System-Level Safety and Certification Implications of Linux in Airborne Avionics

Haoran Lu *

Posted Date: 20 April 2026

doi: 10.20944/preprints202603.0354.v7

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

 doi:10.20944/preprints202603.0354.v7

Article

System-Level Safety and Certification Implications of Linux in Airborne Avionics

Haoran Lu

Symthosm (XianDao ZhiJue), Shanghai, China; 37183985@qq.com

Abstract

This paper presents a certification-oriented, system-level analysis of using Linux in safety-critical airborne avionics, with emphasis on Design Assurance Level (DAL) A/B systems. Linux is a feature-rich general-purpose operating system (OS) whose open and dynamic system semantics can be difficult to finitely bound and operationally "freeze" at integration time. We analyze how key architectural characteristics of Linux—including a large trusted computing base (TCB), asynchronous kernel activity, mutable memory mappings, monolithic privilege domains, and a rapidly evolving toolchain—interact with assurance objectives commonly expected under DO-178C and DO-330. The analysis identifies eight independently sufficient certification-relevant risk factors affecting temporal determinism, spatial isolation, fault containment, configuration stability, and lifecycle assurance feasibility. To avoid fragmented observations, these factors are consolidated into a unified causal framework that traces certification challenges back to two consequence categories: airworthiness feasibility constraints and semantic complexity. The framework also evaluates commonly proposed mitigations (e.g., PREEMPT_RT, containers/cgroups, and static configuration) and explains why these measures may not address the underlying system-level issues. The contribution of this work is a structured argumentation framework that makes architectural implications explicit and supports operating-system selection in integrated modular avionics.

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

1. Introduction

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

This shift in industry demographics creates a structural divergence in system-architecture choices. Well-resourced organizations—especially those with prior experience in safety-critical projects—typically adopt mature, certifiable real-time operating systems (RTOSs) such as VxWorks 653 or PikeOS. These platforms are explicitly designed around partitioning principles and provide the deterministic behavior, analyzable system semantics, and long-term configuration stability required under RTCA DO-178C (Radio Technical Commission for Aeronautics: Software Considerations in Airborne Systems and Equipment Certification). Their high licensing and

 doi:10.20944/preprints202603.0354.v7

integration costs are acceptable for organizations whose safety processes, program governance, and airworthiness culture already align with traditional avionics expectations [3,21,23].

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

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

This paper addresses that misconception from a system-safety and certification perspective. By examining Linux through the lens of DO-178C/DO-330 lifecycle assurance objectives, we show that Linux's open-world system semantics, timing variability, mutable memory mappings, monolithic trusted computing base (TCB), and continuously evolving toolchain can be misaligned with the architectural assumptions typically used to achieve certifiable airborne platforms at DAL A/B. We present a certification-oriented, system-level analysis that links Linux's architectural properties to concrete assurance impacts and synthesizes them into a unified causal framework. The framework provides a structured basis for system-level certification-oriented reasoning about OS architectural choices.

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. In practice, certifiable avionics platforms are therefore expected to exhibit closed, finitely analyzable execution semantics at integration time—an expectation that general-purpose operating systems typically do not readily satisfy [1–3].

In parallel, the avionics community has developed separation kernels and partitioned real-time platforms as more certifiable alternatives to monolithic kernels, explicitly targeting deterministic scheduling, strong spatial isolation, and fault-containment properties aligned with ARINC 653.

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

More recently, industry initiatives such as the Enabling Linux in Safety Applications project (ELISA) have sought to define processes, tools, and artifacts supporting the evaluation of Linux-based systems in safety-related domains. ELISA collaborates with certification bodies and industrial stakeholders and maintains working groups for sectors including aerospace. Whereas ELISA explicitly states that it cannot produce a certifiable Linux kernel or guarantee compliance with aviation standards, it positions its outputs as exploratory guidance rather than certification-oriented evidence. Accordingly, while ELISA demonstrates increasing industrial interest in adapting Linux to

 doi:10.20944/preprints202603.0354.v7

safety-related use cases, it does not alter the fundamental architectural considerations underlying DO-178C and ARINC 653 compliance [36].

Based on this foundation, Section 7 summarizes OS/platform classes that better align with DAL A/B certification expectations.

3. Certification-Oriented System-Level Analysis Framework

This work proposes a certification-oriented, system-level analysis framework grounded in standards-based reasoning and architectural semantics rather than empirical benchmarking or prototype construction. The framework is intended to be transferable: it can be applied to other candidate operating-system architectures by mapping documented architectural properties to assurance objectives and identifying system-level risk factors for DAL A/B certification under DO-178C and DO-330.

The analysis proceeds in three structured and mutually reinforcing stages:

3.1. Characterization of Linux Architectural Semantics

The first stage systematically characterizes Linux's architectural semantics—its monolithic kernel structure, dynamic memory-management behavior, asynchronous subsystem interactions, and other properties described in the real-time Linux and kernel-execution literature [25,27,30].

3.2. Identification of Independently Sufficient Certification-Relevant Risk Factors

These architectural characteristics are systematically mapped against DAL A/B assurance objectives as commonly interpreted in certification practice. This standards-based mapping yields eight independently sufficient certification-relevant risk factors. Any one of these factors can, on its own, compromise key DO-178C/DO-330 objectives (e.g., determinism, isolation, fault containment, configuration stability, or lifecycle assurance feasibility). Considered together, the eight risk factors reveal the breadth of challenges that arise when attempting to use Linux as a certification baseline for high-integrity airborne systems.

3.3. Unified Causal Framework and Examination of Mitigation Narratives

To avoid treating these eight risk factors as isolated observations, the analysis consolidates them into a unified causal structure. This structure traces the risk factors back to a shared architectural root cause—Linux's open-world, general-purpose design—and two consequence categories: airworthiness feasibility constraints and system semantic complexity. To reduce ambiguity in interpretation, the paper also examines commonly proposed mitigation narratives (e.g., PREEMPT_RT, containers/cgroups, and static configuration) and explains, within the same causal structure, why these measures may not address the potential system-level issues.

4. Certification-Relevant Architectural Risk Factors of Linux in Safety-Critical Avionics

The certification challenges associated with Linux-based avionics can be traced to eight architectural risk factors, each of which can independently impair assurance objectives expected for DAL A/B airborne systems (e.g., under DO-178C/DO-330 practice). Each factor is analyzed below and contrasted with design expectations commonly adopted in avionics-grade partitioned platforms.

 doi:10.20944/preprints202603.0354.v7

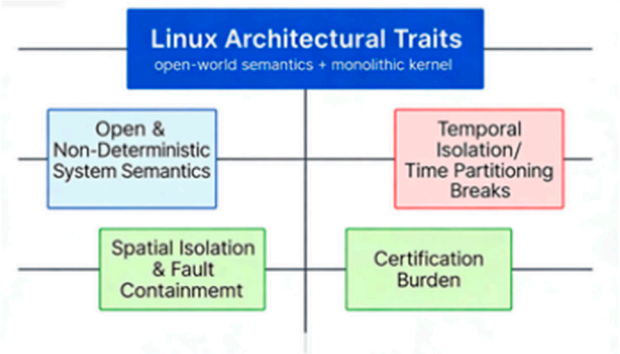

Figure 1. Classification of key Linux architectural traits.

