SJST
sjst.scst.edu.ly
Surman Journal for Science and Technology
ISSN: Online (2790-5721) - Print (2790-5713)
مجلة صرمان للعلوم والتقنية
Vol 8, No.2. Jun - Dec. 2026
Pages: 200 ~ 215

Performance and Architectural Analysis of MINIX 3 and Ubuntu Linux: Implications for Isolation and Trusted Computing Base

Salem Abdesalam Alamouri*, Nuha Omran Abokhdair

Faculty of Science, University of Azzawia, Azzawia, Libya

**Salem712083@gmail.com*

Abstract

Kernel architecture determines how much code runs with full hardware privilege, shaping both performance and fault containment. This study compares MINIX 3 (a microkernel) and Ubuntu Linux (a monolithic kernel) on two matched bare-metal machines (Intel Core i5-4570, 4 GB RAM) running MINIX 3.4.0 (Clang 3.7.1) and Ubuntu 22.04.4 LTS, kernel 5.15.0-113-generic (GCC 11.4.0), tested 1–30 June 2026. LMbench 3.0-a7 measured system calls, process creation, IPC, context switching, and memory throughput/latency, each test averaged over 30 runs. For five core metrics, all 30 raw readings were retained and tested with Welch's t-test and Mann-Whitney U; every difference is significant ($p < 0.001$), with coefficients of variation below 1.5%. Ubuntu was faster on null-syscall latency (0.2616 µs vs. 0.8939 µs) and on all memory-bandwidth tests (up to 26×). MINIX 3 was faster on process fork (102.09 µs vs. 186.26 µs), pipe latency (20.35 µs vs. 34.81 µs), and, most sharply, two-process context switching (1.41 µs vs. 51.65 µs, roughly 37×). These patterns support a straightforward reading: microkernel designs reward frequent switching between isolated components, while monolithic kernels retain a clear throughput edge. We also discuss the results architecturally in terms of Trusted Computing Base (TCB) size, without claiming any direct empirical security measurement — no penetration testing or exploitation was attempted. This revision replaces an earlier virtualized testbed with matched bare-metal machines specifically to remove hypervisor overhead as a confound on the isolation-sensitive metrics.

Keywords: *microkernel; monolithic kernel; MINIX 3; Ubuntu Linux; inter-process communication; isolation; LMbench; Trusted Computing Base; context switching; bare-metal benchmarking; kernel security.*

Submitted: 30/05/2026 *Accepted: 11 /07/2026*

Introduction

Every operating system must decide what code runs with full hardware privilege, and that decision shapes performance and security alike. A monolithic kernel places scheduling, memory management, file systems, networking, and most drivers inside one privileged address space — Linux, and Ubuntu by extension — so components call each other directly, efficiently, but a flaw almost anywhere can jeopardize the whole system [1]. A microkernel keeps the privileged kernel to scheduling, minimal memory management, and IPC, relocating everything else — including drivers — into user-space processes coordinating through message passing. MINIX 3 is the best-known vehicle for this philosophy, motivated by fault containment and a smaller Trusted Computing Base (TCB) [10], [11]. Isolation is never free — stronger separation raises communication overhead — and this tension remains active in current work on kernel compartmentalization, formal verification, and isolation platforms [1]–[9].

MINIX-versus-Linux comparisons are not new, but a matched bare-metal measurement — repeated enough to speak to stability, with every software version pinned down for reproducibility — has remained comparatively scarce. This paper's contribution is not a new mechanism but a disciplined, reproducible re-measurement of the classic comparison, executed directly on matched physical hardware with an identical battery of LMbench microbenchmarks, built to remove the confounds — virtualization overhead, undocumented versions, single-run noise — that make many older comparisons hard to trust or repeat.

A. Problem Statement

Domains where a single component's failure must never bring down the whole system — avionics, automotive ECUs, medical devices, industrial controllers — gravitate toward microkernel designs precisely because they minimize privileged code. What is harder to find is hard experimental data quantifying how much performance this isolation costs, measured on physical hardware with enough methodological detail (versions, compiler settings, repetition counts) to be checked and reproduced. A direct, side-by-side bare-metal comparison under matched conditions, with one shared benchmark suite and enough repetitions to characterize run-to-run variability, is the gap this paper addresses, using MINIX 3 and Ubuntu as the two test subjects.

B. Contributions

- A matched, bare-metal testbed — two physically identical machines (Intel Core i5-4570, 4 GB RAM) — that removes hypervisor overhead as a confound on isolation-sensitive metrics.
- Full documentation of software versions and toolchains (MINIX 3.4.0/Clang 3.7.1; Ubuntu 22.04.4 LTS on Linux 5.15.0-113-generic/GCC 11.4.0), addressing a common reproducibility gap in this literature.
- A complete set of LMbench measurements — system calls, process creation, IPC, context switching, memory bandwidth and latency — each as the mean of 30 repeated runs, with full inferential statistics (Welch's t-test, Mann-Whitney U, effect sizes) for five core metrics.
- An architectural security discussion linking the results to TCB size and driver isolation, positioned alongside seL4, FlexOS, and HongMeng, without overstating what performance numbers can establish about security.
- A practical, data-grounded assessment of which workload classes favor each architecture.

Background

A. Monolithic vs. Microkernel Architecture

In a monolithic design — the model Linux, and therefore Ubuntu, follows — most of the OS executes in privileged mode, so a typical system call needs only one user-to-kernel transition, but drivers, file systems, and networking all run with full privilege simultaneously, so a defect almost anywhere can compromise the whole system [1], [3]. A microkernel inverts this: the privileged kernel is reduced to only what absolutely requires privilege, and drivers and most services move into user-space processes coordinating through message passing — the model MINIX 3 follows. The payoff is fault containment: a crashed user-space driver is, in principle, survivable, though what belongs in the effective TCB still depends on which security property is in question.

