Evaluating Web-Based Post-Quantum Cryptography: A Statistical Analysis of ML-KEM-768 in Controlled Browser Environments

Samin Yasar
Independent Researcher
Dhaka, Bangladesh
research@samin-yasar.dev

May 2026

DOI: 10.5281/zenodo.20111815

Abstract

We present an empirical evaluation framework for browser-based post-quantum cryptography using FIPS 203 ML-KEM-768 in WebAssembly and JavaScript environments. Modern web platforms currently lack native PQC interfaces, requiring implementations to execute within managed browser runtimes whose JIT compilation, memory management, and feature-detection behavior complicate reliable performance analysis.

To investigate these effects, we develop StarryCrypt-PQC, an open-source hybrid ML-KEM-768 + X25519 implementation for controlled browser benchmarking. Using a dataset of $N = 462$ benchmark sessions collected across three browser engines, multiple runtime generations, and devices ranging from budget mobile hardware to desktop systems, we evaluate deployment-level performance characteristics of browser-based PQC execution.

Our measurements show that browser runtime generation strongly correlates with observed handshake latency, with Safari/WebKit and recent Chromium releases consistently outperforming older Chromium and Gecko-based runtimes. We additionally observe lower cryptographic-core latency for the WebAssembly implementation relative to pure JavaScript execution, although end-to-end comparisons remain influenced by substantial runtime and framework overhead.

Beyond raw measurements, the study identifies several methodological pitfalls affecting web-based PQC evaluation, including JIT warm-up instability, unreliable WebAssembly SIMD feature reporting, runtime-version confounding, and incomplete instrumentation of browser-managed overhead. We document these limitations explicitly and provide the accompanying implementation artifact and benchmark dataset to support future reproducible research on browser-native post-quantum deployment.

This work is intended as an empirical and methodological contribution focused on reproducible evaluation of browser-based PQC systems rather than a proposal of new cryptographic primitives or protocol designs.

Keywords. Post-Quantum Cryptography, ML-KEM, WebAssembly, JavaScript, Browser Performance, Hybrid Key Exchange

1 Introduction

Large-scale quantum computers, once built, will be able to break widely deployed public-key algorithms such as RSA and elliptic-curve Diffie-Hellman [Sho94]. In response, NIST final-

ized FIPS 203, standardizing the Module-Lattice-Based Key-Encapsulation Mechanism (ML-KEM) [Nat24a]. Although native deployments are progressing rapidly, web-based adoption lags behind. Because modern browser-native cryptographic interfaces, such as the Web Cryptography API [SW17], lack support for post-quantum primitives, researchers and developers must rely on JavaScript virtual machines and WebAssembly (WASM) runtimes [HRS+17]. In these environments, just-in-time (JIT) compilation and garbage collection (GC) introduce unpredictable latencies that complicate both performance measurement and side-channel analysis.

We present a systematic benchmarking methodology and a reproducible dataset for evaluating web-based post-quantum cryptography, with explicit documentation of measurement conditions and methodological limitations. This work is primarily an empirical and methodological contribution focused on reproducible evaluation of browser-based PQC deployments.

A natural assumption is that web PQC performance scales with device compute power: faster hardware yields lower latency. Through a systematic evaluation of $N = 462$ benchmark sessions (post-exclusion; 467 collected, $n = 3$ excluded for warm-up validation, $n = 2$ Chrome 87 outliers excluded) across three browser engines, 22 hardware configurations, and two implementation strategies (WASM and pure JavaScript), we present evidence that browser runtime vintage is the most robust performance predictor in our data, though this association is confounded with hardware distribution—budget-tier devices disproportionately run modern runtimes, preventing separation of browser effects from hardware effects (see §5.1). We observe a 2.99× mean latency reduction for WASM total-handshake latency compared to pure JavaScript (2.34 ms vs. 6.99 ms, with both Chrome 87 outliers excluded). We note that 87% of total latency is uninstrumented framework overhead, so this comparison conflates cryptographic core performance with framework integration costs. We foreground the browser-vintage observation as the primary finding (§5.1) and treat the WASM vs. JS comparison as a secondary descriptive observation. All subgroup analyses in this work are exploratory and should be interpreted as descriptive observations requiring independent replication.