4.1. Open Execution Semantics with Difficult-to-Bound Behavior

Linux follows an open-world execution model: at runtime, a large portion of kernel behavior is driven by asynchronous events (interrupts, deferred work), global resource conditions (memory pressure, I/O activity), and autonomous interactions among subsystems rather than by a closed, requirements-bounded schedule. The set of feasible kernel execution paths and interleavings is therefore difficult to enumerate at integration time, and the resulting integrated behavior is difficult to bound across operating conditions. The combined effect of these mechanisms is reduced predictability in both functional and timing dimensions, which makes it challenging to "freeze" a stable, reviewable certification baseline for DAL A/B systems.

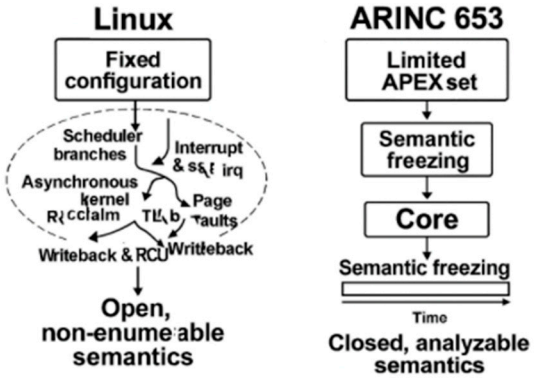

Figure 2. Semantic unfreezing in Linux vs semantic freezing in ARINC 653.

Sections 4.2 and 4.3 detail how this semantic openness translates into temporal nondeterminism and the absence of fixed physical-memory isolation.

4.2. Lack of Temporal Determinism

DO-178C 2.4.1(b) states that a partitioned software component should be allowed to consume shared processor resources only during its scheduled period of execution. In certification practice, this requirement is commonly addressed by architectures that provide a schedulable notion of execution windows (e.g., partition schedules) and, correspondingly, analyzable upper bounds on key scheduling latencies, interrupt/service overheads, and cross-component interference. ARINC 653-style platforms provide such execution windows via fixed, table-driven partition schedules, which bound CPU execution interference between partitions by construction. Linux's process/thread execution model does not readily provide comparable analyzable bounds because scheduling decisions, interrupt contexts, and background kernel work are driven by workload-dependent subsystem state and external events. Processes/threads are difficult to associate with statically provable determinism bounds across the full range of operating conditions [3].

 doi:10.20944/preprints202603.0354.v7

In Linux, temporal nondeterminism can be attributed to two architecturally distinct mechanisms whose impacts on execution timing should be considered separately.

4.2.1. Scheduler-Driven Nondeterminism

Priority-based scheduling in Linux does not necessarily 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 can occur even in the absence of external events. 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/thread until the critical section completes. The duration of these regions is runtime-dependent and influenced by cache state, lock contention, and microarchitectural factors, making their execution time difficult to bound analytically in a way that is robust across workloads.

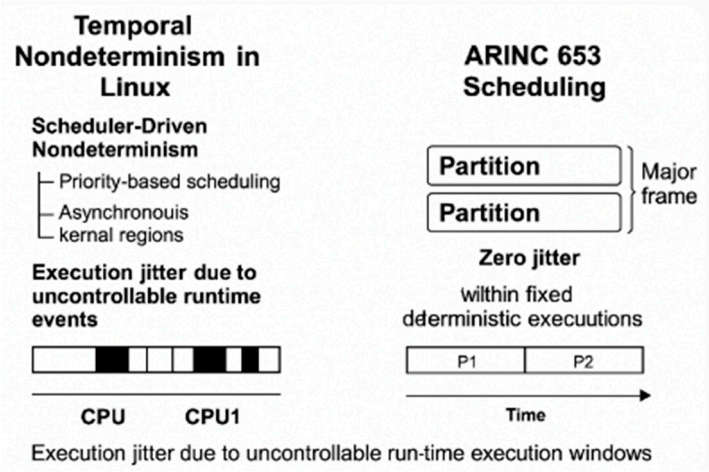

Figure 3. Scheduler-driven execution jitter in Linux vs deterministic ARINC 653 partition.

4.2.2. Event-Driven (Asynchronous) Nondeterminism

Separate from scheduler-driven effects, Linux also contains multiple asynchronous execution sources—including hardware interrupts, software interrupt request (softirq) processing, network-stack activity, timer callbacks and memory-reclaim operations. These activities are triggered by external stimuli or internal system conditions and may occur at arbitrary times. Depending on the execution context, they may either preempt a running task immediately (e.g., in interrupt/softirq contexts) or execute later as deferred work [i](e.g., workqueues or threaded interrupt handlers), still consuming CPU time and delaying time-critical tasks. Together, these effects introduce additional timing variability that is difficult to bound statically.

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

 doi:10.20944/preprints202603.0354.v7

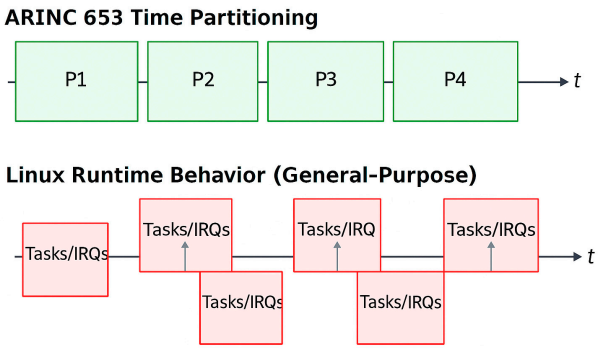

Figure 4. Event-driven execution jitter in Linux vs ARINC 653 fixed time partitions.

