Journal homepage: https://itce.vn.ua/en

Information Technologies and Computer Engineering **Vol. 22, No. 3. 2025**

Article's History: Received: 05.08.2025 Revised: 03.11.2025 Accepted: 23.12.2025

UDC 004.4:004.42 DOI: 10.31649/vitce/3.2025.65

Methodology for designing memory-safe high-performance applications using layered resource isolation

Olha Krasnozhon*
Master
Academician Stepan Demianchuk International University of Economics and Humanities
33000, 4S Demianchuk Str., Rivne, Ukraine
https://orcid.org/0009-0008-0202-9575

Abstract. This study presented a design strategy – Layered Resource Isolation – that reconciled memory safety with high performance by enforcing three explicit tiers of lifetimes and checks: an ephemeral tier for short-lived temporaries, a verifiable tier guarded by structural and aliasing validation at transfer points, and a persistent tier with audited release. The objective was to elevate lifetime boundaries to first-class design elements while avoiding vendor-specific frameworks. Neutral exemplars preserved identical algorithms across baseline and layered variants: a parser and compiler front-end that transforms token streams into abstract syntax trees, a multi-level cache with coherent read-through behaviour, and blocked numerical kernels. The evaluation instrumented allocations, promotions, audited releases, and phase timings, and used paired runs across thirty independent seeds to compare safety incidents per ten million operations, median runtime, ninety-fifth and ninety-ninth percentile latencies, throughput, and peak resident memory. Results showed elimination of leaks, double frees, use-after-free, and invalid frees within the detection horizon in all layered variants, with a one-sided confidence bound placing the incident rate below 0.11 per ten million operations. Tail behaviour improved markedly: ninety-fifth percentiles decreased by 21.8-24.9% and ninety-ninth percentiles by 22.8-27.6% across exemplars and load regimes, peak resident memory fell by 10-16%, steady-state throughput rose by 0.6-4.1%, and median runtime overhead remained near 1-2%. Practically, the approach reduced allocator contention, enabled whole-program reasoning about ownership and aliasing, and converted rare, expensive recovery into predictable boundary validation, offering a replicable methodology for advanced systems software

Keywords: audited release; ownership and alias control; validation checkpoints; allocator and handle contracts; alias guards; typestate encodings

Introduction

The tension between memory safety and high performance remains a central constraint in systems software. Manual allocation and deallocation provide precise control yet introduce error pathways that manifest as leaks, dangling references, double frees, and subtle lifetime violations. Tracing garbage collection reduces explicit ownership burden but can impose non-deterministic pauses, increase cache churn, and obscure the moment when resources are reclaimed. Deterministic destructor-based patterns (RAII – Resource Acquisition Is Initialisation) grant predictable clean-up but localise reasoning to scope boundaries that may not reflect aliasing realities across modules. Under realistic workloads, these mechanisms can degrade throughput, inflate tail latency, and compromise correctness. A design-level methodology that elevates memory safety to a first-class concern while preserving speed is therefore warranted.

Layered Resource Isolation (LRI) is proposed as such a methodology that is suitable for open academic use. The central premise holds that many safeties failures stem from blurred lifetime boundaries and ad hoc promotion of data from transient scopes to long-lived structures. To counter this, LRI partitions memory and related resources into three tiers with clear contracts and checkpoints: an ephemeral tier for short-lived temporaries and scratch buffers with strictly bounded scope; a verifiable tier for state that must pass structural, aliasing, and invariance

Suggested Citation:
Krasnozhon, O. (2025). Methodology for designing memory-safe high-performance applications using layered resource isolation. *Information Technologies and Computer Engineering*, 22(3), 65-76. doi: 10.31649/vitce/3.2025.65

*Corresponding author

checks before any promotion; and a persistent tier for long-lived objects with stable identity, explicit transfer semantics, and audited release. R.N. Watson *et al.* (2025) argued that software ecosystems need standardise principles and measurements for memory safety to curb entire classes of defects. The position paper outlined baseline practices and stresses cross-project coordination so improvements persist beyond individual languages.

This stratification complements – while remaining distinct from – prevailing practices. Manual management emphasises control but lacks uniform validation boundaries; tracing collectors amortises reclamation but relaxes explicit ownership; RAII enforces deterministic clean-up yet depends on local scope reasoning that can mask cross-component aliasing. Compatibility with C/C++ libraries and common toolchains is maintained to enable incremental adoption without vendor lock-in or domain-specific frameworks. A. Fromherz & J. Protzenko (2024) formalised a pathway for compiling C into safe Rust with machine-checked semantics. The work showed how verified transformations can retain performance while inheriting Rust's ownership and lifetime guarantees. V. Astrauskas *et al.* (2022) presented Prusti, a verification toolchain for Rust that enables specification and automated checking of program properties. The project strengthened Rust's safety model by proving invariants that exceed what the borrow checker enforces.

The methodological contribution is framed as strategy and methods rather than invention. First, a vocabulary for lifetime governance is established – entry and exit contracts for each tier, typestate or equivalent static encodings where available, and lightweight runtime guards where static proof is impractical. Second, neutral exemplars are used to illustrate application without domain coupling: parser and compiler front-ends where tokens become abstract syntax trees through verifiable promotion; cache layering where lines migrate only via validated coherence steps; and numerical kernels where working sets remain in the ephemeral tier and results are promoted under explicit invariants. S. Amar *et al.* (2023) introduced Capability Hardware Extension to RISC-V for Internet of Things (CHERIoT), bringing capability-based spatial and temporal memory safety to embedded devices. The evaluation demonstrates fine-grained compartmentalisation with minimal footprint, suggesting feasibility in constrained environments. A.E. Michael *et al.* (2023) proposed MSWasm (Memory-Safe WebAssembly), enforcing memory-safe execution of unsafe code via WebAssembly isolation. The runtime establishes sound boundaries for legacy components without demanding extensive rewrites.

S. Xu *et al.* (2024) developed Condo to harden container isolation by protecting kernel permission metadata. The approach reduces escalation vectors by safeguarding critical authorisation paths. Several gaps motivate this inquiry. Existing literature and practice provided powerful individual mechanisms – ownership types, region disciplines, hazard pointers, epoch reclamation, and advanced allocators – yet guidance on composing these elements into a language-agnostic, design-time methodology remains limited. Moreover, prior work often isolates formal safety guarantees from performance characterisation, whereas engineering practice demands co-optimisation. D. Greenspan *et al.* (2024) presented LOaPP (Low-Overhead Protection for Persistent memory), a low-overhead scheme that protects persistent memory objects at rest. Results indicated that practical security policies can coexist with high performance in PM-backed systems. H. Huang *et al.* (2024) introduced vKernel, which gives each container private code and data spaces to tighten intra-kernel separation. Experiments showed improved isolation and reduced cross-container interference with modest overhead.

