The Hidden Risks of Using Linux in Aviation Systems

Haoran Lu *

Posted Date: 6 March 2026

doi: 10.20944/preprints202603.0354.v2

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

 doi:10.20944/preprints202603.0354.v2

Article

The Hidden Risks of Using Linux in Aviation Systems

Haoran Lu

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 process 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

In recent years, the aviation sector and emerging low-altitude mobility industries have increasingly adopted open-source operating systems such as Linux in airborne and vehicle-integrated platforms. This shift is largely driven by pressures for rapid development, ecosystem reuse, and the availability of extensive tooling. However, this trend also reflects a broader socio-technical phenomenon in which general-purpose software technologies are integrated into high-integrity systems without sufficiently aligning with established safety-engineering principles. As a result, technology adoption decisions risk being guided more by engineering convenience than by the requirements of safety-critical assurance.

A persistent misconception across the industry is the assumption that Linux can be elevated to avionics-grade reliability through incremental hardening or real-time extensions. Such assumptions overlook foundational mismatches between Linux's architectural properties—open system semantics, nondeterministic timing behavior, mutable memory mappings, and a large, continuously evolving trusted computing base—and the deterministic, partition-oriented safety models mandated by ARINC 653. These discrepancies undermine core safety mechanisms such as fault isolation, temporal partitioning, and the ability to establish verifiable upper bounds on system behavior, which are essential for high-integrity airborne systems.

More importantly, the use of general-purpose operating systems conflicts with the lifecycle assurance expectations defined in DO-178C and the tool-qualification obligations of DO-330. These standards require closed, analyzable system behavior, stable baselines, and strong evidence chains that can be independently reviewed and reproduced. When a complex, nondeterministic platform such as Linux is introduced into a certification context, safety assurance becomes both technically fragile and organizationally costly, increasing the likelihood of latent hazards emerging during operation.

This study addresses these concerns by framing Linux's architectural limitations not merely as technical constraints but as sources of systemic safety risk. Building on the universal functional-safety

 doi:10.20944/preprints202603.0354.v2

principles derived from IEC 61508, the analysis reveals how deviations from partition-based design, determinism, and verifiable execution semantics can propagate through the software lifecycle, affecting hazard control, safety governance, and certification credibility. By situating the evaluation within a broader socio-technical perspective, this work clarifies why the integration of general-purpose operating systems into airborne systems challenges the foundational assumptions of safety-critical system design and offers cross-domain insights for safety-driven technology selection and governance.

II. Why Linux Seems Attractive for Avionics

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

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 nature eliminates licensing fees and enables teams to reuse tooling, applications, and developer skills from adjacent industries. This results in faster prototyping and a lower barrier to entry for new avionics projects.

These advantages have led many engineering teams to incorrectly infer that Linux can be adapted to meet the requirements of safety-critical airborne systems. This misconception is pervasive across both Western and Asian aviation industries, with teams underestimating the fundamental architectural gaps between general-purpose operating systems (OSs) and avionics-grade safety-critical platforms.

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 for DAL A/B airborne systems. Each deficiency is analyzed below, with direct comparisons to avionics-grade platform design requirements.

 doi:10.20944/preprints202603.0354.v2

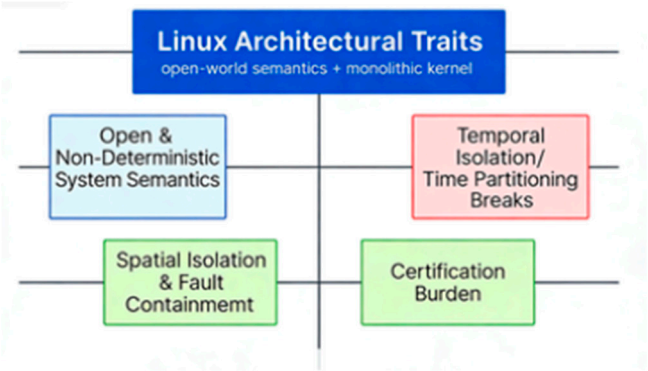


Figure 1.

