Article

Not peer-reviewed version

The Hidden Risks of Using Linux in Aviation Systems

Lu Haoran *

Posted Date: 4 March 2026

doi: 10.20944/preprints202603.0354.v1

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

 doi:10.20944/preprints202603.0354.v1

Article

The Hidden Risks of Using Linux in Aviation Systems

Lu Haoran

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

Abstract

This paper provides a rigorous examination of eight fundamental architectural deficiencies that render the Linux kernel unsuitable for deployment in safety-critical avionics. These deficiencies include inadequate temporal determinism, the absence of physical memory isolation, driver-induced contamination of global kernel state, an excessively large and unbounded Trusted Computing Base (TCB), open and nondeterministic system semantics, insufficient inter rocess fault containment, unstable kernel behavior due to continuous patching, and a highly complex toolchain that imposes prohibitive DO-330 qualification burdens. Through a technical and standards-aligned analysis, this paper demonstrates that Linux cannot satisfy the determinism, verifiability, isolation, and lifecycle stability required for airworthiness certification, making it inherently incompatible with certifiable airborne platforms.

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

---

I. Introduction

The rapid global expansion of advanced air mobility—including eVTOL, UAM, and next-generation IMA—has attracted significant investment and new entrants from multiple regions, with China, the United States, and Europe all experiencing large inflows of capital and engineering talent from adjacent industries such as consumer electronics, robotics, and automotive.

Many of these teams possess strong Linux experience but lack background in airworthiness and functional safety, leading them to view Linux as a natural starting point for airborne system development.

This cross-industry influx has created a recurring pattern:

engineers tend to project their prior commercial-software intuition onto safety-critical avionics, overestimating the transferability of Linux's perceived advantages—rapid prototyping, rich tooling, broad hardware support—while underestimating the certification obligations associated with DAL A/B systems.

This paper clarifies where such assumptions diverge from DAL A/B expectations. Section II summarizes why Linux looks attractive; Section IV formalizes an evaluation framework (input→process→output) and presents eight architectural and lifecycle limitations; Section V distills engineering anti-patterns with a practical checklist.

II. Why Linux Seems Attractive for Avionics

Linux is widely considered appealing for avionics projects due to three core advantages that drive its adoption in commercial and embedded markets:

 doi:10.20944/preprints202603.0354.v1

A. Engineering Richness and Rapid Development

Linux provides broad driver availability, modern networking and debugging infrastructure, and a highly productive development environment. These capabilities allow teams to integrate hardware quickly and build sophisticated features with minimal platform effort.

B. Advances in Real-Time Responsiveness

PREEMPT_RT significantly reduces typical kernel latencies by introducing threaded interrupt handlers, priority-inheritance locks, and preemptible RCU. With many of these mechanisms upstream, Linux is often perceived as suitable for time-sensitive embedded work.

C. Cost, Reuse, and Availability of Expertise

Linux's open-source model eliminates licensing costs and allows reuse of existing toolchains, middleware, and developer expertise across industries, lowering prototyping costs and entry barriers.

III. What Airworthiness Really Requires

Civil aviation certification is fundamentally objective-driven: design assurance must be proportional to failure severity (DAL A–E), supported by rigorous verification, traceability, configuration management, and qualified tools under DO-330. Airworthy systems require upper-bound timing guarantees, fault containment boundaries, closed-world, finitely analyzable system semantics, stable baselines, and reproducible toolchains—properties evaluated at the system-lifecycle level rather than through average-case runtime performance.

IV. Core Limitations of Linux in Safety-Critical Avionics

Linux's incompatibility with safety-critical avionics stems from eight fundamental architectural deficiencies, all of which directly conflict with avionics certification standards (e.g., DO-178C, ARINC 653) and safety principles. Each deficiency is analyzed below, with direct comparisons to avionics-grade platform design requirements.

