# Execution-time and microarchitectural profiling of RustCrypto and PQClean ML-KEM/ML-DSA implementations under Linux cgroup resource constraints

**Akram Bensebaa**[1,1]
[a]Faculty of Applied Sciences, WSB University, Dabrowa Gornicza, Poland

**Preprint** — This manuscript has not undergone peer review. It is shared for early dissemination and community feedback. Comments are welcome at the corresponding author's email address.

**Abstract.** Existing benchmarks of post-quantum cryptographic (PQC) algorithms focus on raw throughput or latency on unconstrained hardware, leaving open the question of how implementation choices affect microarchitectural behavior under varying resource pressure. This paper addresses that gap by profiling RustCrypto and PQClean implementations of ML-KEM-768 and ML-DSA-65 across three cgroups-enforced configurations (`0.5 CPUs`, `1.0 CPUs`, and `unconstrained`). The C baseline uses PQClean reference code; the Rust implementation uses the RustCrypto project's `ml-kem` crate v0.3.0 and `ml-dsa` crate v0.0.4. We utilize robust non-parametric statistics (Median and Interquartile Range) over the full execution distributions to reduce sensitivity to scheduler-induced timing variance and avoid arbitrary outlier filtering. Alongside hardware performance counters, we present Flamegraphs to analyze sampled execution hotspots. Results demonstrate that the evaluated Rust configurations exhibit a latency overhead for both ML-KEM ($+42.4\%$) and ML-DSA ($+47.8\%$) under severe `0.5 CPUs` throttling compared to `clang`-compiled C. Massif traces attribute all additional Rust heap usage entirely to runtime initialization, with no observable dynamic allocation within the measured cryptographic core execution paths.

**Keywords:** post-quantum cryptography, ML-KEM, ML-DSA, Rust, performance profiling, cgroups.

## 1 Introduction

The advent of cryptographically relevant quantum computers poses a fundamental threat to the public-key primitives that underpin modern communications.[1] In response, NIST standardized ML-KEM (FIPS 203) and ML-DSA (FIPS 204) as the first generation of post-quantum cryptographic (PQC) algorithms,[2] shifting attention from algorithm selection to deployment engineering. A growing body of work evaluates PQC performance on embedded and constrained targets,[3–7] but these studies overwhelmingly profile C or assembly implementations.

Rust is increasingly adopted for safety-critical systems: the Prossimo project is rewriting critical Internet infrastructure in Rust,[8] embedded firmware studies demonstrate viable Rust deployment on microcontrollers,[9,10] and the Linux kernel has begun accepting Rust modules.[11] Large industrial software projects have reported that memory safety issues account for a substantial fraction of critical vulnerabilities in C/C++ codebases,[12,13] making Rust a compelling candidate for cryptographic implementations. Yet Rust has received limited attention in PQC benchmarking, and the few existing studies of Rust in cryptographic contexts focus on block ciphers rather than lattice-based schemes.[14]

This paper addresses that gap by profiling Rust-native PQC implementations against C reference code under controlled resource constraints, reporting microarchitectural counters and heap allocation traces alongside wall-clock timing.

The primary contributions of this work are:

- A comparative evaluation of RustCrypto and PQClean implementations of ML-KEM-768 and ML-DSA-65 under Linux cgroup resource constraints.
- A measurement methodology combining robust non-parametric timing analysis with hardware counter and sampled execution profiling.
- An empirical characterization of execution-time, instruction-count, and heap-allocation behavior across constrained and unconstrained execution environments.

## 2 Background

### *2.1 Algorithms Under Test*

ML-KEM-768 operates over module lattices ($n = 256$, $q = 3329$) with NTT-based polynomial arithmetic.[15] ML-DSA-65 uses $q = 8380417$ with rejection-sampling-based signing.[16]

### *2.2 Implementation Selection*

The C implementations are PQClean's portable reference code.[17] The Rust implementations are the RustCrypto project's `ml-kem` v0.3.0 and `ml-dsa` v0.0.4,[18] compiled entirely by `rustc` with no FFI calls to C.