4.3. No Physical Memory Isolation

DO-178C Section 2.4 and 2.4.1 notes that partitioning may be achieved by allocating unique hardware resources to each software component (only one software component is executed on each hardware platform in a system). Alternatively, partitioning provisions may be made to allow multiple software components to run on the same hardware platform. Interpreting the first approach, placing components on different hardware platforms effectively creates distinct physical resource domains, eliminating shared-memory interaction by construction. For the second approach (co-resident components on the same platform), partitioning must still ensure that one component is not allowed to corrupt or modify another component's code, input/output (I/O), or data storage areas; in practice this requires demonstrable protection of memory and storage resources via hardware and/or a combination of hardware and software, as described in the standard. Linux relies on a dynamic and mutable virtual-memory architecture that can be difficult to reconcile with establishing fixed, partition-style physical memory regions. Several core mechanisms illustrate this behavior:

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

Because Linux performs these modifications autonomously, a process/thread is difficult to associate with a fixed, statically provable physical-memory region over all operating conditions. Linux does not typically provide a partition-style mechanism that prevents the kernel from reassigning or altering physical pages used by an application when global memory-management actions (e.g., reclaim, compaction, migration) are triggered. Similarly, Linux does not aim to provide the fixed, hardware-enforced memory ownership boundaries at the granularity assumed by ARINC 653-style partitions.

Linux's dynamic page-table management can be at odds with certification expectations that a partition's physical memory be statically allocated, hardware-isolated, and invariant throughout system operation.

 doi:10.20944/preprints202603.0354.v7

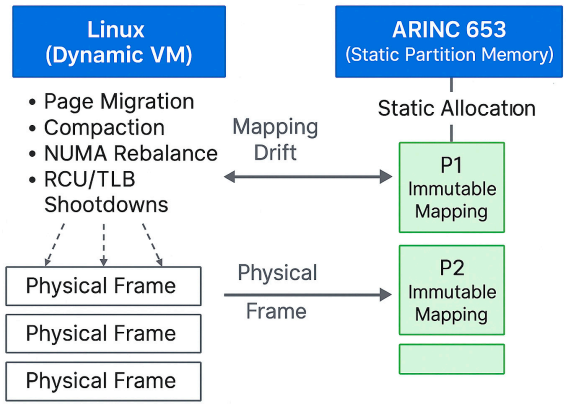

Figure 5. Linux memory mapping drift vs ARINC 653 fixed memory mapping.

4.4. Lack of Fault Isolation

Linux does not readily provide certification-grade inter-process fault containment. User processes ultimately depend on a single, shared, monolithic kernel that handles every system call. Because all processes share the same privileged kernel address space and kernel-resident global structures, a fault in one process may corrupt kernel state that is globally visible and 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, virtual file system (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, which execute in global kernel context and can therefore amplify system-wide impact once shared kernel state is corrupted.
- shared and dynamically managed physical-memory pools and page-table state, rather than fixed, non-overlapping, MMU-enforced partition memory regions;

Since these structures are not partition-scoped—and are not designed to be restricted or made private to individual processes—Linux may not prevent a fault originating in one process from affecting the execution correctness, timing, or stability of others under stressed or abnormal conditions. This propagation mechanism is inherent to monolithic-kernel design and can be difficult to align with the fault isolation expectations often applied to DAL A/B airborne software under DO-178C 2.4.1(c), including the need to avoid adverse effects across partitioned components.

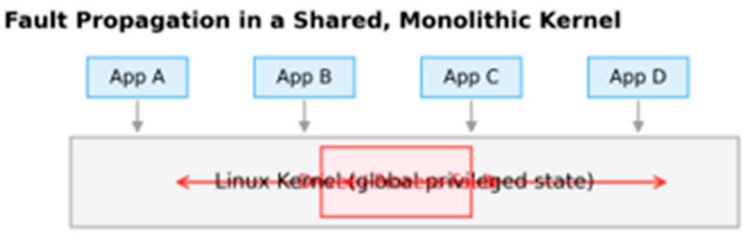

Figure 6. Fault propagation in Linux.

4.5. Driver Contamination of Kernel Global State

Linux device drivers execute with full kernel privilege, sharing the monolithic kernel's global address space. Eventually, a single defective driver can inadvertently corrupt system-wide shared state, including the shared kernel data structures and execution pathways discussed above. Unlike

 doi:10.20944/preprints202603.0354.v7

the previous section, where a partially faulty user component may only affect others indirectly by first corrupting kernel state through constrained interfaces, an in-kernel driver can directly write to global kernel memory and thereby directly compromise unrelated components across the system.

Because these structures are global, faults originating in one driver can propagate across unrelated components, undermining the system's ability to isolate erroneous behavior. This propagation pathway can be difficult to align with avionics fault-containment principles, where failures within one component are expected not to compromise the integrity or availability of others.

By contrast, avionics separation-kernel / ARINC 653-style RTOS baselines are designed to reduce the privileged-kernel failure domain and can place device drivers outside the privileged kernel address space; a broader baseline comparison is summarized in Section 4.8.

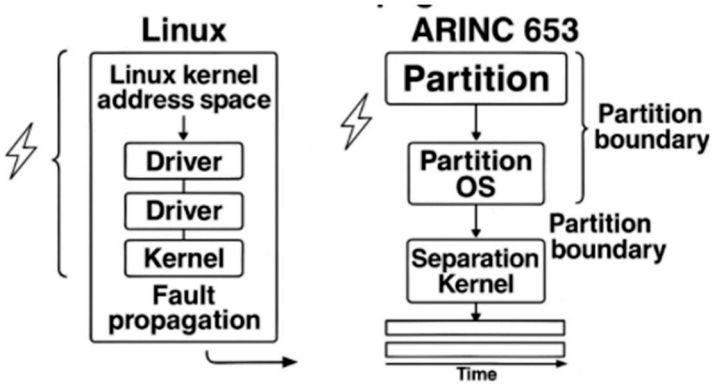

Figure 7. Linux monolithic kernel with drivers vs ARINC 653 separation kernel.

4.6. Overly Large Trusted Computing Base (TCB)

Linux integrates millions of lines of kernel code spanning diverse and continuously evolving subsystems, including device drivers, networking stacks, filesystems, IPC frameworks, tracing and debugging infrastructures, and numerous optional kernel features; all resident components form part of the Trusted Computing Base (TCB). Under DO-178C, every TCB-resident element contributes directly to certification scope and must undergo full lifecycle assurance activities. For a codebase of this scale and complexity, the resulting verification burden is often economically burdensome and operationally difficult to sustain for DAL A/B programs.

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

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

Certification begins with the software planning process and the Plan for Software Aspects of Certification (PSAC). Under DO-178C 4, planning must define the software life cycle and its inter-process relationships, select and define the software life cycle environment (methods and tools), and produce software plans/standards that are placed under change control and reviewed (DO-178C 4.1, 4.2, 4.3, 4.2(g), and 4.6). For DAL A/B, this implies that every software component contributing to safety objectives—and the means used to develop, verify, build, and load it—must be explicitly identified and governed by the plans. For Linux, this would require:

- Declaring all kernel subsystems, bundled drivers, and architecture-specific code paths that can affect safety objectives as part of the certifiable baseline, and assigning appropriate DAL driven objectives in the PSAC.
- Producing planning artifacts (PSAC; SDP, Software Development Plan; SVP, Software Verification Plan; SCMP, Software Configuration Management Plan; SQAP, Software Quality Assurance Plan) that define the software life cycle, transition criteria, and feedback mechanisms, and that also define the software life cycle environment (methods/tools) used to develop, verify,

 doi:10.20944/preprints202603.0354.v7

build, and load the kernel—placing these plans and standards under change control and review (DO-178C 4).