Existing browser-based PQC evaluations often rely on ad-hoc benchmarking with small sample sizes, and many still target pre-finalization "Kyber" drafts rather than the finalized FIPS 203 standard. Few studies systematically compare WASM versus pure JavaScript across multiple browser engines and hardware tiers.

We present *StarryCrypt-PQC*, an open-source hybrid ML-KEM-768 + X25519 key exchange for the web, as a controlled experimental vehicle to investigate these questions. Our contributions are:

1. **Reproducible Benchmarking Framework for Web-Based PQC.** We develop a systematic evaluation methodology for browser-based post-quantum cryptography that addresses practical benchmarking challenges arising from JIT compilation, browser-managed memory behavior, and heterogeneous runtime environments.

2. **Cross-Engine Empirical Evaluation Dataset.** We provide a dataset of 462 benchmark sessions spanning WebKit, Blink, and Gecko runtimes across mobile and desktop hardware configurations. The dataset captures both controlled laboratory measurements and organically collected field sessions.

3. **Analysis of Runtime-Level Performance Behavior.** We present descriptive observations regarding browser-runtime influence on ML-KEM-768 execution latency, including cross-engine variance, runtime-generation effects, and deployment-specific behavior in managed browser environments.

4. **Documentation of Browser Measurement Pitfalls.** We identify several practical issues affecting browser-based cryptographic benchmarking, including unreliable SIMD capability reporting through `WebAssembly.validate()`, JIT warm-up instability, and the difficulty of separating cryptographic execution time from browser-managed framework overhead.

5. **Open-Source Implementation Artifact.** We release an open-source hybrid ML-KEM-768 + X25519 implementation and accompanying benchmarking harness to support reproducibility and future empirical research on browser-based post-quantum cryptography.

2 Background and Related Work

2.1 From Kyber to FIPS 203 ML-KEM

ML-KEM originates from the CRYSTALS-Kyber submission [ABD+21], which progressed through three NIST evaluation rounds before being standardized as FIPS 203 in August 2024. Alongside ML-KEM, NIST standardized ML-DSA (FIPS 204) for digital signatures [Nat24b], completing the primary post-quantum cryptographic suite. While much existing literature evaluates "Kyber Round 3," the finalized standard introduces small but security-critical changes. Failure to adopt these changes results in domain separation vulnerabilities, meaning that different cryptographic operations could be confused with one another.

Our implementation correctly applies the NIST-mandated modifications:

- **Hash Domain Separation:** Use of `0x06` for SHA-3 padding and `0x1F` for SHAKE, as specified by FIPS 202.
- **Dimension Parameter Concatenation:** During key generation, FIPS 203 requires the concatenation of the dimension parameter (k) prior to hashing the seed. Our implementation enforces this 33-byte hash input ($seed \parallel k$), a critical change from the 32-byte hash used in Round 3 implementations.

2.2 Hybrid Key Exchange: What We Follow from the Draft

To protect against the possibility that lattice-based cryptography could harbor undiscovered weaknesses, a hybrid approach combining PQC with well-studied elliptic-curve cryptography is recommended [BBF+19]. The IETF `draft-ietf-tls-ecdhe-mlkem-04` specification [KKWS26] defines three hybrid groups for TLS 1.3: X25519MLKEM768, SecP256r1MLKEM768, and SecP384r1MLKEM1024. Our implementation focuses on the X25519MLKEM768 construction and adopts the following from the draft:

- **Shared-secret concatenation order:** ML-KEM-768 SS ∥ X25519 SS (64 bytes total, 32 bytes each), matching §4.3 of the draft. **We explicitly note an interoperability trap**: the draft specifies this order for historical reasons unique to the X25519MLKEM768 group, which reverses the general hybrid naming convention (where the name would imply X25519 first, ML-KEM second). Implementers must carefully respect this historical inversion to maintain interoperability.
- **X25519 all-zero shared-secret check:** Per RFC 8446 §4.2.8 and the draft §4.3, we abort if the X25519 shared secret is all-zero (indicating an invalid peer public key or small-order point).
- **Byte-length validation:** Ciphertext and public-key lengths are validated against the FIPS 203 ML-KEM-768 parameters.