A. Lack of Temporal Determinism

These advantages aid prototyping and integration but do not by themselves satisfy certifiable avionics requirements; for certification-relevant limitations, see Section IV.

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.

Airborne software certification is driven by system-level guarantees defined by DO-178C and ARINC 653. These guarantees concern predictability, isolation, and verifiability across the entire lifecycle—not average-case behavior.

High-integrity avionics require:

- Deterministic timing with tight WCET bounds and bounded scheduling/interrupt latency.
- Partition-level fault confinement with hardware-enforced separation and no cross-partition propagation.
- A finite, fully analyzable behavioral model (closed semantics) suitable for formal analysis and exhaustive testing.
- Freeze-able, long-term stable baselines with strictly controlled change.
- Version-locked, auditable, DO-330-qualifiable toolchains enabling deterministic, reproducible builds.

 doi:10.20944/preprints202603.0354.v1

These expectations define what any DAL A/B-certifiable platform must meet, irrespective of the operating system. Where general-purpose kernels deviate from these expectations is discussed in Section IV.

Method: Evaluation Framework (Input → Process → Output)

Input: Platform architectural traits & lifecycle controls

- Timing model elements (scheduler preemption paths, interrupt/softirq/workqueues, memory events)
- Spatial isolation mechanisms (MMU/MPU boundaries, page-mapping mutability)
- Failure containment boundaries (kernel/global state sharing)
- Baseline stability (patch cadence, configuration freeze)
- Build/toolchain determinism (multi-tool pipelines, DO-330 qualification scope)

Process: Structured assessment against DAL A/B objectives

- Map traits → objectives: timing determinism, partition-level isolation, closed semantics, stable baselines, qualifiable toolchains
- For each objective: analyze worst-case behavior, fault propagation, state-space boundedness, configuration control, tool evidence
- Rate risk: low/medium/high, with rationale and affected subsystems

Output: Certification-relevant risk posture & guidance

- Consolidated findings per objective (pass/blockers)
- Mitigation options (partitioned kernels, separation kernels, certified RTOS platforms)
- Decision aid: proceed for non-critical functions/redesign for DAL A/B

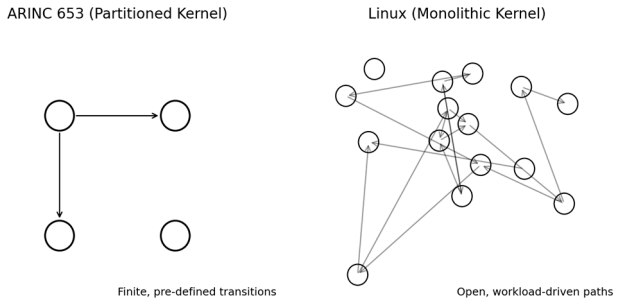

Figure 1. State-space closure comparison: ARINC 653 (finite, pre-defined transitions) vs. Linux (open, workload-driven paths).

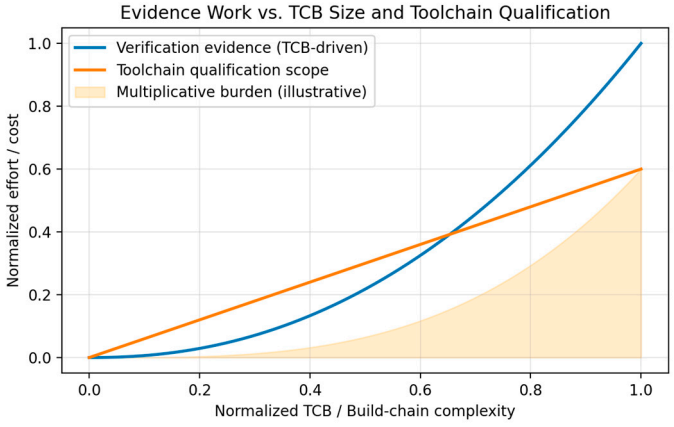

Figure 2. Evidence workload vs. TCB size and toolchain qualification; shaded area illustrates multiplicative burden.

 doi:10.20944/preprints202603.0354.v1

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

- spinlock-protected critical sections,
- per-CPU data updates,
- scheduler state transitions, and
- 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.