- Defining and maintaining an assurance strategy for kernel-wide concurrency, shared memory, interrupts, scheduling, and dynamic allocation, while also controlling the development environment (compiler/linker/loader versions and options). DO-178C notes that introducing new compiler/linker/loader versions or changing options may invalidate prior tests and coverage and requires planned re-verification means (DO-178C 4.4.2(c)).

Because Linux includes thousands of modules and hundreds of interdependent subsystems, planning artifacts would become unmanageably large, and many DAL A/B planning commitments (e.g., complete design traceability or determinism justification) are difficult to demonstrate in a comprehensive and stable manner.

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

DO-178C Section 5 defines a requirements-driven development chain in which high-level requirements (HLR) are developed from system inputs (DO-178C 5.1), refined into a software architecture and low-level requirements (LLR) suitable for direct coding (DO-178C 5.2), implemented as source code from those low-level requirements (DO-178C 5.3), and integrated into executable object code using controlled compiling/linking/loading data (DO-178C 5.4). DO-178C further requires bi-directional traceability across system requirements, HLR, LLR, and source code (DO-178C 5.5), so that completeness and absence of unintended functionality can be demonstrated.

Linux can be challenging to reconcile with this expectation:

- The kernel contains vast quantities of implementation-driven code, developed incrementally without DAL-style requirements decomposition,
- Core subsystems (scheduler, memory management (MM), VFS, network stack, timers, softirq) lack formalized HLR/LLR specifications, making DO-178C-style bi-directional traceability difficult to establish at the kernel scale.
- DO-178C code-level objectives typically require abnormal and exception paths to be explicitly defined and verified. Linux does not systematically enforce complete exception-path handling across kernel decision logic; achieving DAL A/B conformance would require substantial refactoring and added defensive code.

Bringing a Linux kernel baseline into DAL A/B development conformance would likely require reworking substantial portions of the kernel under DO-178C processes, which can reduce the practical advantages that often motivate adopting Linux.

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

DO-178C Section 6 defines verification as a technical assessment performed via a combination of reviews, analyses, and tests (DO-178C 6.0, 6.1, and 6.2). For a Linux-based OS baseline, these verification objectives typically include the following activities:

- Verification of the integrated Executable Object Code against intended requirements, including normal- and abnormal-condition behavior (DO-178C 6.1(d) and 6.1(e)).

DO-178C code-level objectives require abnormal and exception paths to be explicitly defined and verified. Linux does not systematically enforce complete exception-path handling across kernel decision logic; achieving DAL A/B conformance would require substantial refactoring and added defensive code.

- Requirements-based testing, including robustness (abnormal-range) test cases and procedures, executed in appropriate environments (DO-178C 6.4.1, 6.4.2, and 6.4.3, especially 6.4.2.2).
- Detailed review of integration outputs (compiling/linking/loading data and memory map) to confirm that the delivered executable configuration is consistent with the approved baseline (DO-178C 6.3.5).

 doi:10.20944/preprints202603.0354.v7

- Test-coverage analyses to confirm requirements-based coverage and structural coverage up to source-level modified condition/decision coverage (MC/DC) (as applicable) (DO-178C 6.4.4.1 and 6.4.4.2).
- Resolving any uncovered, dead/deactivated, or requirements-unjustified code and maintaining bidirectional verification traceability (requirements ↔ tests/procedures/results), to demonstrate the absence of unintended functionality (DO-178C 6.4.4.3, 6.5, and 6.1(d)).

Linux's scale and open-world, workload-driven kernel behavior make meeting these DO-178C Section 6 verification obligations at the kernel baseline level difficult in practice:

- Reviews/analyses (DO-178C 6.3.4): establishing verifiability and correctness across the full kernel—especially under interrupts and asynchronous execution—requires bounding and reviewing a very large set of interacting subsystems and configurations.
- Requirements-based testing and robustness (DO-178C 6.4.1–6.4.3): many kernel behaviors (e.g., reclaim, workqueues, softirq, and RCU) are triggered by workload-dependent conditions and asynchronous events, making abnormal-range testing and deterministic bounding of timing behavior (including WCET) difficult to close across operating conditions.
- Integration outputs review (DO-178C 6.3.5): the delivered executable configuration depends on large build-time configuration spaces (modules, Kconfig, architecture variants) and many intermediate artifacts, complicating consistent baselining and repeatable regeneration of integration outputs.
- Coverage analyses and closure (DO-178C 6.4.4.1–6.4.4.2): achieving MC/DC at kernel scale implies analyzing and justifying tens of thousands of decision points. Many of these decisions occur in code with high cyclomatic complexity and in paths that are configuration- and platform-dependent, which further complicates coverage closure.
- Dead/deactivated/extraneous code and traceability (DO-178C 6.4.4.3, 6.5, 6.1(d)): to support many hardware targets, the kernel source tree contains large amounts of architecture- and platform-specific driver code that is often not built for a given target configuration. Together with rarely executed fallback/error paths and debug logic, this makes it harder to demonstrate that all delivered functionality is requirement-justified, to close coverage on the enabled configuration, and to keep bidirectional traceability complete and stable across configuration variants.

In many DAL A/B contexts, the verification burden associated with bringing a monolithic Linux kernel into full DO-178C conformance can exceed typical certification cost and schedule envelopes.

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

For DAL A/B, DO-178C's Software Configuration Management (SCM) expectations require a defined and controlled software configuration, the ability to reproduce the Executable Object Code, disciplined baseline establishment, and traceable change control (e.g., DO-178C 7.2.2, 7.2.4, and 7.2.7). The Linux-specific concern is that continuous upstream evolution makes baseline stability difficult and amplifies re-verification and evidence maintenance. This lifecycle mechanism is discussed in Section 4.8.

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

Under DO-178C Section 8, the SQA process provides confidence that the software life cycle processes and their outputs conform to the approved plans and standards, that deficiencies are detected and tracked to resolution, and that required transition criteria are satisfied (DO-178C 8.1 and 8.2). For software submitted for certification, SQA also includes a conformity review to obtain assurance that life cycle processes and data are complete and that the delivered executable configuration is controlled and can be regenerated from archived data (DO-178C 8.3).