However, our code is a *general-purpose cryptographic wrapper*, not a TLS 1.3 stack. The following **are not implemented** because they are TLS-specific and outside our scope:

- TLS wire-format client/server share framing (e.g., the 1216-byte TLS client share, which is a composite structure concatenating the 32-byte X25519 public key and the 1184-byte ML-KEM encapsulation key).

- TLS 1.3 key schedule integration (RFC 8446).
- FIPS 203 §7.2 encapsulation-key validation at the *TLS framing boundary* (e.g., rejection of malformed encapsulation keys before they enter the TLS handshake state machine). At the *cryptographic boundary*, the underlying PQClean dependency strictly validates the 1184-byte ML-KEM encapsulation key length and enforces the modulus check ($x < 3329$) during deserialization; only the TLS-specific framing-layer validation is omitted.
- TLS alert handling (`illegal_parameter`, `internal_error`).

For application-level key derivation, we apply HKDF-SHA-256 over the 64-byte concatenation. We provide a security rationale (§3.3) informed by existing hybrid KEM composition literature, establishing that this construction preserves IND-CCA2 security under standard assumptions, provided the info string is globally unique per application context. The construction applies HKDF with a zero-length salt and a context-specific info string for two purposes: (1) **domain separation**, ensuring the output key material is bound to the specific application context, and (2) **output length control**, deriving a uniform 32-byte symmetric key regardless of IKM structure. **We note that the zero-length salt does not inject cryptographic entropy** (RFC 5869 §3.1); when the salt is empty, HKDF-Extract reduces to HMAC-SHA-256($0\text{x}3636\ldots36_{64}$, IKM)—a well-defined PRF application under the HMAC security proof. As shown by Krawczyk [Kra10], HKDF extraction security does not depend on salt secrecy, randomness, or even presence—the pseudorandomness of the extracted key derives from the min-entropy of the IKM. **Critical deployment requirement:** Because the salt is zero-length (identical across all invocations), key separation across different applications or protocol instances relies *entirely* on the info string being distinct. If two applications use this library with an identical info string, they will derive identical key material, enabling cross-protocol key-reuse attacks. Users of this library MUST select a globally unique info string per application context (e.g., `''MyApp-v1-MLKEM768-X25519-SessionKey''`). The default info string (`''Starrycrypt-PQC v1 | X25519MLKEM768 | AES-256-GCM''`) is intended for benchmarking only and MUST be overridden in production deployments. This HKDF step is an application-specific convenience wrapper for non-TLS use cases; it is **not part of the TLS 1.3 draft**, which feeds the raw 64-byte concatenation directly into the standard TLS 1.3 key schedule [Res18]:

$$SS_{\text{final}} = \text{HKDF-SHA-256}(\text{salt} = \emptyset, \text{IKM} = SS_{\text{ML-KEM}} \parallel SS_{\text{X25519}}, \text{info} = \text{ctx}) \tag{1}$$

By the security rationale in §3.3, this construction preserves IND-CCA2 security under standard assumptions: an attacker must break *both* ML-KEM-768 and X25519 to compromise the derived key, subject to the info-string uniqueness requirement.

2.3 Related Work

Browser PQC Benchmarking. Prior browser-based PQC evaluations include Bernstein et al.'s JavaScript NTRU Prime implementation [BCLvV18] (measuring polyfills for lattice operations, but pre-WASM and limited to Firefox). Other studies exist as grey literature (blog posts, GitHub repositories) without peer-reviewed citable artifacts; we provide URLs for these in the artifact repository [Yas26]. These works demonstrated feasibility but did not address JIT warm-up artifacts, cross-engine variance, or FIPS 203 compliance—and typically reported results from fewer than 50 total sessions. Our work scales to 462 sessions across three browser engines and uses a systematic time-based warm-up protocol with post-hoc validation for tier-up completion.

Hybrid Key Exchange Standards. Bindel et al. [BBF+19] formalized the security model for hybrid key encapsulation. The IETF has since advanced concrete specifications, including the general hybrid design framework [SFG26] and the concrete `draft-ietf-tls-ecdhe-mlkem-04` specification [KKWS26]. Our implementation follows the draft's shared-secret concatenation order and X25519 validation checks, applied in an application-level (non-TLS) context.