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.

In contrast, 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. The timing behavior of Linux versus ARINC 653 is illustrated in Figure 1.

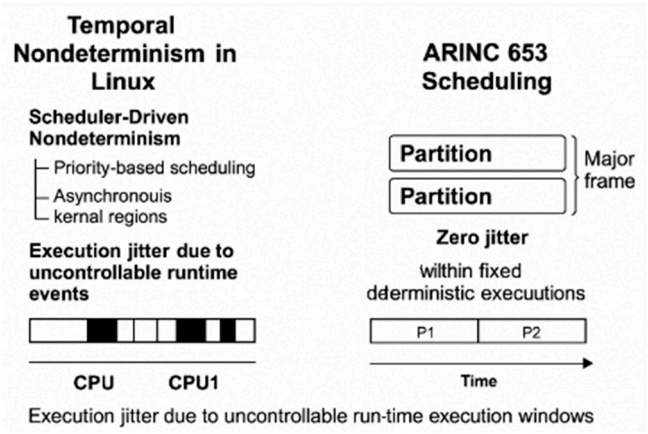

Figure 1. Scheduling determinism.

B. 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.

In contrast, 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:

 doi:10.20944/preprints202603.0354.v1

- 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.
- 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.

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.

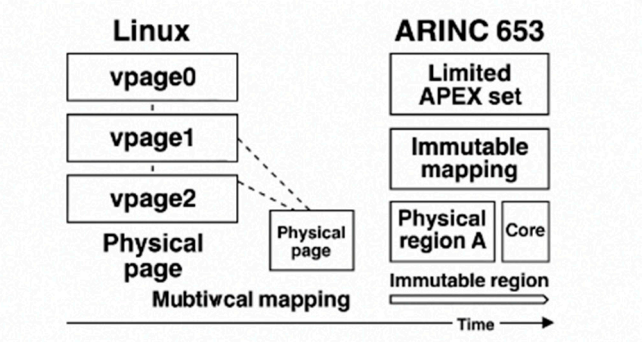

Figure 2. Memory mapping drift.

C. 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-

 doi:10.20944/preprints202603.0354.v1

containment principles, where failures within one component must not compromise the integrity or availability of others.

In contrast, partitioning hypervisors—such as XtratuM—execute device drivers inside hardware-enforced isolated partitions. Each partition is provided with a constrained, virtualized view of system resources, and no driver can modify the separation kernel's trusted domain or the state of other partitions. This architectural boundary prevents driver-induced faults from affecting safety-critical functions, thereby maintaining the strong spatial and fault isolation required for certifiable avionics systems.

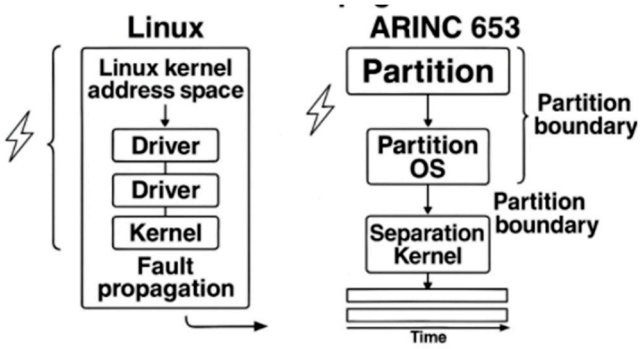

Figure 3. Kernel containing driver.

D. 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.

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.

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

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

 doi:10.20944/preprints202603.0354.v1

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 specifications, 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.

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.