The objective of the present study was to articulate LRI as a transparent methodology – including design rules, validation checkpoints, and cross-tier contracts – and to examine its implications for safety and performance using neutral software exemplars. The research problem is articulated as follows: how can a layered lifetime discipline reduce memory misuse while preserving high throughput and predictable latency in production-grade codebases? The working hypothesis states that explicit tier boundaries with mandatory validation checkpoints reduce both incidence and impact of misuse – and that the resulting locality and deterministic reclamation improve tail behaviour despite modest checking overheads. A secondary hypothesis anticipates that whole-programme reasoning is strengthened when cross-tier transfers are promoted to first-class operations governed by contracts.

Materials and Methods

Methodology: Layered resource isolation

The study operationalised a design methodology – Layered Resource Isolation – to reconcile memory safety with high performance. LRI partitioned memory and related handles into three tiers with bounded lifetimes and mandatory checkpoints. The ephemeral tier hosted temporaries and scratch buffers whose arenas were created and torn down at natural phase boundaries. The verifiable tier retained the intermediate state subject to structural, bounds, and aliasing checks; only objects that satisfied these contracts were eligible for promotion. Before promotion, the object in the verifiable tier undergoes a gate check: its structural integrity and bounds are validated, the absence of ephemeral and overlapping aliases is confirmed, typestate and single ownership are verified with no writers in flight and no epoch skew, then the object is sealed, assigned a stable identity, recorded in the ownership table, and logged for audited release. The persistent tier encapsulated long-lived objects with stable identity, explicit transfer semantics, and audited release. Cross-tier movement was governed by compile-time typestate encodings where available and by lightweight runtime guards otherwise. Allocator and handle contracts defined entry/exit conditions for each tier and prohibited references that crossed lifetime boundaries unchecked. The method remained language- and vendor-neutral and avoided domain-specific workflows; it was framed

as a strategy and set of methods rather than an invention, ensuring free academic use.

Neutral exemplars and implementation materials

To isolate lifetime discipline from algorithmic effects, three neutral software exemplars were implemented twice each: a baseline using idiomatic manual allocation and RAII/region patterns, and an LRI variant that preserved identical algorithms and dataflow while routing all allocations and transfers through tier contracts. The exemplars were (i) a parser and compiler front-end that transformed token streams into abstract syntax trees; (ii) a cache layering stack with coherent read-through behaviour across two and three levels; and (iii) numerical kernels drawn from blocked dense linear algebra and transform routines. These three exemplars were selected as representative because they collectively span orthogonal memory access and aliasing patterns (pointer-rich AST (Abstract Syntax Tree) graphs, coherent multi-level cache lines, and regular blocked arrays), remain domain-neutral and reproducible without vendor dependencies, provide clear checkpoints for promotion contracts, and, by keeping algorithms identical between the baseline and the LRI variant while changing only the tier wiring, allow observed effects to be attributed specifically to the layered architecture. All implementations targeted a portable POSIX (Portable Operating System Interface) environment and a standard C/C++ toolchain, with no vendor frameworks or device-specific APIs. Instrumentation in both variants recorded allocations, deallocations, and timing at phase boundaries; the LRI variant additionally logged promotion attempts, alias-guard triggers, and audited releases. Safety was examined by running sanitiser-assisted builds of the baseline to surface leaks, double frees, use-after-free, and invalid frees, and by examining LRI contract outcomes (e.g., rejected promotions) that would have manifested as defects without tiering. Safety was assessed using Clang/LLVM AddressSanitizer (ASan) with leak detection (LSan) enabled, supplemented by UndefinedBehaviorSanitizer (UBSan; bounds, null, vptr, and pointer-overflow checks); ThreadSanitizer (TSan) was applied to the cache exemplar to exclude race confounders, whereas MemorySanitizer (MSan) was not employed due to deliberate scratch-buffer initialisation patterns.

Measurement protocol and statistical analysis

Workloads were designed to stress allocation and lifetime behaviour, independent of domain semantics. Arrival processes covered near-normal traffic, heavy-tailed allocation sizes, and bursty phases that induced allocator contention. Each configuration (subsystem×regime×variant) was executed across 30 independent seeds to enable paired comparisons and variance estimation. Seed variation was controlled by a deterministic Pseudo-Random Number Generator (PRNG) at the harness level: for each run, one of 30 unique seeds parameterised all stochastic components of the workloads, including inter-arrival timings under the latency regimes, allocation sizes and lifetimes, burst positions and durations, parser input permutations, cache access orders and tiebreakers, and initial tiling choices in numerical kernels; build and environment settings were held constant to ensure independence and reproducibility. Metrics included median runtime (p50), tail latencies (p95, p99), steady-state throughput, peak resident memory, and incident rates per operation; in addition, dispersion statistics (interquartile range – IQR) and the p99/p50 ratio were computed as auxiliary indicators of latency spread and tail heaviness.

Throughput was computed as subsystem-specific operations per second over a steady-state window (warm-up and cool-down excluded) in a portable POSIX C/C++ harness: for the parser, tokens successfully parsed into AST nodes per second; for the cache stack, completed get and put operations per second with both hits and misses counted; and for the numerical kernels, completed blocked-kernel tile updates per second. Per-seed values were then summarised as LRI-versus-baseline relative deltas to normalise for hardware. Code size was also compared as an overhead proxy: "lines of core code" were counted per exemplar and per variant (baseline vs LRI), including only hand-written kernel sources and excluding tests, harness utilities, build configuration, third-party libraries, generated files, blank lines, and comments, and the metric was reported both as absolute counts and as LRI-to-baseline deltas. Relative deltas were computed as LRI versus baseline for each seed to control for run-to-run variability. Statistical analysis followed a paired, non-parametric plan suitable for skewed latency distributions: the Wilcoxon signed-rank test was applied to per-seed deltas, with Holm-Bonferroni adjustment for families of comparisons across regimes and subsystems ($\alpha = 0.05$). For throughput and peak memory, 10,000-replicate bias-corrected bootstrap confidence intervals were computed on paired deltas.

For safety with zero observed incidents in LRI, one-sided Clopper-Pearson bounds estimated the upper rate consistent with the observations. Deterministic seeding and fixed build configurations ensured reproducibility. This protocol aligned with the Results section by contrasting structurally identical baselines against LRI-governed variants, attributing any safety improvements and tail-latency reductions to explicit lifetime layering, validation checkpoints, and audited release rather than to algorithmic changes. An ablation study was also conducted by disabling individual mechanisms (typestate checks, alias guards, ephemeral pooling, and audited release) while holding all other factors constant. Additionally, sensitivity tests were conducted by varying checkpoint frequency, ephemeral-pool size, and the batching policy for audited release.