3 Implementation and Security Architecture

Implementing cryptographic primitives in WASM requires careful engineering to overcome the lack of direct memory management and constant-time execution guarantees in web browsers.

3.1 Defense-in-Depth Memory Zeroization

WebAssembly executes within a linear memory space that JavaScript can read. If cryptographic secrets (e.g., private keys, decapsulated shared secrets) are not explicitly erased after use, they remain vulnerable to cross-site scripting (XSS) or memory-dump attacks.

We implemented a two-tier zeroization strategy:

1. **C/WASM Layer:** An explicit `mlkem_zeroize` function uses a `volatile` pointer cast to prevent dead-store elimination—a compiler optimization that removes writes it considers unnecessary—from deleting the `memset` operation before the memory is freed.

2. **JavaScript Layer:** A strict wrapper protocol ensures all `Uint8Array` views that expose secret data call `.fill(0)` immediately after the hybrid session key is derived.

   SECURITY WARNING—Memory zeroization is NOT guaranteed in WASM/JS runtimes. This library is **unsuitable for protecting long-lived secrets** (e.g., private keys persisted in browser state such as `localStorage`, `IndexedDB`, or session storage). Three independent mechanisms can defeat zeroization in managed browser environments:

   1. **JIT optimization ignores C-level `volatile`:** While the `volatile` cast prevents dead-store elimination by the Emscripten C compiler, JIT compilers in WASM runtimes (e.g., V8's TurboFan, SpiderMonkey's IonMonkey) perform subsequent optimization passes that do not preserve C-level `volatile` semantics. A JIT compiler may determine that a buffer is never read again and eliminate the zeroizing write, **leaving secret material in accessible WASM linear memory**.
   2. **Engine-internal buffer copies:** JavaScript engines may create internal copies of buffer contents during type conversions, garbage collection, or when marshalling data across the WASM/JS boundary. These engine-internal copies are **not accessible for explicit zeroization** and may persist until garbage collection reclaims them—at which point the memory may be reused without being zeroed.
   3. **WASM linear memory visibility:** The WASM linear memory space is readable from JavaScript via `WebAssembly.Memory.buffer`. Any secret that survives zeroization in this memory space is accessible to any JavaScript code running in the same origin, including code injected via XSS.

   A complete audit would require heap snapshots or engine-level instrumentation to verify that no un-zeroed copies persist after garbage collection. **No such audit has been performed for this library.** We therefore characterize our zeroization as a *best-effort defense-in-depth* measure, not a formal guarantee of secret erasure in managed runtimes. **Deployment recommendation:** This library should be used only for ephemeral key exchange (e.g., single-session TLS-style handshakes where secrets are derived and consumed within milliseconds). It must **not** be used to protect long-lived secrets in browser state, where the window of exposure extends from milliseconds to hours or days.

3.2 PQClean Dependency and WASM Bindings

Our ML-KEM-768 implementation is derived from the PQClean project [PQC26], a collection of post-quantum cryptographic implementations maintained by the PQCrypto community. This section documents the provenance, modifications, and WASM integration for reproducibility.

PQClean version and provenance. The core cryptographic routines (KeyGen, Encaps, Decaps) are adapted from PQClean's `ml-kem-768/clean` implementation, corresponding to PQClean commit `bea734c` (April 2026). The PQClean implementation itself traces to the CRYSTALS-Kyber reference code [ABD+21] with modifications for FIPS 203 compliance. We note that PQClean is a community-maintained aggregation of reference implementations; it is not a NIST-endorsed distribution. The code carries the following licenses:

- **ML-KEM core:** Public domain / CC0 (CRYSTALS-Kyber authors)
- **SHA-3/SHAKE (fips202.c):** Public domain / CC0
- **Randombytes (randombytes.c):** MIT License (Daan Sprenkels, 2017)

Modifications from PQClean baseline. We made the following modifications to the PQClean reference code:

1. **WASM export layer:** Added `wasm_export.c` providing Emscripten-specific bindings with `EMSCRIPTEN_KEEPALIVE` annotations for the following functions: `mlkem_malloc`, `mlkem_free`, `mlkem_zeroize`, `mlkem_keypair`, `mlkem_enc`, `mlkem_dec`, `sha3_256_wasm`, `sha3_512_wasm`, `shake256_wasm`.
2. **Emscripten random source:** Modified `randombytes.c` to use `EM_ASM_INT` for browser `crypto.getRandomValues()` integration when compiled with `__EMSCRIPTEN__` defined. The original PQClean implementation supports Linux, BSD, Windows, and WASI but not Emscripten's JavaScript runtime.
3. **Memory allocation interface:** Added explicit `mlkem_malloc`/`mlkem_free` wrappers to expose heap management to JavaScript, enabling pre-allocated buffers for performance-sensitive applications.
4. **No algorithmic modifications:** The core ML-KEM arithmetic (NTT, polynomial operations, CBD sampling, Fujisaki-Okamoto transform) is unchanged from PQClean. All FIPS 203 compliance properties (hash domain separation, dimension parameter concatenation) are inherited from the upstream implementation.

Emscripten compilation. The WASM module is compiled with Emscripten v3.1.52 using the following flags:

- `-O3`: Aggressive optimization for throughput
- `-s WASM=1`: WebAssembly output (not asm.js fallback)
- `-s MALLOC=emmalloc`: Emscripten's minimal memory allocator
- `-s ALLOW_MEMORY_GROWTH=1`: Dynamic heap expansion
- `-s MODULARIZE=1 -s EXPORT_NAME="MLKEMModule"`: ES6 module wrapper
- **Explicitly excluded:** `-msimd128` (SIMD disabled; see §5.3 for detection reliability issues)

Known upstream security issues. As of this writing, we are not aware of any published security vulnerabilities in PQClean's ML-KEM-768 implementation. However, we note the following caveats:

- PQClean's "clean" implementation prioritizes readability over constant-time hardening. It uses algorithmic constant-time operations (e.g., Barrett reduction, value-barrier patterns) but does not employ hardware-specific timing defenses (e.g., cache preloading, DPA-resistant scalar multiplication).
- The implementation has not undergone formal verification or third-party security audit. NIST's FIPS 203 standardization process evaluated the ML-KEM algorithm, not specific implementations.
- Browser environments introduce additional attack surfaces (JIT-induced timing variance, garbage collection pauses, degraded timer precision) that are not addressed by the PQClean codebase.

Reproducibility. The complete source code, build scripts, and compilation environment are available in the artifact repository. The WASM binary can be reproduced by installing Emscripten v3.1.52 and running `make all` with the exact flags listed above. We recommend independent compilation and comparison of the resulting WASM binary hash before production deployment.

3.3 Security Analysis of the Hybrid Construction

We now provide a formal security argument for the hybrid construction defined in Equation 1. Our analysis establishes that the construction preserves IND-CCA2 security under standard cryptographic assumptions, subject to a deployment discipline requirement on info-string uniqueness.

Security Rationale for the Hybrid Construction Rationale. Assuming the security of ML-KEM-768 and the hardness assumptions underlying X25519-based ephemeral Diffie–Hellman exchange, and provided the info string is globally unique per application context, the output SS_{final} of Equation 1 is computationally indistinguishable from a uniform 256-bit string. This rationale is informed by existing hybrid KEM composition literature; it is not a new formal proof.

Proof sketch. The argument proceeds in three steps.

Step 1: Min-entropy of the concatenated shared secret. By the hybrid KEM security composition result of Bindel et al. [BBF+19], the concatenation $SS_{\text{ML-KEM}} \| SS_{\text{X25519}}$ preserves IND-CCA2 security under the assumption that at least one of the two component schemes remains secure. In particular, any PPT adversary that cannot break both KEMs simultaneously cannot distinguish the concatenated shared secret from a uniform 512-bit string. This implies that the IKM input to HKDF has min-entropy ≥ 256 bits under the stated assumption.