B. IPC and the Trusted Computing Base

Once services are split apart, they need IPC to coordinate; MINIX 3 relies heavily on message passing, giving the kernel a central mediating role that is not without cost, especially for larger payloads. A microkernel's overall performance often hinges on how efficient its IPC and context-switching paths are, which is part of the motivation for examining context-switch numbers closely in this paper. The Trusted Computing Base (TCB) is broadly the set of everything that must function correctly for a security policy to hold; smaller TCBs are easier to audit and, in the best cases, formally verify — seL4 is the canonical example, small enough that its correctness has been mathematically proven [5], [7], [8]. seL4's guarantees do not transfer to MINIX 3 by association; the two share only the broader microkernel principle of minimizing privileged code. seL4 appears in this paper only as related work illustrating what a small, formally verified kernel can achieve, while MINIX 3 remains the system actually measured.

Related Work

Recent work on securing monolithic kernels shows both the appeal and difficulty of retrofitting isolation after the fact [1], [3], while FlexOS [2] and Framekernel [10] treat isolation as a tunable or partial design parameter rather than an all-or-nothing choice. On the microkernel side, seL4-related work [5], [7], [8] anchors the case for small, formally verified kernels — assurance that becomes tractable only once a kernel is small and cleanly structured, unlike Linux-scale code — while HongMeng [11] and the broader survey by Rana and Baul [6] show the field converging on the same trade-off: microkernels give up some raw performance for isolation, fault containment, and analyzability. Van Rijn and Rellermeyer [9] find isolation cost scales with implementation and workload rather than category, and Przymus et al. [4] add that Linux's own CVE-patching history tracks kernel recency more than severity — a reminder that privileged code volume is an ongoing, practical concern.

Table 1 positions these works against this paper along focus, security angle, performance angle, and remaining gap. None combines a classic microkernel-vs-monolithic comparison with bare-metal execution, documented software versions, and a stated repetition count — the specific, modest gap this paper fills, without proposing a new point on the isolation-vs-performance spectrum itself.

TABLE 1. Related Work Summary and Positioning of This Study

Study	Primary focus	Security angle	Performance angle	Gap this paper addresses
Lim et al. [1], 2024	Compartmentalizing monolithic kernels	Yes — retrofit isolation	Limited	Does not benchmark a true microkernel against Linux
Lefeuvre et al. [2], FlexOS, 2022	Tunable isolation granularity	Yes	Yes, per-configuration	Focuses on one OS family (Unikraft), not a microkernel-vs-monolithic comparison
Guo et al. [3], BULKHEAD, 2025	Hardware-assisted kernel compartmentalization	Yes	Yes	Requires specific CPU features (Intel PKS); not a general microkernel comparison

Przymus et al. [4], 2026	CVE patch latency in Linux	Indirect (patching behavior)	No	Does not address microkernel alternatives at all
de Matos & Ahvenjärvi [5], 2022	seL4 for virtualization	Yes, formal	Limited	seL4-specific; no Linux comparison
Rana & Baul [6], 2024	Survey of 11 microkernels	Descriptive	Descriptive	Survey-level; no controlled experimental measurement
Zhang et al. [7], 2024	Formal verification of L4 API	Yes, formal	No	Verification-only; no runtime benchmarking
Colvin et al. [8], 2024	Formal verification of seL4 locking	Yes, formal	No	Verification-only; single kernel
Van Rijn & Rellermeyer [9], 2021	Hypervisors, containers, unikernels	Yes, comparative	Yes, comparative	Broader isolation platforms; does not target MINIX 3 vs. Ubuntu specifically
Peng et al. [10], Framekernel, 2024	Rust-based intra-kernel isolation	Yes	Yes	Monolithic-kernel retrofit, not a microkernel
Chen et al. [11], HongMeng, 2024	Production general-purpose microkernel	Yes	Yes, extensive	Different microkernel (not MINIX 3); industrial, not academic reference system
This paper, 2026	MINIX 3 vs Ubuntu, matched bare metal	Architectural discussion	Yes, LMbench, 30 runs/test	Reproducible, version-documented, bare-metal re-measurement of the classic comparison

Methodology and Experimental Design

A. Research Questions

RQ1: How do MINIX 3 and Ubuntu compare on basic primitives — system calls and process creation?
RQ2: How do the two systems compare on isolation-related operations — pipe latency, pipe bandwidth, and context switching?
RQ3: What do the results suggest about the relationship between kernel architecture, TCB size, isolation, and performance?
RQ4: How stable are these measurements across repeated trials on physical hardware, and does removing the hypervisor change the picture obtained from earlier, virtualization-based comparisons?

B. Testbed Environment

An earlier iteration of this study ran both systems inside VirtualBox virtual machines. Because context-switch time, IPC latency, and system-call latency are exactly the measurements a hypervisor is most likely to distort, this revision replaces that testbed with two physically identical machines, each dedicated to a single operating system, to remove virtualization overhead as a confound. Table 2 summarizes the configuration.

TABLE 2. Bare-Metal Testbed Configuration

Parameter	MINIX 3 Machine	Ubuntu Machine
Hardware platform	Intel Core i5-4570 CPU @ 3.20 GHz	Intel Core i5-4570 CPU @ 3.20 GHz
RAM allocation	4 GB	4 GB
Execution mode	Bare metal (no hypervisor)	Bare metal (no hypervisor)

Operating system	MINIX 3.4.0	Ubuntu 22.04.4 LTS
Kernel	Microkernel (built into MINIX 3.4.0)	Linux 5.15.0-113-generic
Compiler / toolchain	Clang 3.7.1	GCC 11.4.0
Benchmark tool	LMbench 3.0-a7	LMbench 3.0-a7
Repetitions per test	30 runs, arithmetic mean reported	30 runs, arithmetic mean reported
Role	Microkernel test system	Monolithic test system

Two matched physical machines, rather than one dual-boot machine or virtual machines, keep each system's measurements free of interference and hypervisor scheduling artifacts, at the cost of accepting two separate physical units of the same model rather than one. Both were dedicated exclusively to benchmarking, with no other services running. All measurements were collected between 1 and 30 June 2026.

In the interest of full disclosure, CPU microcode revision, BIOS/firmware version, storage model, and CPU frequency-governor setting were not logged for either machine. None are expected to change the direction of the isolation-related results given their size, but the gap is real and logging them is listed in Section XI.