A. Open System Semantics with Unpredictable Behavior

Linux exhibits inherent open-world system semantics, characterized by a vast, dynamically evolving set of execution paths whose behavior is shaped by runtime operating conditions, workload characteristics, and autonomous interactions among core kernel subsystems. These execution paths arise from multiple independent kernel mechanisms—including asynchronous kernel threads, memory reclaim and page table mutations, interrupt cascades and softIRQ execution, NUMA rebalancing, and RCU state transitions—that operate independently of application intent and cannot be enumerated, frozen, or formalized at system integration time. The combined effect of these mechanisms results in an unbounded, non-enumerable system state space, driven by hardware events, concurrent execution, dynamic memory pressure, and load-dependent scheduling, which defies closed-form formal analysis and static verification. This core design attribute stands in direct opposition to ARINC 653's architectural philosophy, which mandates closed, finitely analyzable system semantics at integration time: all partition schedules are fixed, memory regions are statically allocated, inter-partition communication is deterministic, and dynamic execution path creation is prohibited, constraining the separation kernel to a small, finite, and fully analyzable state machine. Linux's semantic openness is the root cause of its subsequent temporal and spatial isolation failures, as it inherently precludes the static, bounded, and fully verifiable behavior required for high-integrity DAL A/B avionics systems.

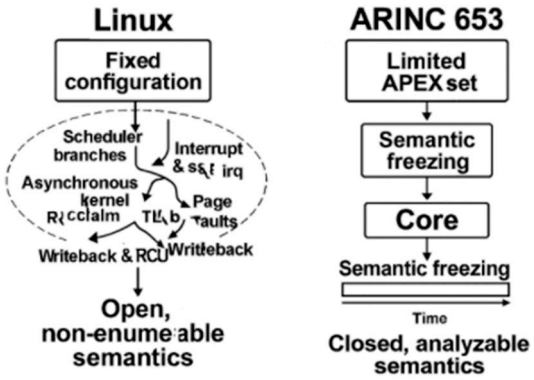


Figure 2. Semantic unfreezing.

B. Lack of Temporal Determinism

Deterministic execution requires that the system provide analyzable upper bounds on thread release latency, dispatch delay, kernel service time, and interference from other software components. The Linux process–thread execution model cannot provide such guarantees because scheduling decisions, interrupt handling, and kernel activity are driven by dynamically evolving

 doi:10.20944/preprints202603.0354.v2

workload, subsystem state, and asynchronous events. As a result, neither processes nor threads can be associated with a statically provable worst-case response time.

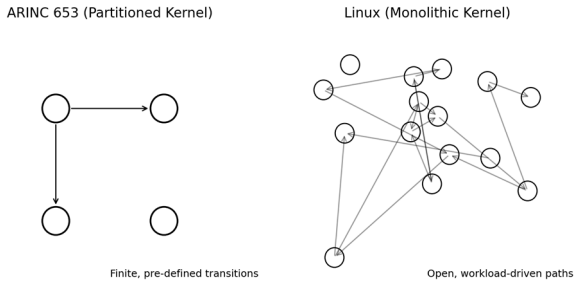


Figure 3. Execution Path comparison.

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.

1) Scheduler-Driven Nondeterminism

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

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

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

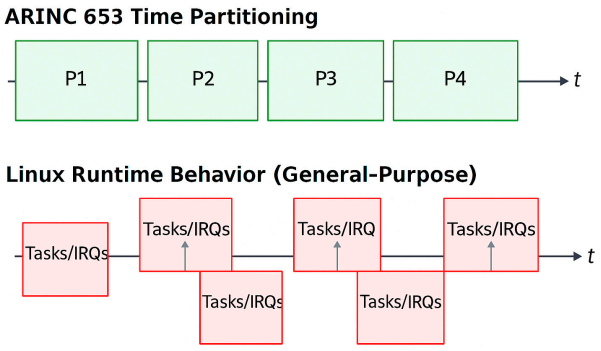


Figure 4.

2) Event-Driven (Asynchronous) Nondeterminism

 doi:10.20944/preprints202603.0354.v2

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 Fig. 3.