Linux makes these SQA expectations difficult to satisfy in a certification-grade manner because:

 doi:10.20944/preprints202603.0354.v7

- Upstream kernel development is performed by a large, distributed community, so an applicant cannot realistically audit the full set of life cycle processes for plan/standard compliance or systematically manage approved deviations at the scale envisioned by SQA audits (DO-178C 8.2(d)), nor can it treat upstream change sources as controlled "suppliers" in the DO-178C sense without establishing a separate, project-owned governance and audit regime (DO-178C 8.2(i)).
- The kernel's enormous TCB makes effective independent verification and SQA oversight impractical: auditing process execution, tracking deviations and problem reports to closure, and performing a meaningful software conformity review that demonstrates completeness and regenerability of the delivered executable configuration (DO-178C 8.3) becomes prohibitively large in scope when the OS itself is part of the certified baseline.

By contrast, avionics separation-kernel architectures mitigate this auditability problem by minimizing the certified OS TCB and restricting what must be treated as life-cycle evidence: the certified kernel is kept intentionally small and static, so that SQA and conformity activities remain tractable and do not scale into an "audit explosion."

4.7. Complex Toolchain Imposes Prohibitive DO-330 Qualification Burden

Linux relies on a broad and layered build-and-integration toolchain (compilers, linkers, build/configuration systems, code generators, and scripting utilities). Compared with typical avionics RTOS baselines, this toolchain involves more transformation stages and more versioned intermediate artifacts between source code and the delivered executable.

DO-178C Section 12 clarifies that tool qualification is needed only when the use of a software tool eliminates, reduces, or automates required life-cycle processes without the tool output being verified as specified in DO-178C 6 (DO-178C 12.2.1). The associated tool-qualification objectives, activities, and Tool Qualification Level (TQL) determination are addressed by DO-330; the impact criteria include cases where a tool can insert errors into airborne software, or can fail to detect errors by automating verification activities whose output is used to justify reduced verification effort (DO-178C 12.2.2; DO-330).

Linux delivery images are often produced by many build and integration tools, not just a compiler: kbuild/Kconfig, compiler/linker suites (e.g., GCC/LLVM and binutils), device-tree tools, rootfs/packaging systems (e.g., Yocto/Buildroot), plus numerous scripts. In DAL A/B programs, these tool outputs directly feed later verification and analysis activities (e.g., integrated Executable Object Code verification, object-code checks, WCET, and coverage closure). Accordingly, the applicant must decide which specific tool functions will be trusted for "certification credit"—i.e., where a tool output is accepted without equivalent independent verification to replace or reduce otherwise required reviews, analyses, tests, or coverage. This can involve not only build/integration tooling but also verification tooling (e.g., WCET-analysis tools and test/coverage automation such as Cantata, unit-test harnesses, and mocking/stubbing frameworks). If a tool is trusted in that way, its relevant functions may need DO-330 qualification (DO-178C Section 12); if not, the applicant must independently verify the tool's affected outputs (e.g., extra reviews, tests, or object-code checks). Either path increases the work needed to control tool versions, intermediate artifacts, and the resulting certification baseline.

By contrast, avionics-oriented RTOS programs typically keep the build-and-release tool environment tightly bounded and stable (often via vendor-controlled toolchains and certification support packages), which reduces the number of tool functions and intermediate artifacts that must be justified for DAL A/B and simplifies establishing repeatable build-and-load data as part of the certification baseline.

 doi:10.20944/preprints202603.0354.v7

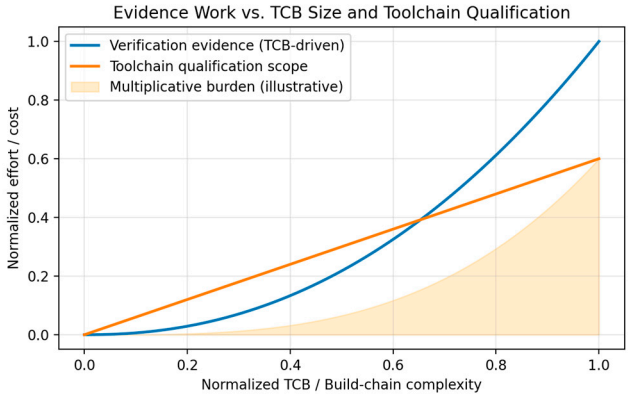

Figure 9. TCB and toolchain: the certification cost curve.

In addition, DO-178C expects compiling/linking/loading data to be controlled and repeatable as part of the certification baseline (DO-178C 5.4 and 6.3.5). This expectation can be more difficult to operationalize in Linux-style deployment stacks that include dynamically linked user space and runtime loader configuration: library composition and symbol resolution may vary across deployments unless the full image, loader settings, and dependency set are explicitly frozen and verified. As a result, the boundary of "certification-relevant build and load data" may extend beyond the kernel build itself and further increase lifecycle evidence and change-impact scope.

Overall, Linux's scale and ecosystem expand both the build/integration tool surface (and its intermediate artifacts) and the set of verification-related tool functions that must be justified. In DAL A/B programs, such tool functions must be either qualified under DO-330 when their outputs are used to replace or reduce required life-cycle activities, or otherwise backed by independent verification of the affected outputs—either of which increases assurance effort relative to a more tightly bounded OS/tool baseline.

4.8. Continuous Patch Stream Destabilizes Certified Baselines

Linux evolves at a rapid pace, and maintaining a stable, certifiable baseline can be difficult to reconcile with its development model.

Kernel updates are frequent because Linux must support a vast hardware ecosystem, address security vulnerabilities, and resolve behavioral regressions across numerous subsystems. Importantly, changes are not limited to peripheral drivers; they can modify core kernel semantics and internal interfaces that affect timing behavior, memory management, and fault-propagation pathways.

In certification terms, baseline changes—even small ones—may invalidate previously accepted evidence and trigger re-verification activities (impact analysis, regression testing, artifact updates, and re-establishment of traceability). When the baseline is forced to move continuously, programs can enter a re-verification spiral in which maintaining long-term, archived, and releasable configurations becomes increasingly difficult. The situation is particularly challenging for PREEMPT_RT, where upstream refinement (e.g., threaded interrupt behavior, sleeping spinlocks, priority inheritance details, and scheduler/locking interactions) can alter latency distributions and invalidate timing-analysis assumptions.

In certification terms, baseline changes—even small ones—may invalidate previously accepted evidence and trigger re-verification activities (impact analysis, regression testing, artifact updates, and re-establishment of traceability). When the baseline is forced to move continuously, programs can enter a re-verification spiral in which maintaining long-term, archived, and releasable configurations becomes increasingly difficult. This is particularly challenging for PREEMPT_RT, where upstream refinement (e.g., threaded interrupt behavior, sleeping spinlocks, priority

 doi:10.20944/preprints202603.0354.v7

inheritance details, and scheduler/locking interactions) can alter latency distributions and invalidate timing-analysis assumptions.