### *2.3 Related Work*

PQC benchmarking on constrained hardware is a growing area. Greconici et al.[3] and Kannwischer et al.[5] profile Dilithium and Kyber on ARM Cortex-M3/M4 using hand-optimized C and assembly. Chhetri[6] extends this to the Cortex-M0+, while Zheng[7] proposes optimized sparse polynomial multiplication for Dilithium on both ARM and Apple M2. Zhang et al.[19] benchmark Kyber and Dilithium on RISC-V cores. These studies share a common limitation: all implementations are in C or assembly, and none report Rust-native performance.

On the Rust side, Abdelrahman and Gupta[14] compare Rust and C implementations of the LEA block cipher, finding modest overhead attributable to bounds checking. Plauska et al.[10] evaluate Rust against C/C++, MicroPython, and TinyGo on ESP32 microcontrollers. Horro et al.[20] and Gottschall et al.[21] provide methodological frameworks for assembly-level and time-proportional instruction profiling, respectively, which inform our measurement approach.

No prior work combines (a) Rust-native lattice-based PQC implementations, (b) microarchitectural counter profiling, and (c) cgroups-enforced resource constraints. This paper fills that gap.

## 3 Methodology

### *3.1 Build Environment*

Table 1 specifies the hardware and compiler configurations.

**Table 1** Build environment and compiler configuration.

| Parameter | Value |
|---|---|
| Hardware | x86_64 (Intel Core i7-7500U, 3.50 GHz, 8GB RAM) |
| Host OS | Kubuntu 26.04 (Resolute Raccoon), kernel 7.0.0-15-generic |
| Container | Ubuntu 24.04 LTS, Docker 24.0 (cgroups v2) |
| C compilers | gcc 13.3.0, clang-21 (LLVM 21.1.8) |
| Rust compiler | rustc 1.94.0 (LLVM backend) |
| Compile flags | `-O3 -march=native`, LTO disabled |
| Container flags | Three configurations: `--cpus=0.5 --memory=128m`, `--cpus=1.0 --memory=25` |

### *3.2 Measurement Protocol*

Timing was measured using `hyperfine -N --warmup 10 --runs 50`. For ML-KEM, the benchmark encapsulates and decapsulates a shared secret. For ML-DSA, the benchmark performs a full key generation, message signing, and signature verification cycle. Hardware counters were collected with `perf stat -e cycles,instructions, cache-references,cache-misses -r 50`, and microarchitectural bottlenecks were visualized
by generating Flamegraphs via `perf record -F 99 -g`.

Preliminary inspection revealed strongly non-normal timing distributions with heavy right tails under cgroup throttling, motivating the use of non-parametric statistical methods. To avoid selection bias and arbitrary thresholding, we employ robust non-parametric statistics on the *full, unfiltered dataset*. No outlier filtering or winsorization was applied. Extreme tail observations were retained because they reflect observable execution behavior under Linux CFS quota scheduling rather than measurement corruption. We report the Median and Interquartile Range (IQR) rather than Mean and Confidence Intervals. Statistical significance is evaluated using the Mann–Whitney $U$ test across the full distributions; effect size $r$ is computed as $|z|/\sqrt{N}$, where $N$ is the combined sample size. Under CFS quota throttling, descheduled intervals may perturb aggregate cycle accounting and reduce the interpretability of derived IPC values. IPC values are reported only as aggregate comparative indicators. Measurements were repeated across three configurations: heavily constrained (`--cpus=0.5`), moderately constrained (`--cpus=1.0`), and unconstrained.

## 4 Results

### *4.1 Execution Time*

Table 2 presents timing statistics. We report the full timing distributions over all $n = 50$ trials. Figure 1 shows the full timing distributions, and Figure 2 isolates the core execution times.