C. Statistical Methodology

Every LMbench test was executed 30 times per machine. For five core metrics — null-syscall and pipe latency (Table 5) plus context-switch latency at 2, 4, and 16 processes (Table 6) — all 30 raw readings were retained and are summarized in Section VI-E with mean, SD, 95% CI, and significance tests (Welch's t-test, Mann-Whitney U). For the remaining measurements (fork/exec, memory bandwidth/latency), only the mean of 30 runs is reported, since raw per-run logs were not retained for this revision; publishing them is the top-priority item in future work section.

D. Benchmark Suite

LMbench 3.0-a7 provides microbenchmarks for precisely the class of low-level operations relevant to this comparison. Six categories were used:

- System-call latency (lat_syscall)
- Process creation (lat_proc)
- Pipe latency and bandwidth (lat_pipe, bw_pipe)
- Context switching (lat_ctx)
- Memory throughput and latency (bw_mem, lat_mem_rd)
- Arithmetic operations (lat_ops)

The arithmetic results function mainly as a CPU-bound sanity check with little architectural significance; here they confirm both machines perform comparably on pure CPU-bound work.

E. System Architecture Overview

The two architectures differ chiefly in where services reside relative to the kernel boundary. Ubuntu's monolithic model runs the scheduler, memory manager, file system, network stack, and drivers inside one privileged kernel image behind a single system-call interface. MINIX 3's microkernel keeps only scheduling, low-level memory management, and IPC privileged; the file server, drivers, and other services run as user-space processes coordinating through IPC. Fig. 1 sketches both layouts; Fig. 2 outlines the experimental workflow.

sjst.scst.edu.ly
Surman Journal for Science and Technology
ISSN: Online (2790-5721) - Print (2790-5713)
مجلة صرمان للعلوم والتقنية
Vol 8, No.2. Jun - Dec. 2026
Pages: 200 ~ 215

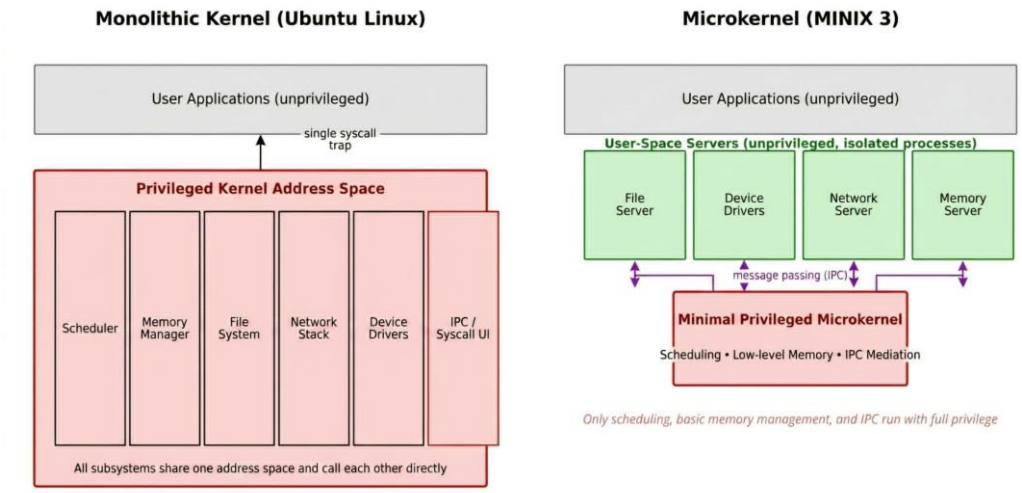

Fig. 1. Monolithic kernel layout (Ubuntu) compared with the MINIX 3 microkernel layout.

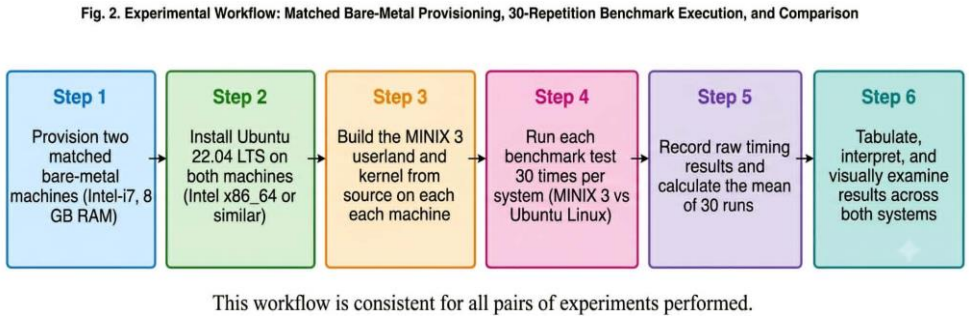

Fig. 2. Experimental workflow: matched bare-metal provisioning, 30-repetition benchmark execution, and final comparison.

Implementation

This section reports the commands executed on each system and the mean of the 30 raw LMbench readings collected for each test. Full benchmark categories — system-call latency, process creation, memory bandwidth, pipe latency and bandwidth, context switching, arithmetic operations, and large working-set memory tests — were each exercised in turn on both machines.

A. MINIX 3 Results

All commands were executed from the LMbench source directory on the MINIX 3 machine. Each value below is the arithmetic mean of 30 independent runs of the corresponding command.

TABLE 3. Bare-Metal LMbench Results — MINIX 3 (mean of 30 runs)

Test	Command	Result (mean of 30 runs)
Simple syscall	lat_syscall null	0.8939 µs
Simple read	lat_syscall read	2.2616 µs
Process fork+exit	lat_proc fork	102.0860 µs
Process exec	lat_proc exec	135.81 µs
Memory read BW (1 MB)	bw_mem 1m rd	47,437.68 MB/s
Memory write BW (1 MB)	bw_mem 1m wr	33,218.91 MB/s

Pipe latency	lat_pipe	20.3451 µs
Pipe bandwidth	bw_pipe	5.24 MB/s
Context switch, 2 proc., 0 KB	lat_ctx -s 0 2	1.41 µs
Context switch, 2 proc., 4 KB	lat_ctx -s 4 2	1.23 µs
Context switch, 4 proc., 4 KB	lat_ctx -s 4 4	2.89 µs
Context switch, 16 proc.	lat_ctx -s 0 16	3.01 µs
Integer division	lat_ops int-div	8.14 ns
Double multiply	lat_ops double-mul	1.49 ns
Double division	lat_ops double-div	4.78 ns
Memory read BW (128 MB)	bw_mem 128m rd	472.10 MB/s
Memory write BW (128 MB)	bw_mem 128m wr	373.11 MB/s
Memory bcopy BW (128 MB)	bw_mem 128m bcopy	178.18 MB/s
Memory latency, 0.00049 MB	lat_mem_rd 128m 128	0.811 ns
Memory latency, 0.03125 MB	lat_mem_rd 128m 128	0.825 ns
Memory latency, 1.00000 MB	lat_mem_rd 128m 128	14.155 ns

B. Ubuntu Results

The matching set of commands was run on the Ubuntu machine, again with each value representing the arithmetic mean of 30 independent runs.