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

DO-178C mandates strict, repeatable configuration management:

- Every version, file, requirement, and tool must be baselined and traceable.
- Any change requires impact analysis, regression evidence, and re-verification for affected DAL objectives.
- Linux's characteristics conflict sharply with these requirements:
- The kernel evolves at a rapid pace with constant patch churn across all subsystems.
- Security fixes, driver updates, and architectural changes modify the kernel's global behavior, invalidating prior baselines.
- Maintaining a stable, frozen Linux baseline contradicts the upstream model and imposes massive regression testing costs.
- The Linux toolchain (kbuild, gcc/llvm, binutils, scripts) forms a complex, multi-tool configuration that must itself be DO-330 qualified for use in DAL A/B—effectively infeasible.

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.

 doi:10.20944/preprints202603.0354.v1

- 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.

Contrast with ARINC 653 TCB Philosophy

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.

E. Open System Semantics with Unpredictable Behavior

Linux exhibits a vast, dynamically evolving set of execution paths whose behavior depends on runtime conditions, workload characteristics, and internal kernel subsystem interactions. These execution paths arise from multiple, largely autonomous kernel mechanisms that operate independently of application intent and cannot be enumerated or frozen at integration time.

1). Asynchronous Kernel Threads

The kernel continuously schedules a variety of global and per-CPU worker threads—such as workqueue workers, softirq daemons, RCU callback threads, per-CPU housekeeping threads, and driver-initiated kernel tasks.

Because these threads are activated based on internal conditions (e.g., deferred work, I/O events, timers, lock contention, and scheduler heuristics), they create non-deterministic, interleaving control flows that expand the system's reachable state space beyond any finite representation.

2). Memory Reclaim and Page-Table Mutations

Linux relies on demand paging, LRU-based memory reclaim, compaction, and page migration, all triggered by dynamic memory pressure and workload-dependent access patterns.

These mechanisms modify page tables, trigger TLB invalidations, relocate physical pages, and alter memory locality during runtime.

Because reclaim and migration events are data-dependent and unpredictable, memory-management behavior cannot be captured by a fixed, analyzable model at integration time.

3). Interrupt Cascades and SoftIRQ Execution

Linux supports nested interrupts, interrupt-driven deferred execution, and softirq cascades (e.g., NET_RX, NET_TX, TIMER, RCU).

The activation order, nesting depth, and execution duration of these paths depend on device activity, interrupt rates, queue backlogs, and internal kernel throttling.

As a result, the global interrupt-handling behavior forms a non-bounded control-flow graph, where execution paths and execution orders cannot be exhaustively enumerated for verification.

4). NUMA Rebalancing and Cross-Node Page Migration

On NUMA systems, Linux periodically performs automatic NUMA balancing, scanning access patterns and relocating memory pages across NUMA nodes.

This process modifies page tables, changes memory locality, and dynamically reshapes memory-access costs across CPUs.

Because these operations depend on real-time access statistics, they introduce additional, load-dependent, non-deterministic transitions into the kernel's execution semantics.

5). RCU State Transitions and Epoch Progression

The Read-Copy-Update (RCU) subsystem advances through quiescent states, grace periods, and callback executions based on CPU scheduling, interrupt activity, preemption status, and per-CPU state machines.

 doi:10.20944/preprints202603.0354.v1

RCU's correctness relies on system-wide coordination, but its actual transitions are not predictable or statically predefinable, as they depend on asynchronous interactions among CPUs and tasks.

The combined effect of the above mechanisms is that Linux operates under open-world semantics, characterized by:

Unbounded control-flow growth, as new kernel paths may be activated at any time depending on runtime stimuli

Non-enumerable system states, driven by hardware events, concurrency, memory pressure, and dynamic scheduling

Load-dependent behavior, where timing, ordering, and subsystem interactions vary with execution conditions

Semantic drift, as the kernel adapts to internal states and external events in ways that cannot be frozen for certification

Such properties make Linux's total behavior neither finitely representable nor suitable for closed-form formal analysis, directly conflicting with high-assurance verification requirements (e.g., DO-178C DAL A/B) that rely on a bounded, analyzable state space.