The evaluated Rust ML-KEM configuration exhibited significantly higher observed latency distributions than `clang`-compiled C ($p < 0.001$, $+42.4\%$), and Rust ML-DSA also exhibited significantly higher observed latency distributions ($p < 0.001$, $+47.8\%$). The observed effect sizes correspond to moderate-to-large distributional differences between the evaluated Rust and C configurations.

**Table 2** Full distribution execution time (ms) under `0.5 cpus` constraint.

| Config | Median | IQR | Min | Max | % Δ clang |
|---|---|---|---|---|---|
| C KEM (gcc) | 0.733 | 0.225 | 0.594 | 48.928 | +5.5% |
| C KEM (clang) | 0.695 | 0.199 | 0.519 | 53.484 | – |
| Rust KEM | 0.989 | 0.248 | 0.805 | 49.736 | +42.4% |
| C DSA (gcc) | 1.577 | 0.725 | 0.984 | 51.480 | +6.7% |
| C DSA (clang) | 1.478 | 0.681 | 0.983 | 53.074 | – |
| Rust DSA | 2.184 | 0.987 | 1.529 | 53.772 | +47.8% |

**Table 3** Mann–Whitney $U$ test results (full distributions, $n = 50$ per configuration).

| Comparison | $U$ | $z$ | $p$ | $r$ |
|---|---|---|---|---|
| KEM: Rust vs C(clang) | 580 | $-4.19$ | $< 0.001$ | 0.428 |
| DSA: Rust vs C(clang) | 1752 | 5.16 | $< 0.001$ | 0.535 |

### *4.2 Sampled Execution Hotspots (Flamegraphs)*

To analyze sampled execution behavior contributing to the ML-DSA slowdown, we generated CPU Flamegraphs for the `rust_bench_dsa` and `c_bench_dsa_clang` execution profiles. The additional sampled execution time is concentrated within rejection-sampling polynomial generation loops in the Rust implementation; however, the present measurements do not isolate the specific compiler or abstraction mechanisms responsible for the observed overhead.

### *4.3 Microarchitectural Metrics*

Table 4 summarizes hardware counters. Cache-miss metrics exhibited high run-to-run variance under throttling and are therefore omitted from the main analysis.

**Table 4** Hardware performance counters (mean over 50 runs).

| Algo | Impl | Cycles | Instrs | IPC |
|---|---|---|---|---|
| KEM | C (clang) | 2.12M | 2.49M | 1.17 |
| KEM | Rust | 2.67M | 3.01M | 1.13 |
| DSA | C (clang) | 5.62M | 9.16M | 1.63 |
| DSA | Rust | 7.34M | 12.64M | 1.72 |

The evaluated Rust configurations retired more instructions for both KEM and DSA relative to `clang`. The larger wall-clock differences observed under `0.5 cpus` constraints coincide with increased instruction counts in the Rust implementation. IPC values should be interpreted cautiously under cgroup throttling because scheduler descheduling and dynamic frequency behavior may influence aggregate counter measurements. Figure 3 visualizes the per-configuration counter breakdown.