Results

Benchmark setup, datasets, and measurement protocols

The evaluation tested whether the proposed LRI methodology – built around ephemeral, verifiable, and persistent tiers with explicit entry/exit contracts and mandatory validation checkpoints – reduced memory safety defects while

maintaining or improving performance. To ensure that any differences arose from lifetime governance rather than algorithmic changes, each exemplar subsystem was implemented twice with identical algorithms and dataflow: a baseline version using idiomatic manual allocation and RAII/region discipline, and an LRI version that preserved the same logic but routed all allocations, handles, and promotions through tier contracts.

Three neutral, domain-agnostic exemplars were chosen because they stress different forms of resource management without invoking vendor workflows or device-specific contexts. First, a parser and compiler front-end transformed token streams into abstract syntax trees (ASTs). In the baseline, tokens, stacks, and intermediate nodes coexisted in a single allocation regime; in the LRI variant, short-lived tokens and stacks resided in ephemeral pools; partially built AST nodes and candidate symbols were placed in the verifiable tier and were promoted only after structural checks (balanced subtrees, resolved ownership of lexemes); and finalised AST and symbol tables moved to the persistent tier with audited release. Second, a cache layering stack implemented a coherent, read-through policy across two and three levels. The baseline relied on conventional invalidation paths; in the LRI version, allocations for request contexts and hash probes were ephemeral, candidate lines and decoded metadata remained verifiable until coherence and alias checks passed, and only then were stable lines persisted, and demotion and deallocation proceeded through audited release. Third, numerical kernels (blocked dense linear algebra and transform routines) used the same loop ordering and tile sizes in both implementations; the LRI variant confined tile-local scratch to ephemeral pools, kept partial results verifiable until dimensionality and bounds invariants were asserted, and promoted only consolidated outputs and reusable plans.

A common harness collected counts of allocations, deallocations, promotions, demotions, and audited releases; sanitiser-assisted incident detection (for the baseline) covering leaks, double-frees, use-after-free, and invalid free; LRI contract outcomes (accepted vs. rejected promotions, alias-guard triggers, audited release results); and timing at subsystem phase boundaries with p50/p95/p99 latencies and throughput under steady and bursty regimes. Workloads covered near-normal interarrival, heavy-tailed allocation-size distributions, and burst phases introducing allocator contention. Each configuration was repeated across 30 independent seeds per regime to bound variance and support paired statistical comparisons. In Table 1, there were the exemplar subsystems and their baseline/LRI variants, core lines of code, total allocation counts, promotion and audited-release volumes, and the workload regimes exercised.

Table 1. Workloads, scale, and instrumentation coverage

Subsystem (variant)	Lines of core code	Allocations (×10⁶)	Promotions (×10⁶)	Audited releases (×10⁶)	Regimes exercised
Parser/Front-end (baseline)	7,420	182.6	–	–	Near-normal, heavy-tailed, bursty
Parser/Front-end (LRI)	8,105	176.9	94.2	61.1	Near-normal, heavy-tailed, bursty
Cache layering (baseline)	6,033	139.4	–	–	Near-normal, heavy-tailed, bursty
Cache layering (LRI)	6,612	131.7	72.8	48.0	Near-normal, heavy-tailed, bursty
Numerical kernels (baseline)	5,487	214.9	–	–	Near-normal, heavy-tailed, bursty
Numerical kernels (LRI)	5,998	205.5	118.6	79.3	Near-normal, heavy-tailed, bursty

Source: created by the author

The table establishes parity of algorithmic scope while revealing structural differences introduced by LRI. Code size grew modestly (8-11%) because contracts and manifest-level tier wiring were added; however, allocation counts fell consistently in LRI (-3.1% in the parser, -5.5% in the cache, -4.4% in numerics). This reduction reflects ephemeral pooling and fewer accidental long-lived clones after disciplined promotion. Promotions and audited releases – nonexistent in the baseline – quantify verifiable persistent flows and accountable deallocations: the parser executed 94.2 million promotions and 61.1 million audited releases, indicating that not all verifiable objects required eventual persistence and that persistent objects were reclaimed under audit rather than ad hoc free paths. The cache and numerics show similar shaping of resource lifecycles (72.8/48.0 and 118.6/79.3 million promotions/audited releases, respectively). Crucially, the regimes exercised are identical across variants, enabling paired comparisons in subsequent tables. The setup therefore met the study's requirement: same algorithms, same workloads, and additional lifetime structure.

Safety outcomes across tiers and transfers

The primary question was whether explicit tier boundaries with validation checkpoints would reduce or eliminate common memory-safety defects. Baseline variants were compiled and run under sanitiser assistance to surface latent issues that might not crash immediately; LRI variants logged contract outcomes and audited every persistent-tier release. In Table 2, there were aggregated sanitiser-detected

incident rates per ten million operations – leaks, double-frees, use-after-free, and invalid frees – together with LRI-specific counts of rejected promotions and audited-release failures across regimes.

Table 2. Safety incidents per 10 million operations (mean across seeds)

Subsystem/Regime	Leaks	Double-frees	Use-after-free	Invalid free	Rejected promotions	Audited release failures
Parser (baseline, near-normal)	1.6	0.2	0.9	0.4	–	–
Parser (baseline, heavy-tailed)	2.3	0.3	1.8	0.6	–	–
Parser (baseline, bursty)	2.9	0.4	2.1	0.7	–	–
Parser (LRI, all regimes)	0.0	0.0	0.0	0.0	3.7	0.0
Cache (baseline, near-normal)	0.9	0.1	0.5	0.2	–	–
Cache (baseline, heavy-tailed)	1.5	0.2	1.1	0.4	–	–
Cache (baseline, bursty)	1.8	0.3	1.3	0.5	–	–
Cache (LRI, all regimes)	0.0	0.0	0.0	0.0	2.4	0.0
Numerics (baseline, near-normal)	0.7	0.0	0.3	0.1	–	–
Numerics (baseline, heavy-tailed)	1.1	0.1	0.8	0.3	–	–
Numerics (baseline, bursty)	1.4	0.1	1.0	0.4	–	–
Numerics (LRI, all regimes)	0.0	0.0	0.0	0.0	2.9	0.0

Source: created by the author

The baseline exhibited non-zero rates of every sanitiser-trackable defect class, with magnitudes increasing under heavy-tailed and bursty regimes. For instance, parser leaks rose from 1.6 to 2.9 per 10 million operations between near-normal and bursty regimes (+ 81.3%), while use-after-free nearly doubled (0.9→2.1, + 133.3%). Cache and numerics showed the same pattern, albeit with smaller absolute values in numerics owing to more regular lifetimes. In stark contrast, LRI variants recorded zero leaks, double-frees, use-after-free, and invalid frees across all regimes within the detection horizon. Rejected promotions were treated as contract outcomes rather than defects; they indicate verifiable-tier objects that did not satisfy structural or aliasing contracts and therefore were not persisted. The observed rates – 3.7, 2.4, and 2.9 per 10 million operations in parser, cache, and numeric – amount to approximately 0.0037-0.0024% of attempted promotions and correspond to conditions that, in the baseline, correlate with the sanitiser incidents seen in the adjacent rows (e.g., promoting an AST node holding a borrowed pointer to an ephemeral token buffer, or persisting a cache line with incoherent metadata).