- 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 inter-process communication (IPC);
- complete disallowance of shared kernel-writable global state between partitions;
- static temporal partitioning with fixed, predefined time windows to guarantee CPU execution isolation between partitions;
- hierarchical health monitoring (HM) mechanisms supporting fault detection, containment, and recovery at process, partition, and system levels;
- fault-containment boundaries that ensure a failure inside one partition cannot corrupt the separation kernel or any other partition;
- partition placement of complex services (e.g., device drivers, networking stacks, and I/O services) outside the certified kernel, so that faults in these components do not directly mutate kernel-internal global state and can be contained by partition boundaries;
- a tightly controlled and auditable certification baseline (configuration identification, change records, and reproducible release configurations) to support long-term evidence maintenance under DO-178C DAL A/B;
- a bounded and stable build-and-load tool environment, with controlled compiling/linking/loading data, to reduce certification-relevant toolchain surface and simplify repeatability and review.

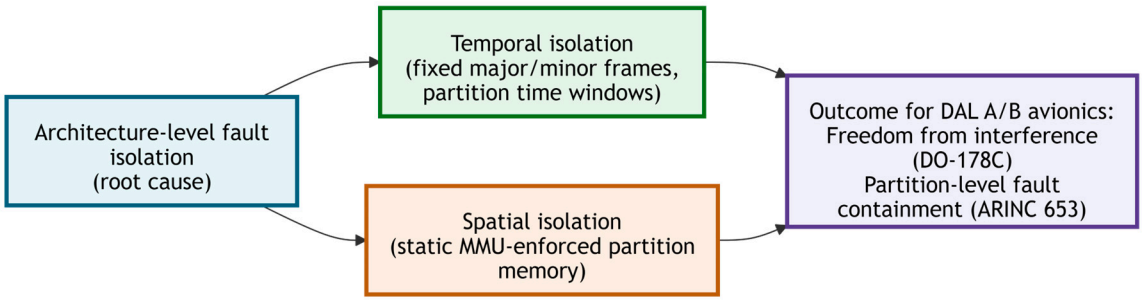

Figure 8. Relationship among architecture-level fault isolation and temporal/spatial isolation for DO-178C DAL A/B.

As such, ARINC 653 systems are designed to provide strong inter-partition fault isolation, with failures intended to be contained to the originating partition. This property is often central to supporting DO-178C-aligned fault-propagation arguments for DAL A/B functions.

5. Unified Causal Structure of Linux's Certification Challenges

This work is summarized through a unified causal chain that explains how Linux's general-purpose design choices can translate into certification-critical challenges in safety-critical avionics contexts.

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

5.1. Airworthiness Infeasibility

Airworthiness feasibility constraints lead to two additional certification-critical properties:

 doi:10.20944/preprints202603.0354.v7

5.1.1. An Excessively Large Trusted Computing Base (TCB) (Section 4.6)

5.1.2. A Highly Complex Toolchain That Drives DO-330 Qualification Burdens (Section 4.7)

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

5.1.3. Continuous Patch-Stream Evolution (Section 4.8)

5.2. Complex and Open System Semantics (Section 4.1)

The second primary consequence is driven by two major system-level challenges:

5.2.1. Temporal Non-Isolation – Arising from Asynchronous Kernel Activity, Dynamic Scheduling Behavior, and Non-Preemptible Regions (Section 4.2)

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

- Dynamic physical-memory ownership and page-table state (i.e., non-immutable logical-to-physical mappings), (Section 4.3)
- Driver-induced contamination of globally shared kernel state, (Section 4.5)
- Lack of enforced fault-containment boundaries. (Section 4.4)

Accordingly, the causal chain provides an integrated interpretation of the eight certification-relevant risk factors discussed in this paper, suggesting that they are closely related outcomes of Linux's foundational general-purpose design philosophy rather than isolated issues.

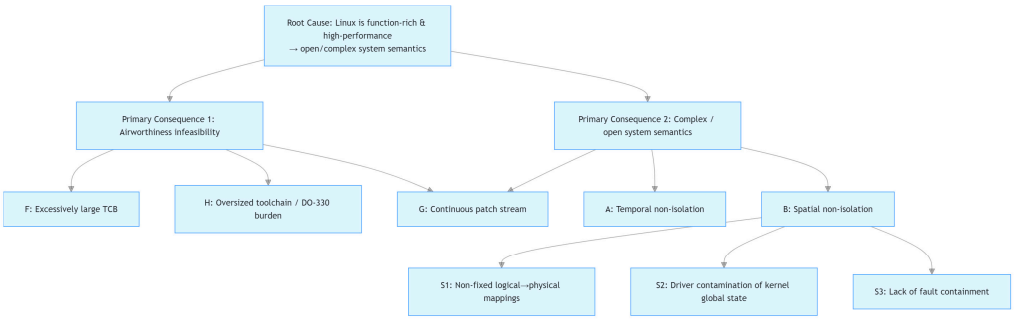

Figure 10. Two primary consequences and eight derived architectural risk factors forming the Linux certification risk structure.

6. Common Misunderstandings and Why They Fail

6.1. PREEMPT_RT = Determinism

6.1.1. Misconception

Applying PREEMPT_RT makes Linux sufficiently deterministic for DAL A/B airborne workloads and enables controlled timing interference and deterministic CPU-time allocation.

6.1.2. Reality

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

Significant interference sources may remain:

- interrupt request (IRQ) exit softirq cascades;

 doi:10.20944/preprints202603.0354.v7

- memory management events (page faults, reclaim, compaction)
- cross-CPU Translation Lookaside Buffer (TLB) shootdowns that can outrank RT tasks;
- driver execution time that depends on firmware/direct memory access (DMA) completion/lock contention;
- dynamic voltage and frequency scaling (DVFS) and CPU C-state exits that introduce microarchitectural stalls outside scheduler control;
- dynamic scheduler behavior (wakeup rules, load balancing, task migration, housekeeping threads) that produces runtime-dependent jitter.

These sources can delay dispatch/preemption for a runnable real-time task and can also steal CPU time while the task is executing, producing runtime-dependent timing interference.

6.1.3. Avionics Impact

For DAL A/B avionics, temporal determinism arguments require analyzable upper bounds on scheduling latency and interference (Section 4.2). PREEMPT_RT can reduce some sources of latency, but it does not by itself introduce ARINC 653-style fixed execution windows (temporal partitioning); establishing partition-level schedulability evidence under DO-178C therefore remains challenging.