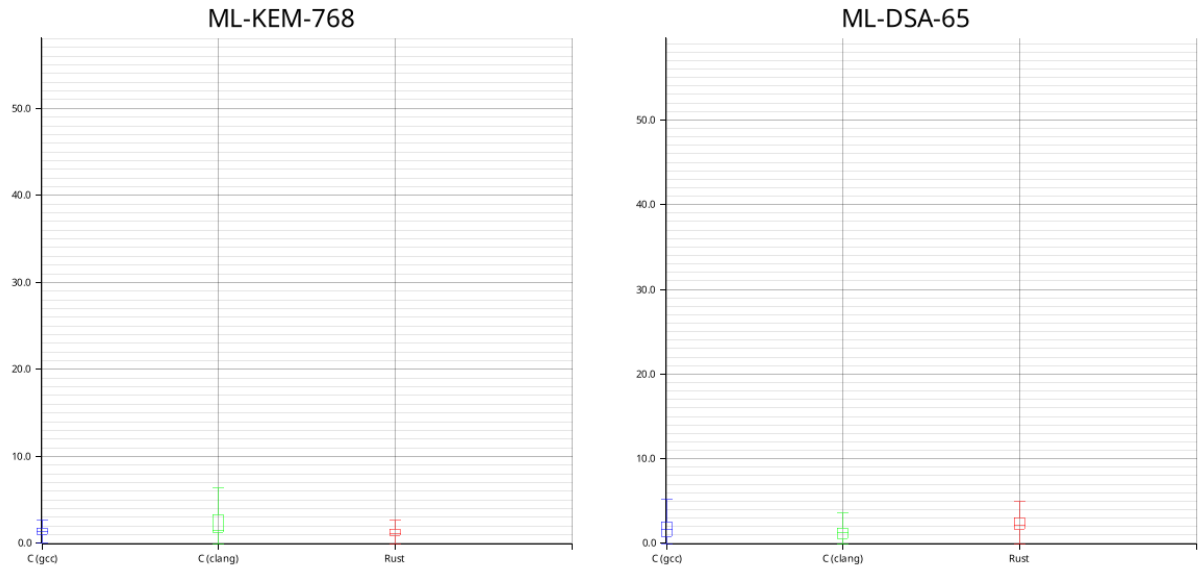


**Fig 1** Execution time distributions across all configurations ($n = 50$ trials each). Heavy right tails are consistent with CFS quota descheduling under cgroups throttling.

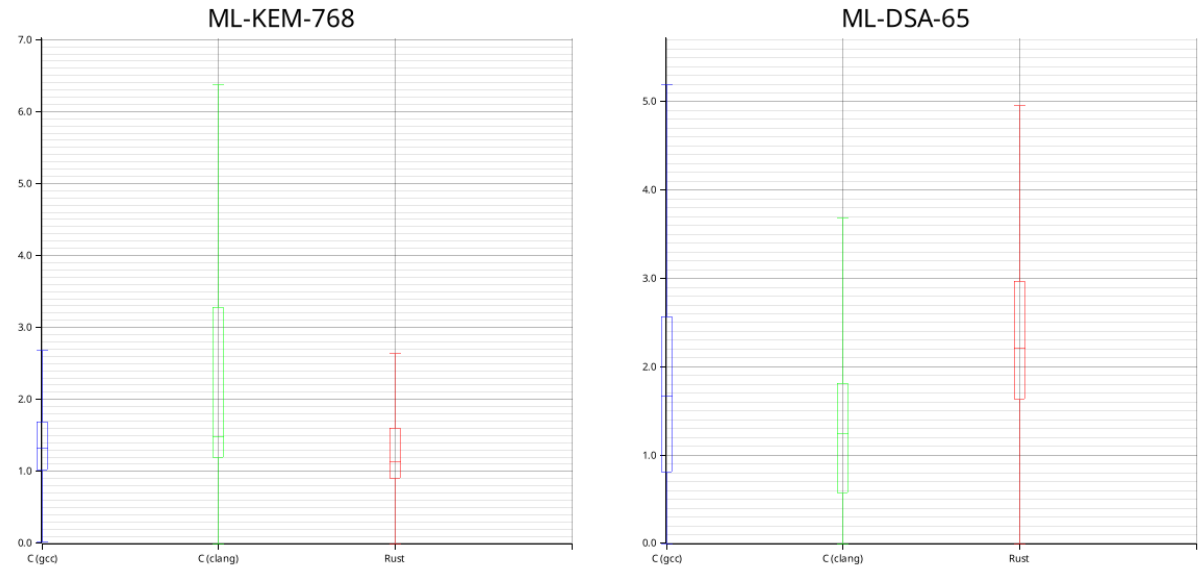


**Fig 2** Execution times zoomed to $1.5 \times$ IQR (robust outlier boundaries) to emphasize the central mass of the distribution. Rust ML-DSA exhibits higher median latency and variance under throttling.