No audited release failures occurred, demonstrating that persistent-tier deallocation happened only when all owning handles had been revoked or transferred per contract. A one-sided 95% confidence bound on the LRI incident rate with zero observed events placed the upper bound below 0.11 per 10 million operations, at least an order of magnitude under baseline means for the same subsystems and regimes. The directionality is consistent: when lifetime governance is explicit, entire defect classes disappear rather than merely decline. Mechanistically, the elimination follows directly from the tier semantics. Ephemeral pools are torn down wholesale at natural block boundaries, so "missed free on an error path" cannot accumulate into leaks; verifiable-tier promotion gates prevent any persistent object from holding pointers into ephemeral memory; alias guards catch stale handles before invalidation; audited release ensures that finalisation order is checked against ownership and alias maps, turning silent hazards into contract failures at the boundary rather than undefined behaviour deep in the program.

Latency, throughput, and peak memory footprint

The secondary question was whether LRI's contracts and audits, together with tier-aware allocations, preserved or improved performance metrics – especially tail latency – without imposing unacceptable median overhead. Timing probes were placed at phase boundaries in each subsystem, and results were summarised as relative deltas of the LRI variant against its baseline counterpart; negative deltas indicate reductions relative to baseline. In Table 3, there were relative deltas of the LRI implementation versus the baseline for median runtime, p95/p99 latency, throughput, and peak memory under near-normal, heavy-tailed, and bursty loads.

Table 3. Election and commit time percentiles (ms)

Subsystem/Regime	Median runtime Δ (p50)	p95 latency Δ	p99 latency Δ	Throughput Δ	Peak memory Δ
Parser (near-normal)	+1.3%	-22.4%	-24.1%	+0.9%	-10.7%
Parser (heavy-tailed)	+1.6%	-24.9%	-27.6%	+2.8%	-12.9%
Parser (bursty)	+1.7%	-23.1%	-26.3%	+3.2%	-13.8%
Cache (near-normal)	+1.2%	-21.8%	-23.2%	+1.1%	-10.1%
Cache (heavy-tailed)	+1.4%	-23.7%	-26.8%	+3.7%	-14.5%
Cache (bursty)	+1.5%	-22.6%	-25.4%	+4.1%	-15.9%

Table 3. Continued

Subsystem/Regime	Median runtime Δ (p50)	p95 latency Δ	p99 latency Δ	Throughput Δ	Peak memory Δ
Numerics (near-normal)	+1.1%	-21.9%	-22.8%	+0.6%	-10.3%
Numerics (heavy-tailed)	+1.4%	-23.4%	-25.1%	+2.4%	-11.6%
Numerics (bursty)	+1.6%	-22.7%	-24.9%	+2.9%	-12.1%

Source: created by the author

The median runtime overhead remained around the design target of ~1.5% (+1.1 to +1.7%), while tail latencies improved substantially: p95 decreased by 21.8-24.9% and p99 by 22.8-27.6% across all subsystems and regimes. Throughput modestly increased, with the largest gains under heavy-tailed and bursty conditions (+2.4 to +4.1%), exactly where the baseline is most vulnerable to allocator contention and long-path recoveries. Peak memory declined by ~10-16%, a direct consequence of ephemeral pooling (scope-wide teardown), disciplined promotion (fewer unnecessary clones), and audited release (preventing lingering retention). The p99 improvement is particularly instructive. In the parser, late discovery of malformed intermediate structures in the baseline triggered expensive recovery and piecemeal teardown, inflating the tail; in the LRI variant, invalid intermediate states failed fast in the verifiable tier and were discarded cheaply, keeping the slow path out of the hot loop. In the cache stack, explicit alias guards removed serialised read-modify-write episodes caused by stale handles, which under bursty writeback amplified into p99 spikes; with guards, those episodes were prevented or resolved early, flattening the upper tail (Hardin, 2023). In numerical kernels, ephemeral pooling improved locality by keeping temporaries in compact arenas with predictable lifetimes, reducing allocator traffic; promotion gates stopped partial tiles from polluting persistent structures and triggering compensatory passes, again cutting the tail.

Dispersion statistics (not shown in the table but computed from the same runs) reinforce the picture: the interquartile range of operation latencies shrank by 9-13% across all subsystems, while the p99/p50 ratio – an informal tail-heaviness index – declined by 21-25%. The observed median overhead reflected work performed by contract checks and audits; by shifting error handling from sporadic recovery to early, low-cost validation, less time was spent in slow paths. Heavy-tailed regimes showcase this effect: the LRI variant improved p95/p99 and converted a portion of the baseline's throughput variability into stable gains, as seen in +3.7% (cache) and +2.8% (parser) throughput under heavy-tailed loads. Peak memory reductions provide a second-order performance benefit. Smaller footprints improve cache residency of hot data (e.g., near-term AST nodes, cache metadata, and tile descriptors), which feeds back into latency stability. The -14.5% peak reduction in the cache under heavy-tailed regimes aligns with the strongest throughput improvement in that row (+3.7%), suggesting that lifetime discipline acts both as a correctness guard and as a soft capacity optimiser. Statistical checks on paired runs across 30 seeds indicate that p99 improvements remained significant after Holm-Bonferroni adjustment ($\alpha = 0.05$), and bootstrap confidence intervals for throughput and peak-memory deltas excluded zero for heavy-tailed and bursty regimes. Near-normal regimes showed smaller throughput gains (0.6-1.1%) with confidence intervals brushing zero in some seeds, which is expected when slow-path avoidance matters less.

Ablations, sensitivity, and threats to validity

The findings support the working hypothesis: explicit tier boundaries with mandatory validation checkpoints eliminated sanitiser-detectable memory violations and simultaneously improved tail latency (p95 -21.8% to -24.9%, p99 -22.8% to -27.6%) at a small median overhead of ~1-2% with a throughput uptick; moreover, the low rate of rejected promotions together with zero faults in auditable reclamation is consistent with the secondary hypothesis that promoting cross-tier transfers to first-class, contract-governed operations strengthens whole-program reasoning. To isolate which parts of LRI carry the observed benefits, an ablation study disabled one mechanism at a time within otherwise identical LRI implementations: No-Typestate (contracts evaluated only at runtime), No-Alias-Guards (no cross-tier alias checks), No-Pooled-Ephemeral (ephemeral allocations drawn from the general allocator), and No-Audited-Release (persistent deallocation without final audits). Results are expressed relative to full LRI; positive values in latency indicate worse performance than full LRI, while "safety incidents" report newly observed defects per 10 million operations. In Table 4, there were ablation results showing, for each removed mechanism, the additional safety incidents and the change in p99 latency, peak memory, and throughput relative to full LRI.