6.2. Containers/Cgroups = Partitioning

6.2.1. Misconception

Containers/cgroups provide avionics-grade partitioning equivalent to ARINC 653.

6.2.2. Reality

Containers and cgroups 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, memory-management events, and driver paths are shared, faults and timing interference can propagate.

These mechanisms remain application-level resource controls layered on a shared monolithic kernel, so the system-level timing interference and fault-contagion concerns discussed in Sections 4.2–4.5 can still apply.

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

6.2.3. Avionics Impact

 doi:10.20944/preprints202603.0354.v7

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:

- Temporal isolation: cgroup CPU control provides proportional shares rather than fixed execution windows, so it does not resolve the analyzability and interference-bounding concerns discussed in Section 4.2.
- Spatial isolation: cgroups do not provide hardware-enforced, immutable physical-memory ownership; Linux's dynamic memory-management behavior (Section 4.3) therefore still applies.
- Fault isolation: containers/cgroups still share the same privileged kernel and driver domain, so fault-propagation concerns via shared kernel state remain (Sections 4.4–4.5).

Containers/cgroups on a Linux host do not, by themselves, provide the isolation properties typically expected for DO-178C DAL A/B partitioning claims.

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

6.3.1. Misconception

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

6.3.2. Reality

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

6.3.3. Avionics Impact

For DAL A/B certification under DO-178C, the platform is expected to provide robust, hardware-enforced partition isolation, which in practice depends on establishing stable, non-bypassable memory-protection boundaries throughout operation. Linux's dynamic memory subsystem makes establishing such static boundaries difficult (Section 4.3), and as such ARINC 653-style spatial isolation claims are difficult to substantiate for a Linux-based baseline.

6.4. Static Configuration Makes Linux Deterministic

6.4.1. Misconception

Fixing the process/thread set and key application-level parameters (e.g., stack sizes, memory budgets, and IPC topology) can effectively make Linux a closed, deterministic execution environment.

6.4.2. Reality

Static application-level configuration can reduce user space variability (e.g., by fixing the process/thread inventory, stack sizes, memory budgets, and IPC topology), but it does not change Linux's open-world system semantics. The kernel still performs autonomous, runtime-dependent activities driven by interrupts, device/driver behavior, memory pressure, and subsystem heuristics. Hence, new execution paths and interleavings continue to emerge during operation and cannot be completely enumerated or frozen at integration time (Section 4.1).

6.4.3. Avionics Impact

 doi:10.20944/preprints202603.0354.v7

Static configuration is insufficient to support DAL A/B partitioning and schedulability claims under DO-178C.

6.5. Abundant Linux Ecosystem

6.5.1. Misconception

Linux's extensive ecosystem and mature tooling should make it easier to build safety-critical airborne software.

6.5.2. Reality

The breadth of the Linux ecosystem increases engineering convenience but dramatically amplifies certification complexity. Linux depends on a large and heterogeneous collection of compilers, linkers, meta-build systems, configuration generators, device-tree compilers, packaging tools, scripting environments, and subsystem-specific utilities. Depending on intended use, some tool functions may require qualification per DO-330 (as triggered by DO-178C 12.2.1) when their outputs are used without equivalent verification to eliminate, reduce, or automate required life-cycle activities.

6.5.3. Avionics Impact

For DAL A/B avionics, "more tools and libraries" usually means more certification scope: more tool qualification exposure under DO-330. As analyzed in Section 4.8, ecosystem richness tends to increase assurance cost and lifecycle risk, even if it reduces prototype effort. In turn, the richness of the Linux ecosystem—an advantage for general engineering—becomes a certification burden under DO-178C/DO-330 and is misaligned with the lifecycle stability expected in airborne systems.

6.6. Open Source = Reduces Cost

6.6.1. Misconception

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

6.6.2. Reality

For DAL A/B programs, OS licensing fees are not the dominant cost driver; the primary driver is the scope and baseline stability of the DO-178C/DO-330 assurance and certification activities required over the lifecycle.

6.6.3. Avionics Impact

In practice, certification cost tends to scale with the scope and churn of the assurance activities that the applicant must perform and sustain throughout the program. Cost often grows with:

- Evidence volume across the full chain of requirements, design, code, verification, and coverage.
- Baseline volatility: frequent changes (e.g., patches, toolchain upgrades, configuration drift) expand impact analysis, regression testing, and evidence refresh scope.
- Independence and quality assurance activities (e.g., SQA, independent verification/validation as applicable), including reviews, compliance audits, and objective evidence management.
- Authority/assessor engagement overhead borne by the applicant: preparation and conduct of reviews/audits (e.g., SOI-1 to SOI-4—Stages of Involvement), responses and findings closure, plus supporting logistics such as secure facilities, tooling access, and on-site support.
- Supplier and version control across the lifecycle, including long-term reproducibility, auditability, and traceability of all build inputs, tools, and delivered configurations.

 doi:10.20944/preprints202603.0354.v7

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

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

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

Taken together, Sections 4–6 indicate that the certification-relevant risk factors identified in this paper are not readily eliminated by incremental hardening or configuration tightening. Mitigations such as PREEMPT_RT, containers/cgroups, and static application configuration can address specific symptoms, but they do not, by themselves, close the underlying gaps related to closed-world semantics, analyzable partition-level timing, and demonstrable spatial/fault isolation. If modifications become extensive enough to fully align the platform with avionics operating-system assumptions, the resulting system would typically resemble a heavily forked or re-architected platform with a substantially different assurance case and a renewed certification burden, rather than a commercial off-the-shelf (COTS) / upstream Linux baseline.

7. Recommended Avionics OS/Platform Classes

Based on the preceding analysis, we recommend OS/platform classes that better align with certification expectations for deterministic execution, isolation, and evidence stability in DAL A/B airborne systems.

7.1. ARINC 653 Partitioning Kernels

Examples include:

- RTEMS + AIR
- POK
- JetOS

These systems provide:

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

7.2. High-Assurance Separation Kernels with Userspace ARINC Services

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

- Muen SK

The architecture offers:

 doi:10.20944/preprints202603.0354.v7

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

7.3. 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

8. Conclusions

This work analyzed how Linux's architecture and lifecycle model interact with assurance objectives commonly applied to DAL A/B avionics. Architecturally, Linux retains open-world execution semantics (Section 4.1), exhibits timing variability that complicates analyzable temporal isolation (Section 4.2), and employs mutable memory-management mechanisms that challenge establishing immutable, hardware-enforced partition memory regions in the sense generally assumed by partitioned avionics platforms (Section 4.3). In addition, fault-containment boundaries are difficult to demonstrate at the system level when failures can propagate through shared kernel state and privileged driver execution (Sections 4.4 and 4.5). These technical properties, together with lifecycle factors such as baseline stability (Sections 4.6.4 and 4.8), development and traceability practices at scale (Section 4.6.2), tool qualification exposure (Section 4.7), and the feasibility of producing certification evidence across planning, verification, configuration management, and quality assurance (Sections 4.6.1–4.6.5), collectively indicate substantial certification risk for Linux-based approaches in flight-critical DAL A/B contexts.