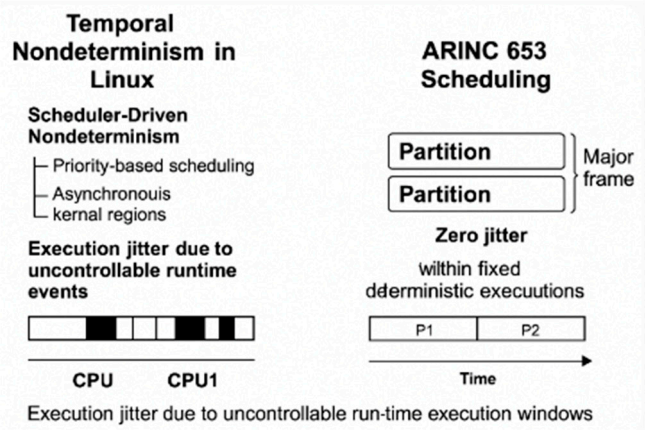


Figure 5. Scheduling determinism.

C. 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:

- 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.

 doi:10.20944/preprints202603.0354.v2

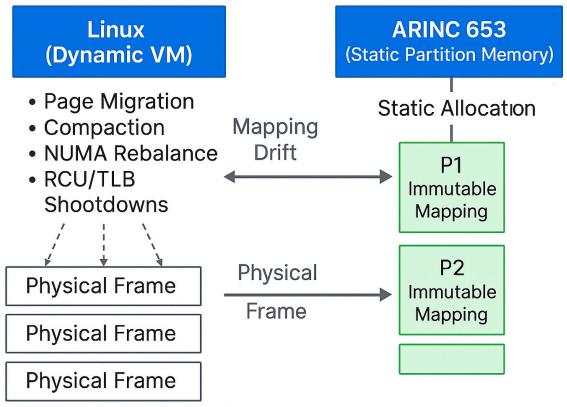


Figure 6.

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.

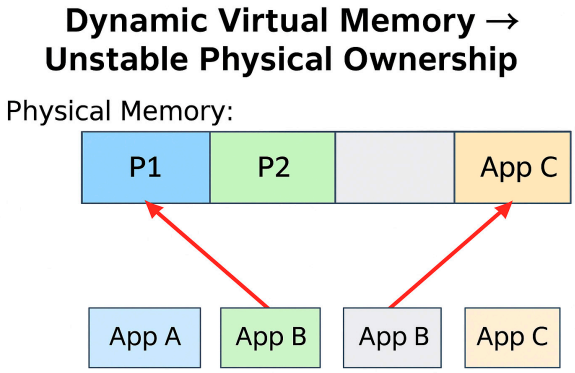
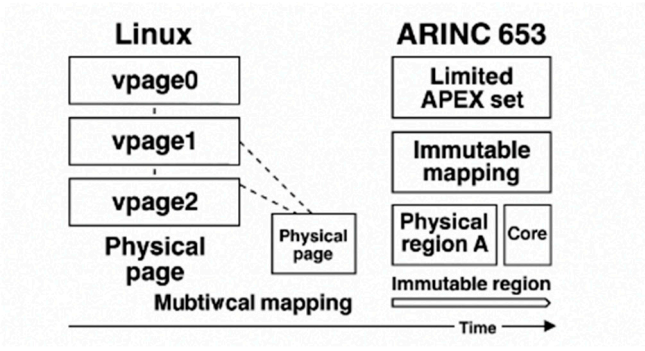


Figure 7.

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.



 doi:10.20944/preprints202603.0354.v2

Figure 8. Memory mapping drift.

D. Driver Contamination of Kernel Global State

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

scheduler queues, such as per-CPU run queues and scheduling structures, whose corruption impacts all tasks sharing the CPU;

memory-management metadata, including allocator structures and slab lists, leading to cross-subsystem memory corruption;

timing-critical kernel variables, such as jiffies or timer-wheel structures, whose misuse destabilizes system-wide timing behaviour.

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

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.

E. 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.

Figure 9.