Table 4. Ablations: effect on key outcomes (relative to full LRI)

Ablation	Safety incidents (per 10M ops)	p99 latency Δ vs. LRI	Peak memory Δ vs. LRI	Throughput Δ vs. LRI
No-Typestate	+0.04 (verifiable failures surfaced late)	+3.1%	+0.8%	-0.5%
No-Alias-Guards	+0.09 (occasional stale-handle misuse)	+4.7%	+0.6%	-0.9%
No-Pooled-Ephemeral	+0.00	+2.6%	+4.3%	-1.4%
No-Audited-Release	+0.00 (no immediate errors)	+0.8%	+2.1%	-0.3%

Source: created by the author

Removing compile-time encodings (No-Typestate) preserved correctness through runtime contracts but shifted some checks later, increasing p99 by 3.1% relative to full LRI and introducing a small but measurable incident rate (+0.04 per 10 million operations) in code paths with deep verifiable lifetimes. Disabling alias guards (No-Alias-Guards) had the most pronounced safety impact among ablations: stale-handle misuse reappeared at +0.09 per 10 million operations, and p99 degraded by 4.7%, underscoring that cross-tier alias hygiene is a key determinant of tail stability. Eliminating ephemeral pooling (No-Pooled-Ephemeral) did not add incidents in the measured horizon but increased peak memory by 4.3% and reduced throughput by 1.4%, tying the footprint advantage to pooled lifetimes.

Removing audited release (No-Audited-Release) produced no immediate sanitiser findings in the experiment window, but peak memory rose by 2.1% and throughput slipped slightly, indicating latent risk and soft capacity loss as unreclaimed objects accumulate for longer. These ablations collectively attribute benefits to distinct components: typestate-like encodings primarily dampen tails by pulling checks forward; alias guards directly prevent the most insidious class of misuse (stale references crossing invalidation); ephemeral pooling drives footprint and allocator contention; and audited release enforces long-horizon hygiene that may not manifest as acute failures in short runs but guards against drift in persistent stores.

Two tuning dimensions were explored. First, checkpoint frequency in the verifiable tier varied from "on promotion only" to "on promotion and again after every N operations" for $N \in \{103,104\}$. The additional checks improved margins under contrived heavy-tailed bursts, reducing the probability that long-lived verifiable objects accumulated inconsistent state; the trade-off was a minor erosion of tail improvements (p99 benefits shrank by 1-2 percentage points) and < 0.3% added median overhead. For the studied exemplars, "on promotion only" sufficed, with higher frequencies recommended when verifiable objects span many epochs by design.

Second, tier granularity controlled ephemeral pool sizes and batching policy for audited releases. Pools $\leqslant 64$ KiB lowered peak memory slightly but increased allocator traffic and fragmentation, reducing throughput by ~0.8%. Pools $\geqslant 1$ MiB boosted numeric-kernel throughput by ~0.4% via better temporal locality but produced transient peaks during bursts. A 256-512 KiB range balanced these effects across subsystems. Audited releases performed best when aligned with natural quiescence points (phase boundaries); overly coarse batching risked short-lived over-retention and measurable p95 inflation. Although the exemplars were chosen for breadth – graph construction and sharing (parser), coherence and alias hygiene (cache), and regular temporaries and locality (numerics) – the study cannot claim universal coverage. Different languages and toolchains will vary in how easily typestate-like encodings are expressed; nevertheless, the core contracts and audits translate to any environment that can enforce entry/exit checks and ownership transfer semantics.

Experiments were long enough to capture steady state and bursts but did not simulate months-long uptimes; in practice, the value of audited release is expected to compound over extended horizons. Taken together, these observations delineate the boundary conditions of the approach: once lifetime governance is explicit, allocator choice becomes a second-order factor, and the principal effects persist across workloads. Sensitivity knobs (checkpoint frequency, ephemeral-pool sizing, and release-batching) trade minor overheads for predictable stability, while ablations attribute most safety and tail improvements to alias guards and early validation. Accordingly, tier contracts and audited release are treated as the primary deployment levers, with allocator selection and granular tuning reserved for workload-specific shaping. The next section synthesises these implications, relating the measured effects to locality, contention, and fault containment, and distilling practical guidance for adoption.

Discussion

The study demonstrated that elevating lifetime boundaries and transfers to first-class design elements – ephemeral arenas for short-lived temporaries, a verifiable tier gated by structural and aliasing checks, and a persistent tier with audited release – suppressed sanitiser-detectable memory-safety defects to zero across all exemplars while improving tail behaviour and shrinking peak footprint. The median overhead remained near one to two per cent, yet p95/p99 latencies declined by roughly a quarter, and peak memory fell by about one-tenth to one-sixth. Ablations indicated that early checks encoded via typestate primarily dampened the upper tail, cross-tier alias guards prevented the most damaging stale-handle paths, pooled ephemeral allocation reduced allocator contention and fragmentation, and audited release contributed to long-horizon hygiene. Taken together, these results supported the central claim that explicit lifetime governance relocates error handling from rare, expensive, slow paths to cheap, predictable boundary validations.

The observed reductions in tail latency were consistent with evidence that memory-hierarchy discipline and working-set control dominate long-path behaviour under contention. M. Vaithianathan (2025) surveyed cache – DRAM (Dynamic Random Access Memory) hierarchies for high-performance workloads and emphasised that balanced placement and NUMA (Non-Uniform Memory Access)-aware policies sustain throughput at scale; by tightening promotions and pruning accidental persistence, the methodology reduced remote traffic amplification that typically harms p99 under load. N.D. Clerigo & J. Teleron (2025) compared allocator and reclamation strategies across systems, noting that lifetime misfit and fragmentation inflate variance; the arena-based ephemeral tier and refusal to persist unverifiable objects matched that prescription and explained the measured shrinkage of

dispersion. J.R.C. Jalaman & J.I. Teleron (2024) examined paging, caching, and allocation under realistic workloads and showed how locality decisions interact with OS policy; the footprint reductions recorded here plausibly compounded those effects, particularly in heavy-tailed regimes where allocator pressure and paging sensitivity align.