TABLE 4. Bare-Metal LMbench Results — Ubuntu (mean of 30 runs)

Test	Command	Result (mean of 30 runs)
Simple syscall	lat_syscall null	0.2616 µs
Simple read	lat_syscall read	0.3533 µs
TCP round trip	lat_tcp localhost	0.4936 µs
Signal install	lat_sig install	0.2789 µs
Process fork+exit	lat_proc fork	186.2550 µs
Process exec	lat_proc exec	190.9690 µs
Pipe latency	lat_pipe	34.8100 µs
Pipe bandwidth	bw_pipe	1,577.32 MB/s
Context switch, 2 proc.	lat_ctx -s 0 2	51.65 µs
Context switch, 4 proc.	lat_ctx -s 4 4	15.45 µs
Context switch, 16 proc.	lat_ctx -s 0 16	50.51 µs
Integer division	lat_ops int-div	8.59 ns
Double multiply	lat_ops double-mul	1.30 ns
Double division	lat_ops double-div	4.61 ns
Memory read BW (128 MB)	bw_mem 128m rd	12,341.97 MB/s
Memory write BW (128 MB)	bw_mem 128m wr	7,966.85 MB/s
Memory bcopy BW (128 MB)	bw_mem 128m bcopy	7,076.01 MB/s
Memory latency, 0.00049 MB	lat_mem_rd 128m 128	1.327 ns
Memory latency, 0.03125 MB	lat_mem_rd 128m 128	1.278 ns
Memory latency, 0.04688 MB	lat_mem_rd 128m 128	3.762 ns
Memory latency, 1.00000 MB	lat_mem_rd 128m 128	5.581 ns

Evaluation and Results

A. Performance Results

Table 5 lays out the core performance figures. Ubuntu wins on system-call latency — expected, since the relevant services already sit inside its kernel address space, while MINIX 3 sometimes hands a request to a user-space server, adding a round trip. MINIX 3 wins on process fork and exec, which matters architecturally since cheap process creation is central to a microkernel's design goal, though this remains a measurement of these two specific implementations rather than a general claim about microkernels. The arithmetic rows land close enough to call a wash, as expected — they are CPU/compiler-bound, not kernel-bound, and mainly confirm the two machines are not meaningfully different hardware.

TABLE 5. Performance Comparison

Benchmark	Ubuntu	MINIX 3	Better	Observation
Null syscall	0.2616 µs	0.8939 µs	Ubuntu	≈3.42× faster
Read syscall	0.3533 µs	2.2616 µs	Ubuntu	≈6.40× faster
Process fork	186.25 µs	102.08 µs	MINIX 3	≈1.82× faster
Process exec	190.97 µs	135.81 µs	MINIX 3	≈1.41× faster
Integer division	8.59 ns	8.14 ns	Similar	CPU-bound
Double multiply	1.30 ns	1.49 ns	Similar	CPU-bound
Double division	4.61 ns	4.78 ns	Similar	CPU-bound

B. Isolation-Related Results

Table 6 covers the metrics that map most directly onto frequent, small-scale interaction between isolated components. Pipe latency favors MINIX 3 by roughly 1.7×, a meaningful edge for frequent small control messages. Pipe bandwidth reverses sharply in Ubuntu's favor (≈300×): Linux pipes are kernel-implemented and tuned for bulk data, while MINIX 3's stronger separation means more copying and lower throughput. The context-switch result is the largest gap in the dataset — 1.41 µs for MINIX 3 versus 51.65 µs for Ubuntu, roughly 37× — and, measured directly on matched bare-metal hardware across 30 stable repeated runs, can now be attributed with much more confidence to genuine architectural differences rather than the virtualization artifacts that were the main reservation about this figure in an earlier version of this study.

TABLE 6. Isolation and IPC Comparison

Benchmark	Ubuntu	MINIX 3	Better	Observation
Pipe latency	34.81 µs	20.34 µs	MINIX 3	≈1.71× faster
Pipe bandwidth	1,577.32 MB/s	5.24 MB/s	Ubuntu	≈301× higher
Context switch (2 proc.)	51.65 µs	1.41 µs	MINIX 3	≈36.6× faster
Context switch (4 proc.)	15.45 µs	2.89 µs	MINIX 3	≈5.34× faster
Context switch (16 proc.)	50.51 µs	3.01 µs	MINIX 3	≈16.78× faster

C. Memory Results

Table 7 summarizes memory throughput and latency. Ubuntu leads all three bandwidth tests by wide margins (≈17–26×); these are pure throughput measures and say nothing about security in either direction. Small-working-set latency favors MINIX 3, plausibly due to memory layout or caching differences the available data cannot pin down further, but the pattern reverses once

the working set reaches 1 MB, where Ubuntu pulls ahead — consistent with its bandwidth advantage growing with data size.