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

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

 doi:10.20944/preprints202603.0354.v2

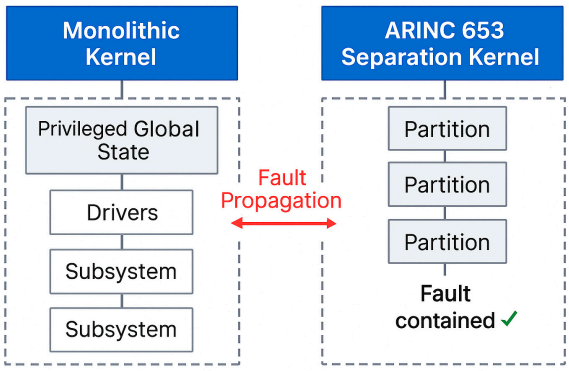


Figure 10.

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.

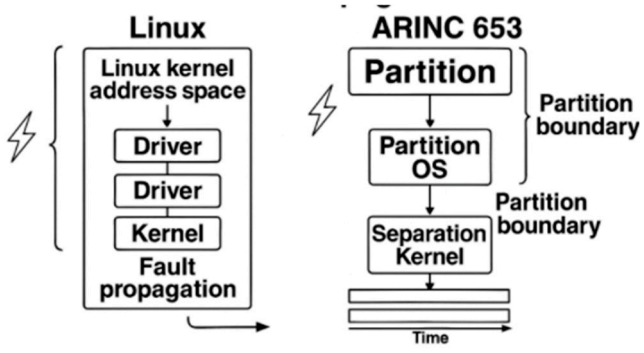


Figure 11. Kernel containing driver.

F. 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

 doi:10.20944/preprints202603.0354.v2

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.

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.

 doi:10.20944/preprints202603.0354.v2

- 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.
- 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.

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

 doi:10.20944/preprints202603.0354.v2

re-establishment of linkage between requirements, design data, and test results. A kernel that changes frequently cannot satisfy this expectation.

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

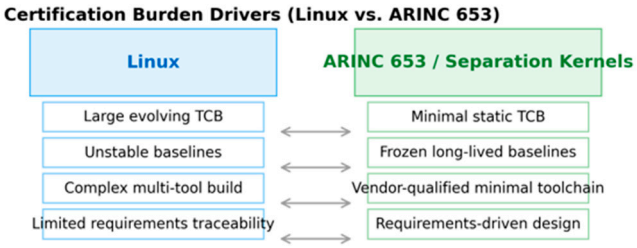


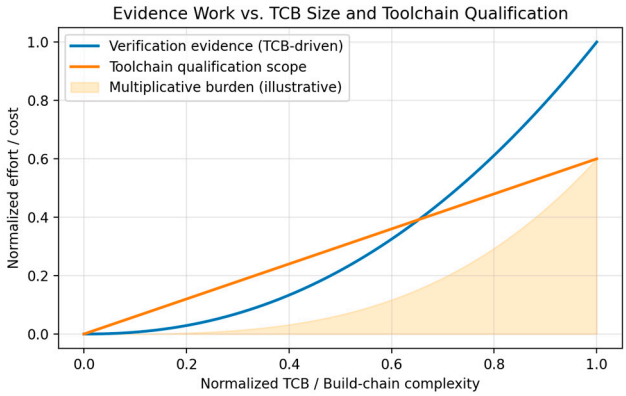
Figure 12.

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, Device Tree(DT) compilers, Python and shell-based code generators, pkg-config, ninja, and many others. Each participates in multiple stages of the build pipeline, producing intermediate files, scripts, headers, configuration databases, or hardware description blobs consumed by the kernel.



 doi:10.20944/preprints202603.0354.v2

Figure 13. TCB & Toolchain: The Certification Cost Curve.

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

Even when augmented with PREEMPT_RT, Linux cannot meet the fundamental temporal-isolation requirements defined by ARINC 653 for DAL A/B software. PREEMPT_RT improves average-case latency and reduces the duration of certain non-preemptible kernel sections, but it does not transform Linux into a system with a closed, statically analyzable timing model. Multiple execution paths within the kernel remain asynchronous, nondeterministic, and unbounded, preventing the establishment of the fixed worst-case temporal guarantees required for time-partitioned avionics systems.

1)First, Linux's interrupt architecture continues to allow unbounded softirq cascades, where the arrival of network, timer, or RCU events can trigger chains of deferred work. Even with interrupt threading enabled, these bottom-half workloads are not bounded in length or frequency and can execute at unpredictable times relative to application tasks. This violates ARINC 653's assumption that all kernel activities within a partition's time window are known and schedulable.