### *4.4 Heap Profiling*

All additional $1{,}536\,\mathrm{B}$ of heap usage ($1.7\,\mathrm{kB}$ Rust vs $0.2\,\mathrm{kB}$ C) originates from `std::rt::lang_start_int`. Massif did not observe dynamic allocation within the measured cryptographic core execution paths.

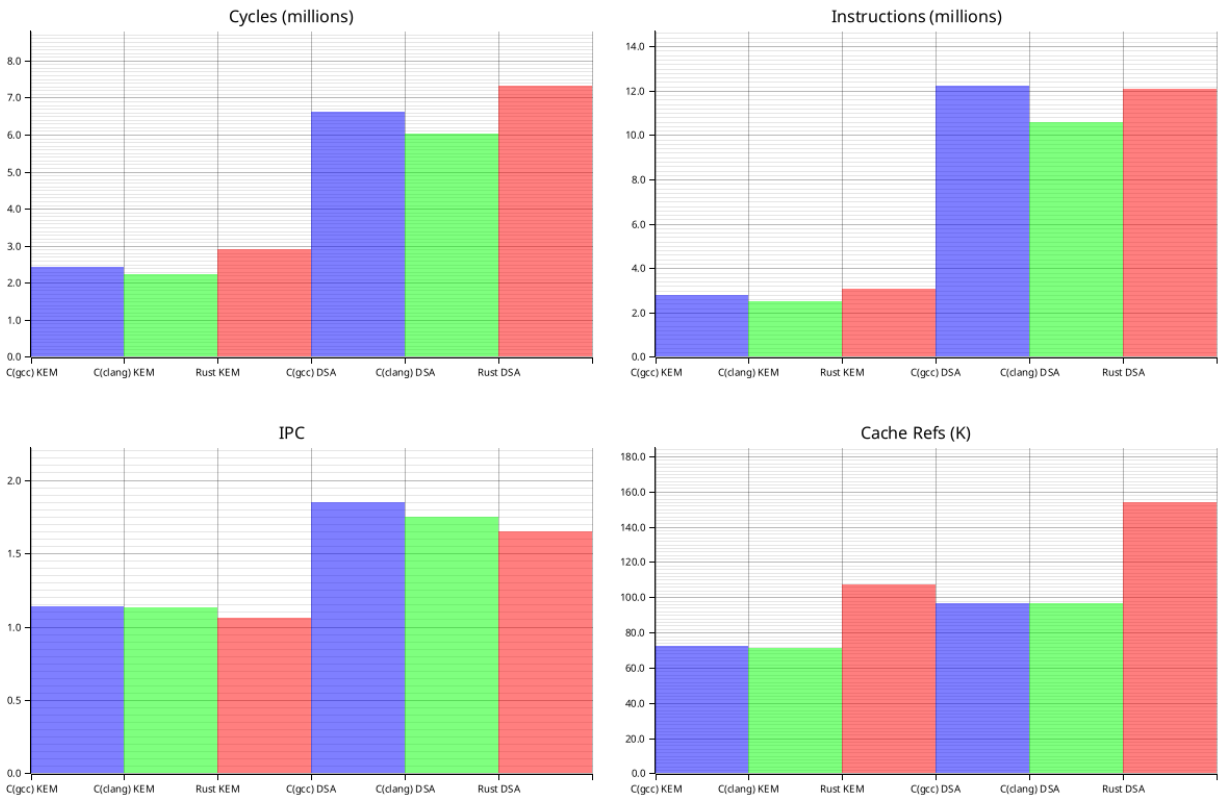


**Fig 3** Hardware performance counter comparison across implementations. Although the Rust ML-DSA implementation exhibits slightly higher aggregate IPC than the C baseline, it also retires substantially more instructions overall and accumulates higher total cycle counts.