ARINC 653-based platforms adopt the opposite architectural philosophy.

At system-integration time, they:

- Fix all partition schedules,
- Allocate static memory regions,
- Define deterministic inter-partition communication,
- Prohibit dynamic creation of execution paths,

Constrain the separation kernel to a small, finite, and fully analyzable state machine.

Because the separation kernel's behavior is static, bounded, and non-evolving, its complete state space is finite and amenable to formal verification, enabling the level of assurance required by high-integrity avionics.

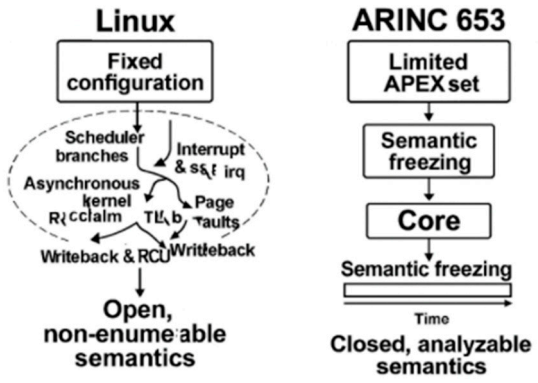

Figure 4. Semantic unfreezing.

F. 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);

 doi:10.20944/preprints202603.0354.v1

- interrupt-handling and softirq pathways, whose execution contexts are shared across all processes regardless of their criticality.

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, or one driver, 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 ARINC 653.

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;
- 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.

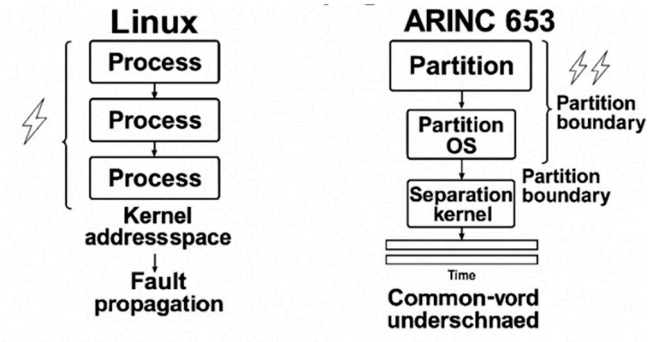

Figure 5. Fault propagation.

G. 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
- 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—

 doi:10.20944/preprints202603.0354.v1

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.

H. 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, devicetree 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.

In contrast, 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. This architectural discipline keeps DO-330 qualification tractable and prevents toolchain evolution from destabilizing certified baselines.

V. Common Misunderstandings and Why They Fail

A. Linux + RT Patches Provide Temporal Isolation

Anti-pattern: Assuming PREEMPT_RT implies certifiable temporal isolation
Why it's risky:

- Improves average latency but leaves asynchronous kernel activity and non-enumerable paths intact
- Cannot provide closed-form worst-case guarantees for ARINC 653 time windows

Checklist:

✓ Define WCET evidence for all critical threads without masking effects from interrupts/softirq/RCU
✓ Demonstrate absence of unbounded preemption points during the assigned time window

See Section IV A., E.

B. Open Source Reduces Cost

Anti-pattern: Equating "open-source = lower certification cost"
Why it's risky:

 doi:10.20944/preprints202603.0354.v1

- Certification scope scales with TCB size and evidence depth, not license fees
- Large, evolving kernel inflates verification and configuration control

Checklist:

✓ Bound the TCB and evidence items; provide stable, freeze-able baselines
✓ Quantify toolchain qualification scope (DO-330) with fixed versions

See Section IV D., H.

C. Cgroups Provide Partition–Equivalent Isolation

Anti-pattern: Treating cgroups as ARINC 653 partitions
Why it's risky:

- Resource shaping ≠ hardware-enforced isolation
- Shared kernel domain permits cross-component interference

Checklist:

✓ Show hardware-enforced memory exclusivity per partition
✓ Prove fault confinement boundaries with no cross-partition propagation

See Section IV A., B., F.