2)Second, the Linux memory subsystem introduces pervasive asynchronous memory events, including page faults, page reclaim, TLB shootdowns, compaction, NUMA migrations, and deferred freeing. These mechanisms are workload-dependent rather than statically defined. They may trigger at any time, consume unbounded CPU cycles, or block higher-priority threads, meaning that Linux cannot guarantee that a task will complete within its assigned time budget even under PREEMPT_RT.

3)Third, the Linux scheduler retains dynamic priority interactions, load balancing, CPU migration, wakeup-preemption behavior, and kernel thread activity. PREEMPT_RT does not eliminate the presence of kthreads responsible for RCU, workqueue flushing, per-CPU housekeeping, and various subsystem maintenance tasks. These kernel threads are not tied to a static ARINC-style major/minor frame and may preempt or delay user tasks at runtime.

4)Fourth, Linux device drivers remain a significant source of execution-time jitter. Drivers execute within the kernel's global scheduling domain, share global locks, and may introduce delays due to DMA completion handling, interrupt storms, or firmware communication. PREEMPT_RT threads certain interrupt handlers but cannot eliminate subsystem-level priority inversion, lock contention, or variable-duration driver execution.

5)Finally, modern Linux systems involve runtime power-management mechanisms, such as DVFS transitions and CPU C-state entry/exit latencies. These transitions have microsecond- to millisecond-scale variability and cannot be statically bounded. Because these mechanisms are not disabled in typical Linux deployments—and disabling them fully often breaks thermal or hardware requirements—the system's timing characteristics remain nondeterministic relative to ARINC 653 expectations.

Together, these characteristics produce an open-ended timing model in which execution paths depend on dynamic system state rather than a predetermined, certified time partition. ARINC 653 temporal isolation requires that each partition executes within a fixed, predefined time window, with all kernel activities analyzable and bounded. Linux, even with PREEMPT_RT, retains numerous

 doi:10.20944/preprints202603.0354.v2

execution paths that cannot be exhaustively bounded, scheduled, or certified, making it structurally incompatible with DAL A/B temporal-partitioning requirements.

B. Open Source Reduces Cost

Open-source platforms are optimized for broad functionality and rapid evolution—attributes that reduce development effort but provide no advantage in a DO-178 context. Certification costs are driven not by licensing, but by the amount of software that must be verified, traced, and analyzed. Every subsystem incorporated into the airborne TCB expands the verification envelope, regardless of whether the software itself is free. In Linux's case, its breadth, configurability, and constant evolution amplify verification workload far beyond any savings from zero licensing fees. As a result, open-source lowers development cost but sharply increases certification cost, especially at higher DALs where evidence requirements become stringent.

C. Cgroups Provide Partition–Equivalent Isolation

Linux control groups (cgroups) offer mechanisms for resource shaping, but they do not—and cannot—provide the architectural, temporal, or spatial guarantees required for ARINC 653-style safety-critical partitioning. Partitioning in ARINC 653 is a hardware-enforced isolation model, whereas cgroups operate entirely within a shared monolithic Linux kernel. As a result, cgroups cannot achieve deterministic execution, physical memory isolation, or fault containment.

The following sections explain why cgroups fundamentally lack the properties required for certified partitions.

1). Inadequate CPU Time Management

a) Cgroups control only long-term average CPU usage, not fixed execution windows

CPU quota and share controllers influence average CPU distribution over time but do not provide exclusive, deterministic execution windows.

b) Execution order is nondeterministic

Task dispatch in Linux remains governed by the general-purpose scheduler. Its behavior depends on:

- run-queue state
- wakeup timing
- load balancing
- kthread activity
- interrupt/softirq interference

This makes execution order nondeterministic and non-provable, violating DO-178C DAL A/B WCET requirements.

c) Tasks can be preempted or interrupted at any time

Even with cgroups, tasks may be paused by:

- interrupts
- softirq execution
- RCU callbacks
- memory reclaim
- kworkers
- driver activity

Thus, a cgroup cannot guarantee exclusive, interference-free execution, which ARINC 653 partitions explicitly guarantee within their time windows.