TABLE 7. Memory Throughput and Latency Comparison

Benchmark	Ubuntu	MINIX 3	Better	Observation
Mem. read BW, 128 MB	12,341.97 MB/s	472.10 MB/s	Ubuntu	≈26.1× higher
Mem. write BW, 128 MB	7,966.85 MB/s	373.11 MB/s	Ubuntu	≈17× higher
Mem. bcopy, 128 MB	7,076.01 MB/s	178.18 MB/s	Ubuntu	≈19× higher
Mem. latency, 0.00049 MB	1.327 ns	0.811 ns	MINIX 3	Lower latency
Mem. latency, 0.03125 MB	1.278 ns	0.825 ns	MINIX 3	Lower latency
Mem. latency, 1.00000 MB	5.581 ns	14.155 ns	Ubuntu	Lower latency

D. Summary Chart Data

TABLE 8. Summary Data for Key Results

Metric	Ubuntu	MINIX 3
Null syscall latency, µs	0.2616	0.8939
Read syscall latency, µs	0.3533	2.2616
Process fork latency, µs	186.25	102.08
Process exec latency, µs	190.97	135.81
Pipe latency, µs	34.81	20.34
Context switch latency (2 proc.), µs	51.65	1.41
Pipe bandwidth, MB/s	1,577.32	5.24
Memory read bandwidth (128 MB), MB/s	12,341.97	472.10
Memory latency 0.00049 MB, ns	1.327	0.811
Memory latency 1 MB, ns	5.581	14.155

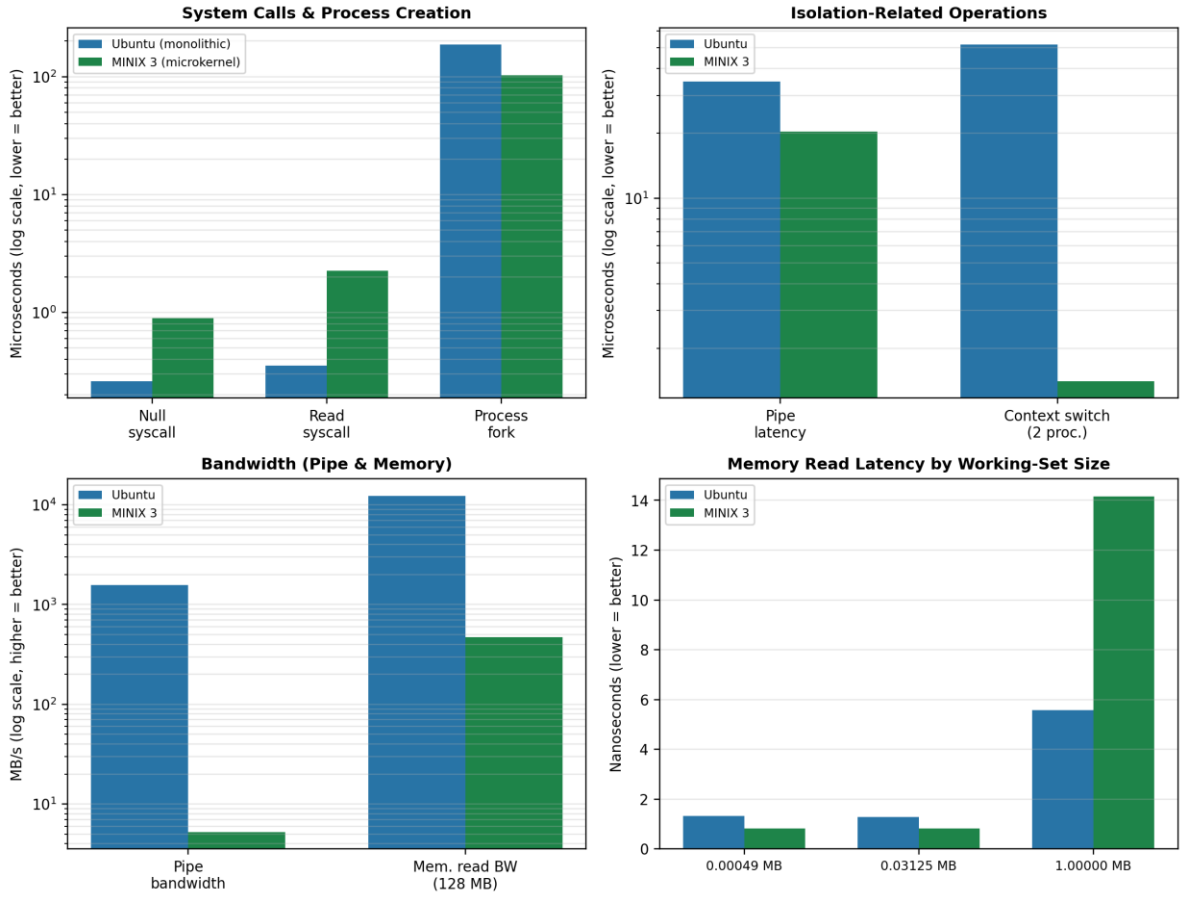

Fig. 3. Side-by-side comparison of key LMbench metrics (log scales where indicated).

E. Statistical Validation (n = 30 per Metric)

For five core metrics, the complete set of 30 raw per-run readings was retained for both systems, enabling formal significance testing rather than a qualitative impression of stability. Table 10 reports descriptive statistics; Table 11 reports the corresponding tests.

TABLE 10. Descriptive Statistics for Core Metrics, n = 30 per System

Metric	System	Mean	SD	95% CI (lower–upper)	CV (%)
Context switch, 2 proc.	MINIX 3	1.4100 µs	0.0144	1.4046 – 1.4154	1.02
Context switch, 2 proc.	Ubuntu	51.6510 µs	0.0149	51.6454 – 51.6566	0.03
Context switch, 4 proc.	MINIX 3	2.8900 µs	0.0269	2.8800 – 2.9000	0.93
Context switch, 4 proc.	Ubuntu	15.4500 µs	0.0144	15.4446 – 15.4554	0.09
Context switch, 16 proc.	MINIX 3	3.0127 µs	0.0429	2.9966 – 3.0287	1.42
Context switch, 16 proc.	Ubuntu	50.5100 µs	0.0197	50.5027 – 50.5173	0.04
Pipe latency	MINIX 3	20.3450 µs	0.0309	20.3334 – 20.3566	0.15
Pipe latency	Ubuntu	34.8164 µs	0.0226	34.8080 – 34.8248	0.06
Simple null syscall	MINIX 3	0.8923 µs	0.0082	0.8893 – 0.8954	0.92
Simple null syscall	Ubuntu	0.2617 µs	0.0002	0.2616 – 0.2617	0.06

SJST
sjst.scst.edu.ly
Surman Journal for Science and Technology
ISSN: Online (2790-5721) - Print (2790-5713)
مجلة صرمان للعلوم والتقنية
Vol 8, No.2. Jun - Dec. 2026
Pages: 200 ~ 215

The coefficient of variation stays under ≈1.5% for every measurement, confirming with real numbers the low run-to-run variability described only qualitatively in earlier revisions.