D. Using Mlock() and Disabling Swap

Anti-pattern: Relying on mlock()/no-swap for spatial isolation
Why it's risky:

- Residency ≠ fixed physical mapping; mappings can remain mutable
- Kernel activities (reclaim/compaction) can relocate pages

Checklist:

✓ Allocate static, non-overlapping physical regions; verify immutability at runtime
✓ Disable mechanisms that mutate page tables for DAL partitions

See Section IV B.

Even without swap, Linux continues to treat all physical memory as a single global pool. Any task may influence global memory pressure and thereby trigger reclaim operations that affect other processes. ARINC 653 requires each partition to have a dedicated, immutable physical memory region; disabling swap does nothing to enforce such boundaries.

2). mlock() locks pages in RAM but does not give exclusive ownership or fixed placement
mlock() prevents specific user-space pages from being paged out, but it does not:

- reserve a guaranteed amount of memory for an application,
- ensure that locked pages come from a specific physical memory range,
- prevent the kernel from compacting, migrating, or remapping those pages,
- isolate the locked pages from interference caused by other processes,
- protect the physical region with MMU hardware boundaries.
- Linux may still:
- migrate mlocked pages during compaction,
- modify their page-table entries,
- trigger TLB shootdowns,
- reclaim adjacent pages and cause cache/TLB side effects,
- allocate kernel memory into nearby physical ranges.

Thus, mlock() preserves residency, not isolation.

 doi:10.20944/preprints202603.0354.v1

E. Static Configuration Can Make Linux Deterministic

Linux's kernel execution paths are inherently dynamic and workload-dependent. Even under aggressive static configuration, the kernel continues to activate numerous internal subsystems—memory reclaim, RCU epoch progression, timer reprogramming, NUMA rebalancing, softirq dispatch, deferred workqueues, and asynchronous driver activity. Each of these mechanisms can introduce new execution paths at runtime, modify control-flow structure, or alter kernel state transitions in ways that depend on instantaneous system conditions rather than any static configuration.

Because these execution paths interact combinatorially—across scheduling state, memory pressure, interrupt timing, cache behavior, and driver activity—the resulting system semantics are open-world, non-finite, and non-enumerable. The full set of reachable kernel states cannot be exhaustively listed, bounded, or frozen at integration time. As a consequence, the behavior of a Linux-based system cannot be reduced to a closed, static operational model, and cannot be subjected to complete formal proof or exhaustive worst-case analysis.

This semantic openness implies that unexpected interactions between subsystems may occur at runtime, producing emergent behavior that is not derivable from a finite set of pre-analyzed states. Such unpredictability directly contradicts the foundational principles of avionics software assurance, which depend on a closed-world, deterministically analyzable execution model to establish verifiability, traceability, and credible safety arguments under DO-178C.

In contrast, ARINC 653 separation kernels are explicitly engineered to avoid semantic explosion: they freeze system behavior at integration time, prohibit dynamic creation of kernel execution paths, and constrain the trusted computing base to a small, finite state machine. This closed-world architecture enables the type of deterministic analysis, state-space bounding, and formal reasoning that high-assurance airborne systems require—but which Linux's dynamic semantics fundamentally preclude.

F. Abundant Linux Ecosystem

Although Linux's extensive ecosystem is often viewed as an engineering advantage, this richness provides little benefit in a certification context. As previously discussed, Linux already lacks strong spatial and temporal isolation due to its shared monolithic kernel. Beyond these architectural issues, the scale of the Linux ecosystem introduces a second, independent barrier: the broader the ecosystem, the larger the portion of the build and integration toolchain that must be addressed under DO-330 (or DO-33*) tool-qualification requirements.