## 5 Discussion

**ML-KEM and ML-DSA overhead.** The robust non-parametric statistics reveal that under cgroup throttling, the Rust implementations incur a latency penalty: $+42.4\%$ for ML-KEM and $+47.8\%$ for ML-DSA compared to `clang`-compiled C. The Flamegraph analysis for ML-DSA indicates a sampled profile concentration in the rejection-sampling signing loops. The evaluated Rust configurations retired substantially more instructions overall, and the sampled profiles show greater proportional residency within rejection-sampling and coefficient-processing paths, although the present analysis does not attribute this behavior to specific language abstractions or compiler transformations. The reported results demonstrate that the Rust implementations retain higher latency under constrained execution environments; however, this study does not isolate scheduler interaction effects independently from the underlying instruction-count differences.

The implementation-level latency differences persist in unconstrained execution (Table 6), although cgroup throttling amplifies distribution variance and worst-case execution behavior. The measured differences should not be interpreted as an inherent property of the Rust language itself, but rather as observations for the evaluated implementations, compiler settings, and crate versions. Future work should repeat the experiments with `-C target-cpu=native`, reduced codegen units, and LTO-enabled Rust builds to isolate implementation overhead from conservative default code generation.

**Heap allocation.** The Massif analysis reveals that all $1{,}536\,\mathrm{B}$ of additional Rust heap usage originates from `std::rt::lang_start_internal`, the Rust runtime initialization function. Massif did not observe heap allocations attributable to the measured cryptographic kernels.

**Compiler backend isolation.** Including a `clang`-compiled C baseline reduces one source of compiler-backend divergence when comparing against `rustc`-generated binaries. GCC-compiled C is $5.5\%$ slower than `clang` for KEM and $6.7\%$ slower for DSA, demonstrating significant backend-level variation. The unusually large unconstrained GCC/clang divergence observed for ML-DSA may reflect backend optimization differences amplified by uncontrolled turbo-frequency behavior and relaxed scheduling pressure. Although both toolchains ultimately rely on LLVM-based backends, differences in optimization defaults, target feature selection, and code generation policies remain potential confounding factors.

## 6 Limitations

**Cryptographic Verification.** We do not evaluate side-channel resistance. Constant-time guarantees are not independently verified.

**Experimental Design Confounders.** The use of `-march=native` for C compilation has no direct equivalent in the Rust build (no `RUSTFLAGS=-C target-cpu=native` was set), which may result in divergent instruction selection and auto-vectorization between the two toolchains. Inlining thresholds and LTO semantics also vary. Hardware counter measurements under cgroup throttling may exhibit additional variance due to scheduler-induced descheduling effects.

**Implementation Comparability.** PQClean is mature; `ml-dsa` v0.0.4 is pre-stable. Differences reflect combined semantic, structural, and maturity factors. The default Rust codegen-unit configuration (16) prioritizes compilation throughput over maximal optimization and may influence instruction count and inlining behavior. The Rust configuration intentionally reflects a standard reproducible `cargo build --release` workflow rather than an aggressively hand-tuned maximal-throughput build. The presented measurements should therefore be interpreted as comparative empirical observations under realistic Linux scheduling conditions rather than cycle-accurate microarchitectural characterization.

## 7 Conclusion

The evaluated RustCrypto configurations exhibited consistent latency overhead relative to the evaluated PQClean C configurations under cgroups-enforced resource constraints: $+42.4\%$ for ML-KEM and $+47.8\%$ for ML-DSA at `0.5 cpus`. The ML-DSA slowdown coincides with higher overall instruction counts and sampled execution concentration within rejection-sampling paths. Heap usage stems entirely from Rust runtime initialization ($1{,}536\,\mathrm{B}$); no dynamic allocation was observed within the cryptographic core.