2). Inadequate Memory Space Management

a) Cgroups limit the amount of memory, but do not reserve a fixed memory allocation

Memory cgroups apply quota-based limits (e.g., memory.max) that constrain how much memory a group may consume.

However, they do not provide a statically reserved memory allotmen. A task in a cgroup:

 doi:10.20944/preprints202603.0354.v2

- is not guaranteed a specific minimum amount of available memory,
- does not receive memory pre-allocated at system initialization, and
- cannot rely on invariant memory availability during execution.

Thus, a cgroup's memory limit represents a ceiling, not a reservation.

b) Cgroups do not define any fixed or exclusive memory address ranges

Even when a cgroup's memory usage is limited, the kernel remains free to assign any virtual or physical page from the global memory pool. Cgroups do not define:

- specific virtual address regions,
- exclusive physical memory ranges, or
- hardware-protected, partition-specific address mappings.

Pages belonging to a cgroup may be:

- allocated from any NUMA node,
- reclaimed under pressure,
- compacted or migrated by the kernel,
- remapped due to page-table updates, or
- affected by global MM subsystem activity.

Therefore, cgroups provide only logical memory accounting, not physical memory isolation.

3). Shared Kernel Prevents Fault Containment and Interference Freedom

Even if cgroups isolate user-space processes logically, they do not isolate kernel-space activity, because:

all cgroups share the same kernel address space,

all system calls execute inside the same kernel instance,

all drivers run with full privilege inside shared global state,

interrupts and kernel threads serve all cgroups uniformly.

This means:

a driver fault can corrupt kernel global state,

a scheduling bug can stall all tasks,

memory pressure caused by one cgroup can trigger reclaim affecting others, RCU stalls or deadlocks propagate across the entire system.

4). Why ARINC 653 Partitions Can Achieve Strong Isolation

a) Strict temporal partitioning

ARINC 653 uses static, table-driven major/minor-frame scheduling.

Each partition receives: a deterministic, exclusive, pre-allocated time slice. No asynchronous kernel operations can preempt or delay a partition during its assigned window.

b) Strict spatial partitioning

ARINC 653 relies on:

static physical memory regions, enforced by hardware MMU/MPU, with immutable mappings during operation.

This guarantees zero cross-partition memory interference.

c) Hardware-level strong isolation

Both time and memory isolation are enforced using:

- MMU/MPU protections a minimal, static separation kernel
- no shared kernel-writable global state
- no driver execution inside the kernel
- fault containment boundaries

As a result, Misbehavior in one partition cannot contaminate others or the separation kernel, satisfying DAL A/B fault-containment requirements.

Table 1. Linux cgroups vs ARINC 653 partitions.

 doi:10.20944/preprints202603.0354.v2

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

D. Using Mlock() and Disabling Swap

1). Disabling swap does not provide physical memory isolation

Turning off swap merely prevents anonymous memory from being paged out to disk; it does **not** modify the semantics of:

- physical page allocation,
- page migration,
- reclaim under memory pressure,
- compaction,
- NUMA balancing,
- page-table updates,
- kernel-side memory sharing.

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.

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

 doi:10.20944/preprints202603.0354.v2

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.

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

 doi:10.20944/preprints202603.0354.v2

Misconception	Concise Refutation	See Section
PREEMPT_RT ⇒ determinism	Typical latency improves; certified time partitioning still unmet	§4.1
Open source reduces cost	Verification/qualification scales with complexity, not license	§4.4
cgroups = partitions	Quotas within shared kernel; no HW isolation/containment	§4.2–§4.3
mlock()/no-swap = isolation	Residency ≠ exclusivity; mappings still change	§4.2
Static config ⇒ analyzable	Core dynamic mechanisms remain essential	§3, §4.1–§4.3

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

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.

In safety-critical industries including avionics, rail transportation, medical devices, automotive electronics and elevators, all safety design philosophies share a common foundational source derived from the European functional safety standard IEC 61508. Although each sector has developed its own specific standards such as DO-178C for avionics, EN 50128 for rail, ISO 26262 for automotive and IEC 60601 for medical equipment, they follow highly consistent safety mechanisms: spatial partitioning, temporal isolation, deterministic behavior, and classification between vital and non-vital functions.