Runtime interference and resource sharing are well-known drivers of unstable tails, and the results under bursty regimes aligned with that literature. J. Kim & K. Lee (2020) analysed I/O resource isolation in serverless runtimes for data-intensive functions and identified interference channels that destabilise throughput; by limiting transient growth and preventing cross-tier aliasing, the methodology reduced time spent in allocator and cache slow paths and thereby reduced sensitivity to such interference. R. Bazuku *et al.* (2023) outlined operating-system trends toward security-aware kernels and lightweight isolation; lifetime discipline at the application layer complemented these shifts by shrinking the volume of undefined behaviour presented to the kernel. R. Abuleil *et al.* (2023) proposed a composite security blueprint for virtualised environments and reported reduced cross-tenant risk; the boundary contracts reported here further narrowed intra-process fault surfaces that virtualisation alone does not address. K. Sharma & P. Khurana (2025) synthesised container hardening patterns centred on least privilege and continuous verification; the audited-release rule and promotion gates embodied a similar verification stance within the process, providing a natural seam for external policy to target. M. Waseem *et al.* (2025) underscored portability and policy consistency in multi-cloud containerisation; because the tiering rules were vendor-neutral and encoded at design time, they fit that portability requirement. A.Y. Wong *et al.* (2023) mapped the container threat landscape and emphasised orchestrator policy and supply-chain assurance; while these concerns are orthogonal to memory discipline, eliminating intra-process misuse closed many avenues that such threats exploit.

At service boundaries, security guidance stressed identity, policy enforcement, and observability. M. Kothapalli (2021) delineated microservices practices along precisely those axes; within that framing, promotion contracts acted as a policy enforcement point that refused unsafe object sharing across interfaces, and audited release created observable, testable lifecycle events rather than ad hoc frees dispersed through the codebase. The empirical disappearance of double frees and use-after-free in the present study indicated that this internal policy materially simplified external enforcement by reducing undefined behaviour at the call boundary.

Fine-grained compartmentalisation inside a process provided another lens to interpret the ablations. D. Adak *et al.* (2025) introduced SpecMPK (Memory Protection Keys) and showed that speculative permission updates reduced the switching overhead of in-process isolation; the upper-tail erosion observed when alias guards were removed mirrored their conclusion that making boundary checks cheap and frequent is essential to practicality. M. Untergug-genberger *et al.* (2024) presented TME-Box (Total Memory Encryption), which attached per-domain encryption to partition data within an address space at modest cost; the methodology here achieved similar separation effects at the software-contract level, with comparable overhead magnitudes. K.D. Duy *et al.* (2023) proposed Capacity, a capability system that enforces fine-grained memory and API access; the refusal to promote unverifiable objects and the prohibition of cross-tier dangling references mirrored capability checks and suggested that software contracts and hardware capabilities are additive rather than exclusive. B. Joseph & R. Kavitha (2025) present radiation-hardened SRAM (Static random-access memory) with in-memory computing to tolerate soft errors in safety-critical contexts; while orthogonal to software lifetime governance, this hardware approach complements LRI by letting verifiable-tier checkpoints and audited promotions confine and discard corrupted states before they reach the persistent tier, thereby limiting blast radius without compromising throughput.

Language design also informed the interpretation of safety and tail stability. W. Bugden & A. Alahmar (2022) empirically compared languages and concluded that memory and concurrency choices materially shift defect rates and runtime costs; the present methodology replicated a portion of those benefits without a language migration by encoding lifetime rules and ownership transfers explicitly. T. Weis *et al.* (2019) introduced Fyr as a systems language pursuing memory and thread safety by construction; that direction was complementary, and the current findings provided a bridge for teams that could not adopt a new language wholesale. N. Yoshimura *et al.* (2024) proposed TECS/Rust (TOPPERS Embedded Component Systems/Rust) to enable componentised memory-safe composition with static checks in constrained environments; the verifiable-to-persistent promotion pattern mirrored such component contracts and suggested that the methodology could be encoded natively where stronger types are available.

A. Partap & D. Boneh (2022) designed an efficient memory-tagging scheme to detect spatial and temporal errors; the explicit boundaries reported here created natural cut-points where tag violations would surface early, simplifying diagnosis and limiting propagation. M. Gross *et al.* (2021) showed that malicious FPGA-SoC (Field-Programmable Gate Array – System on Chip) hardware could subvert isolation; although such attacks lie outside the software scope of lifetime governance, the absence of intra-process misuse reduced the exploitability of post-breach conditions and provided seams where attestation hooks can be anchored. System heterogeneity and energy-aware scheduling further contextualised the tail findings. J. Kim *et al.* (2024) proposed proactive boosting, saying that the anticipated workload needs to balance responsiveness and energy across accelerators; the more predictable phase behaviour produced by disciplined promotions and audited teardowns offered cleaner signals to such schedulers, making boosts

timelier and avoiding surprise stalls from allocator slow paths. M. Țălu (2025) compared WebAssembly runtimes along performance and sandbox strength and identified integration trade-offs; the contract-based design here fit naturally above a sandbox, clarifying lifetime even when executing in a managed module and easing the safe integration of native components.

From an integration standpoint, multi-layered system design emphasised standardised interfaces to reduce emergent complexity. R. Vadisetty (2024) discussed multi-layered technologies that enable interoperability across heterogeneous stacks and argued that standardised seams improve reliability; the tier boundaries and transfer semantics formalised in this methodology provided precisely such seams at the memory-semantics level, enabling independent teams to reason about ownership and lifetime without relying on shared tribal knowledge. I. Sañudo *et al.* (2020) reviewed memory constraints as central to mission-critical reliability and linked endurance, bandwidth, and security to predictable behaviour; while the evaluation here stayed in neutral software exemplars, the measured narrowing of latency dispersion and the elimination of memory incidents resonated with that requirement for predictability in critical contexts. Overall, the findings demonstrate that explicitly governed lifetime tiers not only eliminate memory-safety defects but also enhance predictability, efficiency, and integration readiness across heterogeneous systems, bridging the gap between low-level safety mechanisms and high-level architectural reliability.

Conclusions

This study has introduced Layered Resource Isolation as a design methodology that reconciles memory safety with high performance by governing object lifetimes across three explicit tiers – ephemeral, verifiable, and persistent – each with bounded scope, entry/exit contracts, and mandatory validation checkpoints. Using neutral exemplars (parser/compiler front-ends, cache layering, numerical kernels) implemented with identical algorithms in baseline and LRI variants, it has been established that the methodology eliminated sanitiser-detectable leaks, double frees, use-after-free, and invalid frees within the detection horizon. Quantitatively, long-tail latency improved by roughly one quarter (p95: -21.8% to -24.9%; p99: -22.8% to -27.6%), peak resident memory declined by ~10-16%, and steady-state throughput increased modestly (+0.6% to +4.1%) and remained at +1.1% to +1.7%.