Because Linux relies on a wide array of compilers, linkers, meta-build systems, configuration generators, device-tree compilers, package utilities, and scripting frameworks, each of these components becomes part of the certification scope when used to produce airborne software. A richer ecosystem necessarily expands:

- the number of tools involved in the build pipeline,
- the number of transformations applied to source artifacts,
- the number of scripts and generators that must be controlled or assessed, and
- the number of potential interactions requiring justification or qualification under DO-330.

Eventually, ecosystem breadth directly multiplies certification effort, even before considering version churn or kernel-level complexity. In contrast, certifiable ARINC 653 platforms deliberately employ minimal, stable, and highly controlled toolchains precisely to keep DO-330 obligations tractable.

VI. Recommendations

- Partitioning Kernels/Type-1 Hypervisors: XtratuM + LithOS, POK, JetOS
- Separation Kernels + User-Space ARINC Services: seL4 (MCS), Muen SK
- Commercial Certifiable Platforms: VxWorks 653, INTEGRITY-178B, LynxOS-178, PikeOS, DeOS

 doi:10.20944/preprints202603.0354.v1

VII. 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.

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. 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).
8. 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).
9. 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).
10. 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).
11. 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).
12. 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.
13. A. Crespo, P. Metge, & I. Lopez, *LithOS: A Guest OS for ARINC 653 on XtratuM Hypervisor*, Valencia, Spain: Univ. Politèc. Valencia, 2012.
14. 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.

 doi:10.20944/preprints202603.0354.v1

15. 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.
16. 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.
17. H. Härtig, A. Lackorzynski, & A. Warg, *The Muen Separation Kernel: Design and Formal Verification*, Dresden, Germany: Tech. Univ. Dresden, 2018.
18. J. Rushby, *Design and Verification of Secure Systems*, Menlo Park, CA, USA: SRI Int., 1981.
19. J. Rushby, "A kernelized architecture for safety-critical systems," in *Proc. IFIP Congr.*, Vienna, Austria, 1999, pp. 1–6.
20. Wind River Systems Inc., *VxWorks 653 Platform Datasheet*, [Online]. Available: https://www.windriver.com, 2022.
21. Green Hills Software Inc., *INTEGRITY-178B RTOS for Avionics*, [Online]. Available: https://www.ghs.com, 2021.
22. SYSGO AG, *PikeOS Safety-Certifiable RTOS and Hypervisor*, [Online]. Available: https://www.sysgo.com, 2024.
23. DDC-I Inc., *DeOS Safety-Critical RTOS*, [Online]. Available: https://www.ddci.com, 2024.
24. D. Bovet, & M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA, USA: O'Reilly Media, 2005.
25. R. Love, *Linux Kernel Development*, Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
26. M. Gorman, *Understanding the Linux Virtual Memory Manager*, Upper Saddle River, NJ, USA: Prentice Hall, 2004.
27. The Linux Kernel Organization, *Linux Scheduler Documentation*, [Online]. Available: https://docs.kernel.org/scheduler/, 2024.
28. The Linux Kernel Organization, *Linux Memory Management Documentation*, [Online]. Available: https://docs.kernel.org/mm/, 2024.
29. T. Gleixner, *PREEMPT_RT Patch Overview and Design Philosophy*, San Francisco, CA, USA: Linux Foundation, 2019, [Online]. Available: https://wiki.linuxfoundation.org/realtime/start.
30. The Linux Kernel Organization, *kbuild: The Linux Kernel Build System*, [Online]. Available: https://docs.kernel.org/kbuild/, 2024.
31. The Yocto Project, *Yocto Project Mega-Manual*, [Online]. Available: https://www.yoctoproject.org, 2024.
32. Device Tree Working Group, *Device Tree Specification*, [Online]. Available: https://www.devicetree.org, 2024.
33. 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.
34. a653rs Contributors, *a653rs-linux: ARINC 653 Emulation on Linux*, [Online]. Available: https://github.com/a653rs, 2024.