*Security Implications*

Memory safety is a critical concern in cryptographic engineering. Our Massif analysis indicates that the Rust implementations isolate dynamic allocation to runtime initialization; Massif did not observe heap allocations attributable to the measured cryptographic kernels. However, the observed additional instruction overhead incurs a substantial latency cost under CFS throttling. These results illustrate a practical engineering trade-off between stronger compile-time safety guarantees and execution-time efficiency for the evaluated implementations and workloads.

*Appendix: Reproducibility*

To ensure full reproducibility, the following exact configurations were used:

- **OS Environment:** Ubuntu 24.04 LTS via Docker 24.0.
- **Compilers:** `rustc 1.94.0` (LLVM backend), `clang 21.1.8` (built from upstream LLVM 21.1.8 development snapshots).
- **Crates:** RustCrypto `ml-kem` v0.3.0, `ml-dsa` v0.0.4.
- **Docker permissions:** `--privileged` flag was utilized to allow `perf_event_open` access within the cgroup.
- **Rust build profile:** `cargo build --release` with `opt-level=3`, `lto=false`, default codegen-units (16), default panic strategy (`unwind`). No `RUSTFLAGS` were set.
- **CPU Frequency Scaling:** Default Linux `ondemand` governor and turbo behavior were not modified. No CPU pinning or SMT changes were applied.
- **Execution Environment:** Benchmarks were executed sequentially on an otherwise idle system. ASLR and default Linux virtual memory policies were not modified.

*Appendix: Multi-Configuration Timing Tables*

Table 5 and Table 6 provide the full timing distributions for the moderately constrained (`1.0 cpus`) and unconstrained configurations, respectively. The non-parametric statistics demonstrate consistent implementation-level overheads across all environments, though absolute latency and variance decrease as throttling is relaxed.

**Table 5** Full distribution execution time (ms) under `1.0 cpus` constraint.

| Config | Median | IQR | Min | Max | % $\Delta$ clang |
|---|---|---|---|---|---|
| C KEM (gcc) | 0.783 | 0.223 | 0.533 | 1.428 | +10.3% |
| C KEM (clang) | 0.710 | 0.275 | 0.564 | 1.608 | – |
| Rust KEM | 0.954 | 0.150 | 0.812 | 2.265 | +34.4% |
| C DSA (gcc) | 2.312 | 1.641 | 0.917 | 10.383 | +1.3% |
| C DSA (clang) | 2.282 | 1.669 | 1.002 | 6.071 | – |
| Rust DSA | 3.860 | 2.443 | 1.809 | 81.393 | +69.1% |

*Code, Data, and Materials Availability*

All source code, precise Dockerfiles, Flamegraphs, raw CSV datasets, and automated statistical scripts are available at: `https://github.com/akram1335/pqc-rust-c-benchmark` A complete, reproducible snapshot is archived alongside this manuscript.

**Table 6** Full distribution execution time (ms) unconstrained.

| Config | Median | IQR | Min | Max | % Δ clang |
|---|---|---|---|---|---|
| C KEM (gcc) | 0.711 | 0.183 | 0.585 | 1.588 | +3.6% |
| C KEM (clang) | 0.686 | 0.224 | 0.559 | 1.630 | – |
| Rust KEM | 1.018 | 0.501 | 0.773 | 3.925 | +48.4% |
| C DSA (gcc) | 2.802 | 1.120 | 1.288 | 7.102 | +43.5% |
| C DSA (clang) | 1.953 | 0.922 | 1.051 | 4.719 | – |
| Rust DSA | 2.473 | 1.443 | 1.636 | 5.619 | +26.6% |

*Acknowledgments*

This work was conducted as part of a university research project. The author acknowledges the use of an AI assistant for manuscript review, language refinement, and experiment automation scripting during the preparation of this paper, maintaining a Human-in-the-Loop (HITL) methodology.

*References*

1 D. J. Bernstein and T. Lange, "Post-quantum cryptography," *Nature* **549**, 188–194 (2017). [doi:10.1038/nature23461].