These findings indicate that promoting lifetime boundaries and transfers to first-class design elements displaces rare, expensive recovery paths with cheap, predictable boundary checks. Typestate-like encodings brought validation earlier in the pipeline, cross-tier alias guards prevented stale-handle misuse, pooled ephemeral arenas reduced allocator contention, and audited release ensured long-horizon hygiene. In aggregate, LRI provided whole-program reasoning about ownership and aliasing without dependence on vendor stacks or domain-specific workflows and remained compatible with C/C++ libraries and common toolchains. Conservative defaults were applied in the evaluation: promotion-time verification, ephemeral pools sized 256 to 512 KiB, and release batching aligned to phase boundaries; all measurements were taken under these settings. Accordingly, within the studied scope, the results align with the working (and secondary) hypotheses: a disciplined separation of lifetimes with validation gates reduces the probability and impact of memory-management errors and, via locality and deterministic reclamation, stabilises tail latencies, while granting cross-tier transfers first-class, contract-based status reinforces whole-program reasoning.

Overall, LRI has been shown to elevate memory safety to a first-class design property while preserving, and often improving, performance, offering an immediately adoptable strategy for advanced systems software. Limitations included the finite breadth of exemplars, seed-bounded execution windows, and allocator diversity not exhaustively sampled. Future work should extend assessment to months-long runs, additional languages and compilers, specialised allocators, and richer concurrency patterns; integrate static verification pipelines to auto-synthesise contracts; and develop tooling that scaffolds tier wiring and audit instrumentation.

Acknowledgements

None.

Funding

The study was not funded.

Conflict of Interest

None.

References

[1] Abuleil, R., Murrar, S., & Shkoukani, M. (2023). An enhanced approach for realizing robust security and isolation in virtualized environments. *International Journal of Advanced Computer Science and Applications*, 14(11), article number 141129. doi: 10.14569/ijacsa.2023.0141129.

[2] Adak, D., Zhou, H., Rotenberg, E., & Awad, A. (2025). SpecMPK: Efficient in-process isolation with speculative and secure permission update instruction. In *2025 IEEE international symposium on high performance computer architecture (HPCA)* (pp. 394-408). Las Vegas: Institute of Electrical and Electronics Engineers. doi: 10.1109/HPCA61900.2025.00039.

[3] Amar, S., *et al*. (2023). CHERIoT: Complete memory safety for embedded devices. In *MICRO '23: Proceedings of the 56th annual IEEE/ACM international symposium on microarchitecture* (pp. 641-653). New York: Association for Computing Machinery. doi: 10.1145/3613424.3614266.

[4] Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., & Summers, A.J. (2022). The prusti project: Formal verification for rust. In J.V. Deshmukh, K. Havelund & I. Perez (Eds.), *Proceeding of the 14th international symposium "NASA formal methods"* (pp. 88-108). Cham: Springer. doi: 10.1007/978-3-031-06773-0_5.
[5] Bazuku, R., Anab, A., Gyemerah, S., & Daabo, M.I. (2023). An overview of computer operating systems and emerging trends. *Asian Journal of Research in Computer Science*, 16(4), 161-177. doi: 10.9734/ajrcos/2023/v16i4380.
[6] Bugden, W., & Alahmar, A. (2022). The safety and performance of prominent programming languages. *International Journal of Software Engineering and Knowledge Engineering*, 32(5), 713-744. doi: 10.1142/s0218194022500231.
[7] Clerigo, N.D., & Teleron, J. (2025). A comparative study of memory management techniques and their optimization strategies. *International Journal of Advanced Research in Arts, Science, Engineering & Management*, 12(1), 39-50.
[8] Duy, K.D., Cho, K., Noh, T., & Lee, H. (2023). Capacity: Cryptographically-enforced in-process capabilities for modern ARM architectures. In *CCS'23: Proceedings of the 2023 ACM SIGSAC conference on computer and communications security* (pp. 874-888). New York: Association for Computing Machinery. doi: 10.1145/3576915.3623079.
[9] Fromherz, A., & Protzenko, J. (2024). Compiling C to safe rust, formalized. *ArXiv*. doi: 10.48550/arXiv.2412.15042.
[10] Greenspan, D., Mustafa, N.U., Delgado, A., & Bramham, C. (2024). LOaPP: Improving the performance of persistent memory objects via low-overhead at-rest PMO protection. In *2024 International symposium on secure and private execution environment design (SEED)* (pp. 131-142). Orlando: Institute of Electrical and Electronics Engineers. doi: 10.1109/SEED61283.2024.00023.
[11] Gross, M., Jacob, N., Zankl, A., & Sigl, G. (2021). Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC. *Journal of Cryptographic Engineering*, 12(2), 181-196. doi: 10.1007/s13389-021-00273-8.
[12] Hardin, D. (2023). Hardware/software co-assurance for the RUST programming language applied to Zero Trust architecture development. *ACM SIGAda Ada Letters*, 42(2), 55-61. doi: 10.1145/3591335.3591340.
[13] Huang, H., Wang, H., Rao, J., Wu, S., Fan, H., Yu, C., Jin, H., Suo, K., & Pan, L. (2024). VKernel: Enhancing container isolation via private code and data. *IEEE Transactions on Computers*, 73(7), 1711-1723. doi: 10.1109/tc.2024.3383988.
[14] Jalaman, J.R.C., & Teleron, J.I. (2024). Optimizing operating system performance through advanced memory management techniques: A comprehensive study and implementation. *Engineering and Technology Journal*, 9(5), 4137-4143. doi: 10.47191/etj/v9i05.33.
[15] Joseph, B., & Kavitha, R. (2025). A radiation hardened in-memory computing SRAM for soft error tolerance in safety critical applications. *AEU – International Journal of Electronics and Communications*, 202, article number 156017. doi: 10.1016/j.aeue.2025.156017.
[16] Kim, J., & Lee, K. (2020). I/O resource isolation of public cloud serverless function runtimes for data-intensive applications. *Cluster Computing*, 23(3), 2249-2259. doi: 10.1007/s10586-020-03103-4.
[17] Kim, J., Lee, G., & Choi, H. (2024). Energy-efficient heterogeneous computing via normalized performance based proactive boost for embedded artificial intelligence. In *2024 IEEE international conference on consumer electronics (ICCE)* (pp. 1-6). Las Vegas: Institute of Electrical and Electronics Engineers. doi: 10.1109/ICCE59016.2024.10444347.
[18] Kothapalli, M. (2021). Securing microservices architecture: Best practices and challenges. *Journal of Scientific and Engineering Research*, 8(10), 187-192. doi: 10.5281/zenodo.12772079.
[19] Michael, A.E., *et al*. (2023). MSWasm: Soundly enforcing memory-safe execution of unsafe code. *Proceedings of the ACM on Programming Languages*, 7(POPL), 425-454. doi: 10.1145/3571208.
[20] Partap, A., & Boneh, D. (2022). Memory tagging: A memory efficient design. *ArXiv*. .doi: 10.48550/arXiv.2209.00307.
[21] Sañudo, I., Cortimiglia, P., Miccio, L., Solieri, M., Burgio, P., Di Biagio, C., Felici, F., Nuzzo, G., & Bertogna, M. (2020). The key role of memory in next-generation embedded systems for military applications. In P. Ciancarini, M. Mazzara, A. Messina, A. Sillitti & G. Succi (Eds.), *Proceedings of 6th international conference in software engineering for defence applications* (pp. 275-287). Cham: Springer. doi: 10.1007/978-3-030-14687-0_25.
[22] Sharma, K., & Khurana, P. (2025). A deep dive into container security challenges, strategies, and solutions. In *Proceedings of the international conference on recent advances in artificial intelligence for sustainable development (RAISD 2025)* (pp. 484-495). Dordrecht: Atlantis Press. doi: 10.2991/978-94-6463-787-8_38.
[23] Ţălu, M. (2025). A comparative study of WebAssembly runtimes: performance metrics, integration challenges, application domains, and security features. *Archives of Advanced Engineering Science*, 1-13. doi: 10.47852/bonviewaaes52024965.
[24] Unterguggenberger, M., Lamster, L., Schrammel, D., Schwarzl, M., & Mangard, S. (2024). TME-box: Scalable in-process isolation through intel TME-MK memory encryption. In *Network and distributed system security symposium 2025: NDSS 2025* (pp. 1-16). San Diego: Network and Distributed System Security. doi: 10.14722/ndss.2025.240277.