In rail transportation systems, typical designs adopt module-based memory partitioning to separate system memory into independent blocks for different modules. Meanwhile, cyclic, non-preemptive scheduling is widely used, where vital and non-vital tasks run in turn without preemption. Similarly, in avionics standards represented by ARINC 653 and DO-178C, such deterministic cyclic scheduling and memory partitioning are also strongly recommended to ensure system safety and timing predictability. This non-preemptive cyclic execution model is commonly adopted in both domains to avoid task interference and improve system stability.

 doi:10.20944/preprints202603.0354.v2

In medical software, even when using the singleton pattern, which typically relies on dynamic memory allocation, static memory is frequently employed instead to achieve deterministic execution and reduce runtime safety risks. This practice further demonstrates the consistent safety-driven design principles across all safety-critical sectors.

This paper analyzes the inherent defects of Linux from the bottom-level mechanisms including kernel scheduling, interrupt handling, memory management, MMU-based address mapping and isolation, and timing determinism. The conclusions comply with the unified safety principles inherited from IEC 61508 and can be universally applied to safety assessment, architecture selection and certification in avionics, rail transportation, medical devices and other high-safety industries. Therefore, extreme caution must be exercised when using Linux in other safety-critical software 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. 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.
8. R. Huang, “Research on ARINC 653 partition isolation mechanism for IMA,” Ph.D. dissertation, Sch. Electr. Eng., Northwestern Polytech. Univ., Xi’an, China, 2020.
9. 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, 2010, pp. 1–6.
10. 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.
11. 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.
12. 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.
13. H. Härtig, A. Lackorzynski, and A. Warg, The Muen Separation Kernel: Design and Formal Verification, Dresden, Germany: Tech. Univ. Dresden, 2018.
14. D. Bovet and M. Cesati, Understanding the Linux Kernel, Sebastopol, CA, USA: O’Reilly Media, 2005, pp. 1–986.
15. T. Gleixner, PREEMPT_RT Patch Overview and Design Philosophy, San Francisco, CA, USA: Linux Foundation, 2019 [Online]. Available: https://wiki.linuxfoundation.org/realtime/start (Accessed: Mar. 02, 2026).
16. Wind River Systems Inc., VxWorks 653 Platform Datasheet, 2022 [Online]. Available: https://www.windriver.com (Accessed: Mar. 02, 2026).
17. Green Hills Software Inc., INTEGRITY-178B RTOS for Avionics, 2021 [Online]. Available: https://www.ghs.com (Accessed: Mar. 02, 2026).
18. SYSGO AG, PikeOS Safety-Certifiable RTOS and Hypervisor, 2024 [Online]. Available: https://www.sysgo.com (Accessed: Mar. 02, 2026).

 doi:10.20944/preprints202603.0354.v2

19. DDC-I Inc., DeOS Safety-Critical RTOS, 2024 [Online]. Available: https://www.ddci.com (Accessed: Mar. 02, 2026).

Biography: Haoran Lu was born in Harbin, China, on January 6, 1981. He received the B.S. degree in Information and Computing Science from Heilongjiang University, Harbin, China, in 2004. Early in his career, he joined Honeywell as a Senior Software Engineer, where he contributed to white-box testing and verification activities for safety-critical onboard software deployed in the Airbus A380 program. His work focused on rigorous software assurance, structural coverage analysis, and defect isolation within high-integrity avionics environments. From 2017 to 2022, he worked at AVIAGE Systems, where he played a substantive role in the development, integration, and airworthiness certification of the C919 Integrated Modular Avionics (IMA) platform. In this capacity, he engaged in verification, validation, architecture compliance, and certification-evidence generation under DO-178C and ARINC 653–based processes, contributing to one of China's most significant commercial aircraft programs. In addition to his avionics experience, he has held senior engineering roles at Philips and Shanghai Electric Thales, working on high-safety-critical systems spanning medical diagnostic equipment and railway signaling platforms. These experiences strengthened his expertise in real-time embedded systems, system safety analysis, and the engineering of platforms requiring stringent development assurance. His research interests include safety-critical software engineering, avionics system architecture, real-time operating systems, partitioned and high-integrity runtime environments, and airworthiness certification methodologies.