Linux remains an effective platform for prototyping and offers practical advantages that are attractive to new entrants in emerging aviation domains. However, for flight-critical functions at DAL A/B, the analysis suggests that relying on Linux as the primary execution foundation can create assurance and lifecycle risks that are difficult to control within conventional DO-178C/DO-330 certification expectations. As low-altitude aviation continues to expand and attract cross-industry participants, OS and platform-architecture decisions benefit from early, explicit alignment with partitioning, determinism, isolation, and evidence-stability objectives so that long-term safety assurance is not sacrificed by short-term development convenience.

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.

 doi:10.20944/preprints202603.0354.v7

5. RTCA Inc., *DO-297: Integrated Modular Avionics (IMA) Design Guidance and Certification Considerations*, Washington, DC, USA: RTCA, 2005.
6. RTCA Inc., *DO-330: Software Tool Qualification Considerations*, Washington, DC, USA: RTCA, 2011.
7. SAE International, ARP4754A: Guidelines for Development of Civil Aircraft and Systems, Warrendale, PA, USA: SAE, 2010.
8. P. Wang, Q. Li, and H. Xiong, "Time and space partitioning technology for integrated modular avionics systems," *J. Beijing Univ. Aeronaut. Astronaut.*, vol. 38, no. 6, pp. 721–726, 2012 (in Chinese).
9. F. He, H. Xiong, and X. Zhou, "Overview of key technologies for ARINC 653 partitioned operating systems," *Acta Aeronaut. Astronaut. Sin.*, vol. 35, no. 7, pp. 1777–1796, 2014 (in Chinese).
10. Y. Li, T. Zhou, and J. Li, "Research and implementation of airborne ARINC 653 partition operating system," *Comput. Eng. Appl.*, vol. 51, no. 20, pp. 235–240, 2015 (in Chinese).
11. L. Chen, "Research on deterministic scheduling of avionics partition operating systems," Ph.D. dissertation, Coll. Aeronaut. Eng., Nanjing Univ. Aeronaut. Astronaut., Nanjing, China, 2018 (in Chinese).
12. R. Huang, "Research on ARINC 653 partition isolation mechanism for IMA," Ph.D. dissertation, Sch. Electr. Eng., Northwestern Polytech. Univ., Xi'an, China, 2020 (in Chinese).
13. I. Lopez, P. Parra, M. 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. A. Crespo, P. Metge, and I. Lopez, *LithOS: A Guest OS for ARINC 653 on XtratuM Hypervisor*, Valencia, Spain: Univ. Politèc. Valencia, 2012.
15. J. Delange, L. Pautet, and S. Faucou, "POK: an ARINC 653 compliant operating system for high-integrity systems," in *Reliable Software Technologies – Ada-Europe 2010*, Berlin, Germany: Springer, 2010, pp. 172–185.
16. B. Huber, A. Lackorzynski, A. Warg, et al., "seL4: formal verification of a high-assurance microkernel," *Commun. ACM*, vol. 57, no. 3, pp. 107–115, 2014.
17. I. Kuz, K. Elphinstone, G. Heiser, et al., "MCS: temporal isolation in the seL4 microkernel," in *Proc. 11th Oper. Syst. Platforms Embedded Real-Time Appl. (OSPERT)*, New York, NY, USA: IEEE, 2015, pp. 1–6.
18. H. Härtig, A. Lackorzynski, and A. Warg, *The Muen Separation Kernel: Design and Formal Verification*, Dresden, Germany: Tech. Univ. Dresden, 2018.
19. J. Rushby, *Design and Verification of Secure Systems*, Menlo Park, CA, USA: SRI Int., 1981.
20. J. Rushby, "A kernelized architecture for safety-critical systems," in *Proc. IFIP Congr.*, Vienna, Austria, 1999, pp. 1–6.
21. Wind River Systems Inc., *VxWorks 653 Platform Datasheet*. [Online]. Available: https://www.windriver.com. 2022.
22. Green Hills Software Inc., *INTEGRITY-178B RTOS for Avionics*. [Online]. Available: https://www.ghs.com. 2021.
23. SYSGO AG, *PikeOS Safety-Certifiable RTOS and Hypervisor*. [Online]. Available: https://www.sysgo.com. 2024.
24. DDC-I Inc., *DeOS Safety-Critical RTOS*. [Online]. Available: https://www.ddci.com. 2024.
25. D. Bovet and M. Cesati, *Understanding the Linux Kernel*, Sebastopol, CA, USA: O'Reilly Media, 2005.
26. R. Love, *Linux Kernel Development*, Upper Saddle River, NJ, USA: Addison-Wesley, 2010.
27. M. Gorman, *Understanding the Linux Virtual Memory Manager*, Upper Saddle River, NJ, USA: Prentice Hall, 2004.
28. The Linux Kernel Organization, *Linux Scheduler Documentation*. [Online]. Available: https://docs.kernel.org/scheduler/. 2024.
29. The Linux Kernel Organization, *Linux Memory Management Documentation*. [Online]. Available: https://docs.kernel.org/mm/. 2024.
30. T. Gleixner, *PREEMPT_RT Patch Overview and Design Philosophy*. San Francisco, CA, USA: Linux Foundation, 2019. [Online]. Available: https://wiki.linuxfoundation.org/realtime/start.
31. The Linux Kernel Organization, *kbuild: The Linux Kernel Build System*. [Online]. Available: https://docs.kernel.org/kbuild/. 2024.
32. The Yocto Project, *Yocto Project Mega-Manual*. [Online]. Available: https://www.yoctoproject.org. 2024.

 doi:10.20944/preprints202603.0354.v7

33. Device Tree Working Group, *Device Tree Specification*. [Online]. Available: https://www.devicetree.org. 2024.
34. H. Zhao, S. Gao, and Y. Yang, "Applicability analysis of airborne software based on Linux real-time extension," *Comput. Eng.*, vol. 43, no. S1, pp. 311–315, 2017.
35. a653rs Contributors, *a653rs-linux: ARINC 653 Emulation on Linux*. [Online]. Available: https://github.com/a653rs. 2024.
36. ELISA Project, *FAQs – ELISA: Enabling Linux in Safety Applications*. [Online]. Available: https://elisa.tech/about/faqs/.