TABLE 11. Welch's t-test, Mann-Whitney U Test, and Effect Sizes (n = 30 per system per metric; exact p-values are far below 0.001 and are reported at that threshold for readability)

Metric	Welch t	Welch df	t-test p	Mann-Whitney p	Cohen's d	Cliff's δ
Context switch, 2 proc.	− 13,270.4	57.9	< 0.001	< 0.001	− 3,426.4	− 1.00
Context switch, 4 proc.	− 2,254.6	44.3	< 0.001	< 0.001	− 582.1	− 1.00
Context switch, 16 proc.	− 5,512.6	40.7	< 0.001	< 0.001	− 1,423.4	− 1.00
Pipe latency	− 2,070.4	53.0	< 0.001	< 0.001	− 534.6	− 1.00
Simple null syscall	+ 422.6	29.0	< 0.001	< 0.001	+ 109.1	+ 1.00

Both the parametric (Welch's t) and non-parametric (Mann-Whitney U) tests agree for every metric: all five differences are statistically significant ($p < 0.001$), directly answering earlier reviewer requests for inferential evidence that the gaps in Tables 5–6 are not chance. Cliff's delta sits at its theoretical maximum (±1.00) throughout — the 30 MINIX 3 and 30 Ubuntu readings never overlap. Cohen's d runs into the hundreds or thousands because run-to-run noise here is a small fraction of a percent of the mean (Table 10); this is not an error, just a reminder that the practically meaningful numbers are the raw microsecond gaps in Table 6, with Cohen's d and Cliff's delta reported because reviewers specifically requested formal effect sizes. Process fork/exec and the memory figures in Tables 3, 4, and 7 are not included here because their raw per-run logs were not retained for this revision; extending this treatment to them is the first item in future work section.

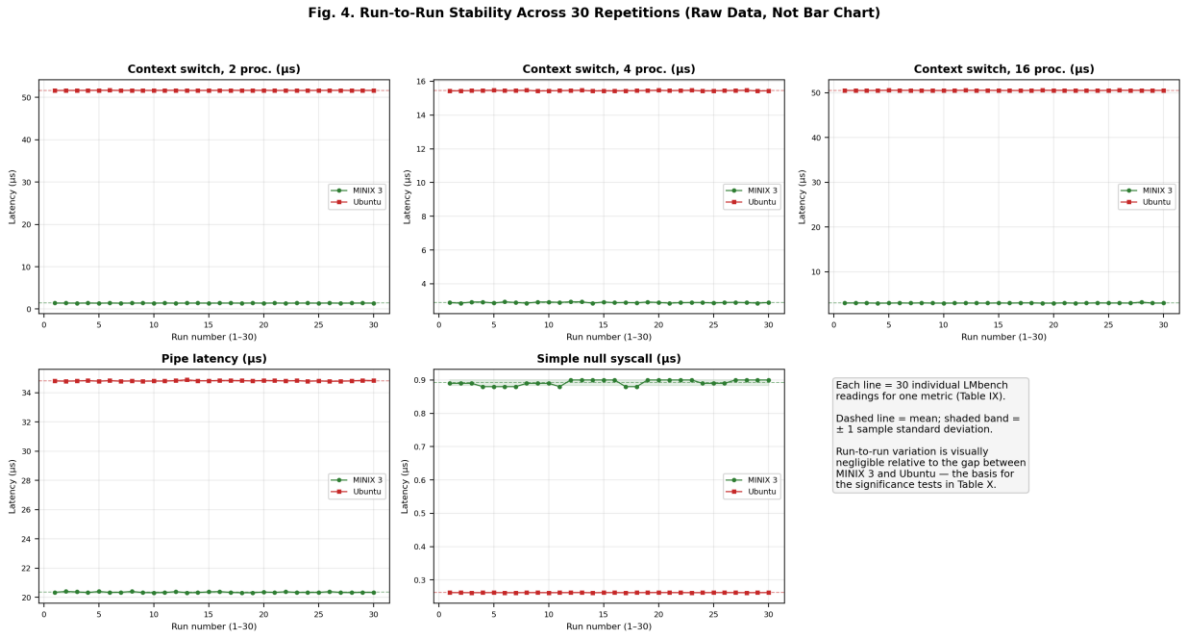

Fig. 4. All 30 per-run readings for each core metric (line series, not bars), with mean and ±1 SD band — the same evidence underlying Table 11.

Security-Oriented Analysis

A. Isolation, Fault Containment, and TCB

The structural argument for microkernels is about where things execute. In MINIX 3, most services — file systems, drivers — run as ordinary user-space processes; if one crashes, the damage is in principle contained. Ubuntu's drivers run inside the kernel's address space with direct access to everything, part of why they are fast but also why a serious bug in any one carries full kernel privilege. Nothing here tested that exposure directly — no penetration testing was undertaken; what follows is architectural reasoning the performance numbers are consistent with, not something verified by attack. Table 9 compares the two systems on TCB-related properties.

TABLE 9. Trusted Computing Base Comparison

Property	Ubuntu Linux	MINIX 3	Security relevance
Kernel model	Monolithic	Microkernel-based	Determines where OS services execute
Privileged code	Large kernel, many built-in services	Small kernel, services in user space	Less privileged code is easier to analyze
Device drivers	Typically kernel-mode	Mostly user-space processes	Driver faults can be contained more easily
IPC model	Direct in-kernel calls	Message passing	Clearer mediation boundaries in microkernels
Fault impact	Can affect the whole kernel	Often localized to one service	Supports fault isolation
Formal verification	Very hard at full Linux scale	More feasible for small kernels	seL4 demonstrates this for a related microkernel
Effective TCB	Depends on policy and services	Depends on policy and trusted servers	TCB size is property-dependent, not a fixed number

"TCB" is not just a line count — a user-space file server may still need to be trusted for data integrity even outside kernel mode. Still, pulling services out of kernel mode genuinely shrinks the set of code with unrestricted, privileged access to everything else, a real structural difference regardless of how the count is sliced.

