discussed in this paper originate from universal functional-safety concepts rooted in IEC 61508, which underpin multiple high-integrity domains including avionics. However, the focus of this work remains on their specific implications for airborne systems under ARINC 653 and DO-178C.

References

1. [1] ARINC Industry Activities, ARINC Specification 653P1-3: Avionics Application Software Standard Interface, Part 1, Annapolis, MD, USA: ARINC, 2015.
2. [2] ARINC Industry Activities, ARINC Specification 653P3-2: Avionics Application Software Standard Interface, Part 3, Annapolis, MD, USA: ARINC, 2014.
3. [3] RTCA Inc., DO-178C: Software Considerations in Airborne Systems and Equipment Certification, Washington, DC, USA: RTCA, 2011.
4. [4] EUROCAE, ED-12C: Software Considerations in Airborne Systems and Equipment Certification, Paris, France: EUROCAE, 2011.
5. [5] RTCA Inc., DO-297: Integrated Modular Avionics (IMA) Design Guidance and Certification Considerations, Washington, DC, USA: RTCA, 2005.

 doi:10.20944/preprints202603.0354.v3

6. [6] RTCA Inc., DO-330: Software Tool Qualification Considerations, Washington, DC, USA: RTCA, 2011.
7. [7] SAE International, ARP4754A: Guidelines for Development of Civil Aircraft and Systems, Warrendale, PA, USA: SAE, 2010.
8. [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. [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. [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. [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. [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. [13] I. Lopez, P. Parra, M. Urueña, 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. [14] A. Crespo, P. Metge, & I. Lopez, *LithOS: A Guest OS for ARINC 653 on XtratuM Hypervisor*, Valencia, Spain: Univ. Politèc. Valencia, 2012.
15. [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. [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. [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. [18] H. Härtig, A. Lackorzynski, & A. Warg, *The Muen Separation Kernel: Design and Formal Verification*, Dresden, Germany: Tech. Univ. Dresden, 2018.
19. [19] J. Rushby, *Design and Verification of Secure Systems*, Menlo Park, CA, USA: SRI Int., 1981.
20. [20] J. Rushby, "A kernelized architecture for safety-critical systems," in *Proc. IFIP Congr.*, Vienna, Austria, 1999, pp. 1–6.
21. [21] Wind River Systems Inc., *VxWorks 653 Platform Datasheet*, [Online]. Available: https://www.windriver.com, 2022.
22. [22] Green Hills Software Inc., *INTEGRITY-178B RTOS for Avionics*, [Online]. Available: https://www.ghs.com, 2021.
23. [23] SYSGO AG, *PikeOS Safety-Certifiable RTOS and Hypervisor*, [Online]. Available: https://www.sysgo.com, 2024.
24. [24] DDC-I Inc., *DeOS Safety-Critical RTOS*, [Online]. Available: https://www.ddci.com, 2024.
25. [25] D. Bovet, & M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA, USA: O'Reilly Media, 2005.
26. [26] R. Love, *Linux Kernel Development*, Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
27. [27] M. Gorman, *Understanding the Linux Virtual Memory Manager*, Upper Saddle River, NJ, USA: Prentice Hall, 2004.
28. [28] The Linux Kernel Organization, *Linux Scheduler Documentation*, [Online]. Available: https://docs.kernel.org/scheduler/, 2024.
29. [29] The Linux Kernel Organization, *Linux Memory Management Documentation*, [Online]. Available: https://docs.kernel.org/mm/, 2024.
30. [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. [31] The Linux Kernel Organization, *kbuild: The Linux Kernel Build System*, [Online]. Available: https://docs.kernel.org/kbuild/, 2024.
32. [32] The Yocto Project, *Yocto Project Mega-Manual*, [Online]. Available: https://www.yoctoproject.org, 2024.
33. [33] Device Tree Working Group, *Device Tree Specification*, [Online]. Available: https://www.devicetree.org, 2024.

 doi:10.20944/preprints202603.0354.v3

34. [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. [35] a653rs Contributors, *a653rs-linux: ARINC 653 Emulation on Linux*, [Online]. Available: https://github.com/a653rs, 2024.