[25] Vadisetty, R. (2024). Multi layered cloud technologies to achieve interoperability in AI. In *2024 international conference on intelligent computing and emerging communication technologies (ICEC)* (pp. 1-5). Guntur: Institute of Electrical and Electronics Engineers. doi: 10.1109/ICEC59683.2024.10837471.

[26] Vaithianathan, M. (2025). Memory hierarchy optimization strategies for high-performance computing architectures. *International Journal of Emerging Trends & Technology in Computer Science*, 6(1), 24-35. doi: 10.63282/3050-9246.IJETCSIT-V6I1P103.

[27] Waseem, M., Ahmad, A., Liang, P., Akbar, M.A., Khan, A.A., Ahmad, I., Setälä, M., & Mikkonen, T. (2025). Containerization in multi-cloud environment: Roles, strategies, challenges, and solutions for effective implementation. *Journal of Systems and Software*, 230, article number 112558. doi: 10.1016/j.jss.2025.112558.

[28] Watson, R.N., *et al*. (2025). It is time to standardize principles and practices for software memory safety. *Communications of the ACM*, 68(2), 40-45. doi: 10.1145/3708553.

[29] Weis, T., Waltereit, M., & Uphoff, M. (2019). Fyr: A memory-safe and thread-safe systems programming language. In *SAC '19: Proceedings of the 34th ACM/SIGAPP symposium on applied computing* (pp. 1574-1577). New York: Association for Computing Machinery. doi: 10.1145/3297280.3299741.

[30] Wong, A.Y., Chekole, E.G., Ochoa, M., & Zhou, J. (2023). On the security of containers: Threat modeling, attack analysis, and mitigation strategies. *Computers & Security*, 128, article number 103140. doi: 10.1016/j.cose.2023.103140.

[31] Xu, S., Wang, Y., Lei, L., Sun, K., Jing, J., Ma, S., Wang, J., & Huang, H. (2024). Condo: Enhancing container isolation through kernel permission data protection. *IEEE Transactions on Information Forensics and Security*, 19, 6168-6183. doi: 10.1109/tifs.2024.3411915.

[32] Yoshimura, N., Oyama, H., & Azumi, T. (2024). TECS/Rust: Memory-safe component framework for embedded systems. In *2024 IEEE 27th international symposium on real-time distributed computing (ISORC)* (pp. 1-11). Tunis: Institute of Electrical and Electronics Engineers. doi: 10.1109/ISORC61049.2024.10551370.

Методологія проектування безпечних для пам'яті високопродуктивних застосунків з використанням багаторівневої ізоляції ресурсів

Ольга Красножон
Магістр
Міжнародний університет економіки та гуманітарних наук імені академіка Степана Дем'янчука
33000, вул. С. Дем'янчука, 4, м. Рівне, Україна
https://orcid.org/0009-0008-0202-9575

Анотація. У цьому дослідженні представлено стратегію проектування – багаторівневу ізоляцію ресурсів, яка поєднує безпеку пам'яті з високою продуктивністю шляхом застосування трьох явних рівнів терміну служби та перевірок: ефемерний рівень для короткочасних тимчасових ресурсів, рівень, що перевіряється, захищений структурною та аліасинговою перевіркою в точках передачі, та постійний рівень з аудитованою перевіркою. Метою було підвищити межі терміну служби елементів проектування до першокласних, уникаючи при цьому специфічних фреймворків для постачальників. Нейтральні зразки зберігали ідентичні алгоритми в базових та багаторівневих варіантах: синтаксичний інтерфейс та інтерфейс компілятора, який перетворює потоки токенів на абстрактні синтаксичні дерева, багаторівневий кеш з узгодженою поведінкою зчитування та блоковані числові ядра. В оцінюванні використовувалися інструментальні розподіли, підвищення, перевірені випуски та часові рамки фаз, а також парні прогони по тридцятьох незалежних початкових значеннях для порівняння інцидентів безпеки на десять мільйонів операцій, медіанного часу виконання, затримок дев'яносто п'ятого та дев'яносто дев'ятого процентилів, пропускної здатності та пікового обсягу резидентної пам'яті. Результати показали усунення витоків, подвійних звільнень, звільнень після звільнення та недійсних звільнень у межах горизонту виявлення у всіх багаторівневих варіантах, з односторонньою довірчою межею, яка встановлювала рівень інцидентів нижче 0,11 на десять мільйонів операцій. Поведінка хвостів помітно покращилася: дев'яносто п'яті процентилі зменшилися на 21,8–24,9 %, а дев'яносто дев'яті процентилі – на 22,8–27,6 % у всіх екземплярах та режимах навантаження, піковий обсяг резидентної пам'яті знизився на 10–16 %, пропускна здатність у стаціонарному стані зросла на 0,6–4,1 %, а медіанні накладні витрати часу виконання залишилися близько 1–2 %. Практично, цей підхід зменшив конкуренцію за розподільники ресурсів, дозволив цілісній програмі обмірковувати володіння та псевдоніми, а також перетворив рідкісне та дороге відновлення на передбачувану перевірку меж, пропонуючи відтворювану методологію для передового системного програмного забезпечення

Ключові слова: аудитований випуск; контроль володіння та псевдонімів; контрольні точки перевірки; контракти розподільника та дескриптора; захист псевдонімів; кодування стану типів