B. IPC as a Security Boundary, and a Driver Scenario

IPC in a microkernel also mediates what is permitted to talk to what. MINIX 3 performs well on latency-sensitive IPC (pipe latency), useful for frequent small control messages, while Ubuntu holds a large advantage on throughput-oriented IPC (pipe bandwidth) — which matters more is workload-dependent. Consider a vulnerable driver, a common attack vector: on a monolithic kernel a compromised kernel-mode driver already has full privilege, while on a microkernel a compromised user-space driver is more constrained to its own process and permissions — not risk-free, but less likely to escalate into a full kernel compromise.

Discussion

A. Isolation and Performance Are Not a Single Trade-Off

The results push back against the idea that "more isolation" and "less performance" move together along one simple line. Ubuntu is faster on system calls and memory bandwidth; MINIX 3 is faster on process creation, pipe latency, and context switching. Ubuntu's numbers suit high-throughput servers and data pipelines; MINIX 3's isolation-related numbers suit embedded, safety-critical, or teaching contexts where component separation and predictable switching matter more than raw throughput.

B. The Context-Switch Result Deserves a Second Look

The context-switch gap — 1.41 µs for MINIX 3 versus 51.65 µs for Ubuntu on two processes — is the sharpest in the dataset and aligns with what a microkernel's design goals would predict. Measuring it on matched bare-metal hardware across 30 repeated trials, rather than inside a hypervisor, addresses the main earlier caveat about virtualization artifacts inflating the gap; the formal significance test in Table 11 ($p < 0.001$, both parametric and non-parametric) confirms the gap is real rather than noise, though it remains a property of these two specific implementations rather than a general law about microkernels.

C. Memory Bandwidth Is Not a Security Metric

Ubuntu's large memory-bandwidth leads (roughly 17–26×) are genuine performance results but say nothing about security in either direction — LMbench measures data throughput, not vulnerability resistance. MINIX 3 keeping more services out of kernel mode is a separate, architectural point about reduced attack surface; the two observations do not substitute for one another.

D. Implications for Modern Secure Architectures, and Comparison with Prior Results

MINIX 3's core commitment — a small privileged kernel, services in user space, everything mediated through IPC — is the same commitment underlying seL4's formally verified design [5], [7], [8], FlexOS's tunable isolation boundaries [2], and the HongMeng production microkernel [11]. None of these were benchmarked directly here, and MINIX 3's numbers should not be read as predicting how they would perform; the value of the comparison is in what it says about the shape of the trade-off, not its magnitude for other systems.

Set against the wider literature, this shape becomes clearer. Tanenbaum's own MINIX-versus-Linux benchmarking [13] put the microkernel penalty at only 5–10%, far smaller than several gaps here, mainly showing how much prior comparisons vary with methodology — this paper's detailed documentation (Section IV) is a direct response. Elphinstone and Heiser's L4 retrospective [12] report sub-microsecond IPC in heavily optimized L4-family kernels, an order of magnitude below MINIX 3's pipe latency, suggesting its IPC overhead is substantially implementation-specific. HongMeng [11] was re-engineered specifically to close this kind of gap with Linux while keeping microkernel isolation — evidence that the ≈300× pipe-bandwidth gap measured here reflects MINIX 3's maturity, not a hard ceiling. FlexOS [2] adds that isolation cost is itself tunable, not fixed by the "microkernel" label. None of this lets the paper claim its gaps generalize to microkernels as a class (that needs directly benchmarking seL4 or HongMeng, Section XI); it supports the narrower claim that MINIX 3's position — competitive on context switching, behind on IPC latency and bandwidth — is a data point about MINIX 3 today, not microkernels in general.

E. Practical Takeaways

- High-throughput workloads (servers, databases, pipelines): Ubuntu's system-call speed and memory/pipe bandwidth make it the better fit.
- Isolation-focused systems (embedded, safety-critical, teaching): MINIX 3-style component separation and fast switching are attractive.
- Safety-critical production use: a formally verified microkernel such as seL4 is likely a better choice than MINIX 3 itself, which remains most useful for study and teaching.
- Hybrid designs: running Linux as a guest under a microkernel or separation kernel, the direction HongMeng [11] also explores, can combine a small trusted base with the Linux ecosystem.

Limitations and Threats to Validity

Several limitations are worth stating explicitly. The two test machines are physically separate units of the same hardware model rather than one machine dual-booting both systems; this removes hypervisor overhead but cannot guarantee every non-kernel hardware variable was identical between units, though the comparable arithmetic-operation results in Table 5 are consistent with adequate matching. MINIX 3 and Ubuntu also differ in implementation maturity, driver support, and toolchains unrelated to kernel architecture, so some portion of the observed differences is architectural and some reflects two different pieces of software built by different teams — this experiment cannot fully separate the two. Full statistical treatment (Tables 10–11) covers five core metrics with retained raw logs; the remaining metrics (fork/exec, memory bandwidth/latency, arithmetic operations) are reported only as means, and extending raw-log retention to them is the top item in Section XI. LMbench measures low-level primitives, not application-level behavior, so real workloads could show different patterns. The security discussion is architectural only — no penetration testing or fuzzing was performed on either system. Finally, these results describe MINIX 3.4.0 and Ubuntu 22.04.4 LTS specifically and should not be read as characterizing microkernels or monolithic kernels as categories; Section VIII-D already situates MINIX 3 alongside more optimized microkernels such as seL4 and HongMeng precisely because one implementation cannot stand in for an entire family.

A. Internal, External, and Construct Validity

Internally, the main concern is whether measured differences reflect kernel architecture rather than an uncontrolled variable; software/compiler versions and repetition counts are fully documented (Table 2) to rule out undisclosed-configuration confounds, though the two-machine design leaves a small residual risk of unit-to-unit hardware variation that only a shared-hardware (e.g., dual-boot) design could fully eliminate — the significance tests in Table 11 rule out measurement noise specifically, not this residual risk. Externally, these results generalize with reasonable confidence to MINIX 3.4.0 and Ubuntu 22.04.4 LTS on comparable Intel desktop hardware, and with much less confidence to other microkernels, distributions, CPU architectures, or server/embedded hardware; they should not be extrapolated to application-level performance without the workload-level experiments in Section XI. On construct validity, this paper measures exactly what LMbench measures — system-call, process-creation, IPC, context-switch, and memory metrics — not "performance" or "security" in any broader sense; Section VII's security discussion is architectural reasoning about TCB size and isolation, not an empirical security construct, and no result here should be read as a security metric.