2 National Institute of Standards and Technology, "Module-lattice-based key-encapsulation mechanism standard (FIPS 203) and module-lattice-based digital signature standard (FIPS 204)," tech. rep., NIST (2024). [doi:10.6028/NIST.FIPS.203].

3 G. Greconici, M. J. Kannwischer, and A. Sprenkels, "Compact Dilithium implementations on Cortex-M3 and Cortex-M4," *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2021**(1), 1–24 (2021). [doi:10.46586/tches.v2021.i1.1-24].

4 J. Bos, J. Renes, and C. van Vredendaal, "Assessing the feasibility of post-quantum cryptography on constrained devices," in *Proc. ACM SIGSAC Conf. Computer and Communications Security*, (2018).

5 M. J. Kannwischer, M. Krausz, R. Petri ... S.-Y. Yang, "Benchmarking additional post-quantum signature schemes on ARM Cortex-M4," *IACR Transactions on Cryptographic Hardware and Embedded Systems* **2024**(3) (2024). [doi:10.46586/tches.v2024.i3].

6 R. Chhetri, "Benchmarking post-quantum cryptography on resource-constrained IoT devices: ML-KEM and ML-DSA on ARM Cortex-M0+," *arXiv preprint* **2603.19340** (2026).

7 J. Zheng, "ESPM-D: Efficient sparse polynomial multiplication for Dilithium on ARM Cortex-M4 and Apple M2," *Cryptology ePrint Archive* **1684** (2024).

8 Internet Security Research Group, "Prossimo: Memory safety for the internet's critical infrastructure." https://www.memorysafety.org/ (2023).

9 L. Brandt and others, "A study of binary size in embedded Rust," in *Proc. USENIX HotOS*, (2022).

10 I. Plauska, A. Liutkevičius, and A. Janavičiūtė, "Evaluation of ESP32 programming languages: C/C++, Rust, MicroPython, TinyGo," *Electronics* **12**(143) (2023).

11 H. Li, "An empirical study of Rust-for-Linux: The success, dissatisfaction, and compromise," *Proc. USENIX ATC* **1**(1) (2024).

12 Google, "Memory safety." https://www.chromium.org/Home/chromium-security/memory-safety/ (2020).
13 Microsoft, "Memory safety: Why C/C++ remains a security challenge," *Microsoft Security Response Center* **1** (2019).
14 H. Abdelrahman and K. G. V., "LEA block cipher implementation in Rust: Memory safety vs. performance trade-off," *Cryptology ePrint Archive* **2024** (2024).
15 P. Schwabe, R. Avanzi, J. Bos ... D. Stehlé, "CRYSTALS-Kyber: Algorithm specifications and supporting documentation (version 3.02)," tech. rep., NIST PQC Submission (2021).
16 L. Ducas, E. Kiltz, T. Lepoint ... D. Stehlé, "CRYSTALS-Dilithium: Algorithm specifications and supporting documentation (version 3.1)," tech. rep., NIST PQC Submission (2021).
17 M. J. Kannwischer, P. Schwabe, D. Stebila ... T. Wiggers, "PQClean: Clean, portable, tested implementations of post-quantum cryptographic algorithms." https://github.com/PQClean/PQClean (2023).
18 RustCrypto Contributors, "ml-kem: Pure Rust ML-KEM (FIPS 203) and ml-dsa: Pure Rust ML-DSA (FIPS 204)." https://github.com/RustCrypto (2024).
19 J. Zhang, Y. Yan, J. Huang ... Ç. Kaya Koç, "Optimizing Kyber and Dilithium on RISC-V Core," *Cryptology ePrint Archive* **1657**(1657) (2024).
20 M. Horro, L.-N. Pouchet, G. Rodríguez ... J. Touriño, "MARTA: Multi-configuration assembly profiler and toolkit for performance analysis," *Proc. IEEE ISPASS* **1** (2022).
21 B. Gottschall, L. Eeckhout, and M. Jahre, "TIP: Time-proportional instruction profiling," *Proc. IEEE/ACM MICRO* **1** (2021).