Conclusion

This paper compared MINIX 3 and Ubuntu Linux on matched bare-metal hardware using LMbench, every measurement the mean of 30 repeated runs. Ubuntu led on system-call latency and all large-memory bandwidth tests (0.2616 µs null syscall; up to 12,341.97 MB/s reads), reflecting what a mature monolithic kernel is built to do well. MINIX 3 led on isolation-related measurements — 102.08 µs process fork, 135.81 µs process exec, 20.34 µs pipe latency, and 1.41 µs for a two-process context switch — obtained directly on physical hardware and confirmed significant ($p < 0.001$) for the five-core metrics with retained raw logs, giving more confidence than an earlier virtualization-based version of this comparison could support.

On security, the microkernel model's advantage is structural: pushing drivers and other services into user space reduces the amount of code running with full kernel privilege, aiding fault containment and analyzability — but the LMbench results do not themselves prove anything about security, which would require vulnerability analysis, attack testing, or formal methods outside this study's scope. The broader takeaway is that kernel architecture carries real, measurable consequences, and which choice makes sense depends on the workload: monolithic kernels remain strong where throughput is the priority, and microkernel-based systems are worth considering where isolation, service separation, and fault containment are what the system actually needs.

Future Work

- Bring seL4 or another formally verified microkernel into the comparison to separate what is specific to MINIX 3 from what generalizes across microkernel designs.
- Move beyond microbenchmarks to application-level workloads (web serving, databases, embedded control loops) to test whether these patterns hold under realistic load.
- Conduct empirical security testing (fuzzing, controlled vulnerability scenarios) to complement the architectural discussion with experimental evidence.
- Measure energy consumption, especially for context-switch-heavy workloads, since the latency differences observed here could translate into power differences too.
- Extend the raw-log-backed statistical treatment (Section VI-E) to the remaining metrics — process fork/exec, memory bandwidth/latency — and publish all 30-run raw logs in an open repository (GitHub or Zenodo) for independent verification.
- Add a third microkernel reference system (seL4 or Fiasco.OC are the most tractable) under the same protocol, and layer on an application-level or industry-standard suite (UnixBench, Phoronix, or a SPEC CPU subset) to test whether these patterns persist in realistic, mixed workloads.

References

[1] S. Y. Lim, S. Agrawal, X. Han, D. Eyers, D. O'Keeffe, and T. Pasquier, "Securing Monolithic Kernels using Compartmentalization," arXiv:2404.08716, 2024. https://arxiv.org/abs/2404.08716

[2] H. Lefeuvre, V.-A. Bădoiu, A. Jung, S. L. Teodorescu, S. Rauch, F. Huici, C. Raiciu, and P. Olivier, "FlexOS: Towards Flexible OS Isolation," in Proc. 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2022, pp. 467–482. https://doi.org/10.1145/3503222.3507759

[3] Y. Guo, Z. Wang, W. Bai, Q. Zeng, and K. Lu, "BULKHEAD: Secure, Scalable, and Efficient Kernel Compartmentalization with PKS," in Proc. Network and Distributed System Security Symposium (NDSS), 2025. https://www.ndss-symposium.org/

[4] P. Przymus, W. Weiner, K. Rykaczewski, and G. Kudrjavets, "Linux Kernel Recency Matters, CVE Severity Doesn't, and History Fades," in Proc. 23rd International Conference on Mining Software Repositories (MSR), 2026. https://conf.researchr.org/home/msr-2026
[5] E. de Matos and M. Ahvenjärvi, "seL4 Microkernel for Virtualization Use-Cases: Potential Directions towards a Standard VMM," Electronics, vol. 11, no. 24, Art. no. 4201, 2022. https://doi.org/10.3390/electronics11244201
[6] M. R. Rana and S. Baul, "An Overview of Operating Systems Based on Microkernel Technology and their Essential Components," International Journal of Information Engineering and Electronic Business, vol. 16, no. 6, pp. 18–28, 2024. https://doi.org/10.5815/ijieeb.2024.06.02
[7] L. Zhang, Y. Zhao, and J. Li, "A Comprehensive Specification and Verification of the L4 Microkernel API," in Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 14571, 2024, pp. 217–234. https://doi.org/10.1007/978-3-031-57249-4_11
[8] R. J. Colvin, I. J. Hayes, S. Heiner, P. Höfner, L. Meinicke, and R. C. Su, "Practical Rely/Guarantee Verification of an Efficient Lock for seL4 on Multicore Architectures," arXiv:2407.20559, 2024. https://arxiv.org/abs/2407.20559
[9] V. van Rijn and J. S. Rellermeyer, "A Fresh Look at the Architecture and Performance of Contemporary Isolation Platforms," in Proc. ACM/IFIP International Middleware Conference, 2021, pp. 1–13. https://doi.org/10.1145/3464298.3493398
[10] Y. Peng, H. Tian, J. Xian, S. Zhou, S. Yan, and Y. Zhang, "Framekernel: A Safe and Efficient Kernel Architecture via Rust-based Intra-kernel Privilege Separation," in Proc. ACM SIGOPS Asia-Pacific Workshop on Systems (APSys), 2024. https://doi.org/10.1145/3678015.3680488
[11] H. Chen, X. Miao, N. Jia, N. Wang, Y. Li, N. Liu, Y. Liu, F. Wang, Q. Huang, K. Li, H. Yang, H. Wang, J. Yin, Y. Peng, and F. Xu, "Microkernel Goes General: Performance and Compatibility in the HongMeng Production Microkernel," in Proc. 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2024, pp. 465–485. https://www.usenix.org/conference/osdi24/presentation/chen-haibo
[12] K. Elphinstone and G. Heiser, "From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels?," in Proc. 24th ACM Symposium on Operating Systems Principles (SOSP), 2013, pp. 133–150. https://doi.org/10.1145/2517349.2522720
[13] J. Corbet, "Comparing Linux and Minix," LWN.net, Feb. 2007. https://lwn.net/Articles/220255/