Step 2: HKDF extraction with a zero-length salt preserves pseudorandomness. Krawczyk [Kra10] proves that the HKDF extraction step PRK = HMAC-SHA-256(salt, IKM) produces a pseudorandom key as long as the IKM has sufficient min-entropy, *regardless of whether the salt is fixed, public, random, or even zero-length.* When the salt is empty, HKDF-Extract reduces to HMAC-SHA-256($0\text{x}3636\ldots36_{64}$, IKM)—a well-defined PRF application under the HMAC security proof. Krawczyk explicitly analyzes this case (Theorem 1 and §3 of [Kra10]): the pseudorandomness of PRK derives from the min-entropy of the IKM, not from the salt. This directly addresses the concern that the zero-length salt "offers no entropy isolation": entropy isolation is provided by the IKM itself (a 512-bit concatenation of two independent KEM shared secrets), not by the salt.

Step 3: HKDF expansion produces pseudorandom output. The expansion step OKM = HMAC-SHA-256(PRK, info$\|$0x01) is a PRF application under the standard HMAC-SHA-256 PRF assumption. Given that PRK is pseudorandom (from Step 2), the output is

computationally indistinguishable from uniform as long as the info string is unique per application context, ensuring domain separation across different invocations.

Composing Steps 1–3, SS_{final} is computationally indistinguishable from a uniform 256-bit string under the stated assumptions. □

Comparison with TLS 1.3 key schedule. The TLS 1.3 key schedule (RFC 8446 [Res18]) applies HKDF-Extract with a *zero-length* salt for the initial early secret derivation—the **identical** setup to our construction. TLS 1.3 then relies entirely on context-specific labels (analogous to our info strings) for key separation across the handshake. Our construction follows the same design pattern: a zero-length salt for extraction, with domain separation delegated to the info/label parameter. The TLS 1.3 key schedule has been formally analyzed within the TLS 1.3 security proofs [Res18], and this analysis confirms that HKDF extraction security does not depend on salt secrecy or randomness.

Deviation from Bindel et al.'s composition. Bindel et al. [BBF+19] define hybrid KEM composition at the KEM level, producing a combined KEM with its own encapsulation/decapsulation interface. Our construction composes at the shared-secret level and then applies HKDF for key derivation. This is a different composition pattern, but it can be analyzed within the same security framework. The key difference is that Bindel et al.'s "dual" combiner applies a key derivation function to the concatenation of shared secrets; our HKDF step serves exactly this role. The concatenation+HKDF approach is at least as strong as the XOR combiner (concatenation preserves strictly more information than XOR), and the HKDF step provides pseudorandom output under standard assumptions.

Info-string uniqueness as a deployment discipline requirement. The security rationale in §3.3 assumes that the info string is globally unique per application context. This is a *deployment discipline requirement*, not a cryptographic weakness of the construction. It is analogous to the requirement in TLS 1.3 that transcript hashes be distinct across different handshake sessions—a property guaranteed by the protocol design rather than by the cryptographic primitive itself. **If two applications use this library with an identical info string, they will derive identical key material, enabling cross-protocol key-reuse attacks.** Users of this library MUST select a globally unique info string per application context (e.g., `''MyApp-v1-MLKEM768-X25519-SessionKey''`). The default info string (`''Starrycrypt-PQC v1 | X25519MLKEM768 | AES-256-GCM''`) is intended for benchmarking only and MUST be overridden in production deployments.

Limitations of this analysis. We emphasize three limitations: (1) This is a proof sketch, not a fully formalized, machine-checked proof. A complete formalization would require instantiating the security definitions of Bindel et al. [BBF+19] and Krawczyk [Kra10] in a unified framework and verifying the composition step-by-step. (2) The analysis assumes the standard HMAC-SHA-256 PRF assumption and the IND-CCA2 security of at least one component KEM; if both are broken simultaneously, the construction provides no security. (3) The info-string uniqueness requirement is a deployment discipline constraint that cannot be enforced by the cryptographic construction itself. Despite these limitations, the analysis provides a sound security argument under standard assumptions and clearly delineates the boundary between cryptographic guarantees and deployment responsibilities. We recommend independent cryptanalysis before production deployment.

4 Benchmarking Methodology

Evaluating performance inside a JavaScript virtual machine requires isolating the cryptographic algorithm from the execution environment's overhead. We addressed several methodological flaws common in previous web cryptography studies.

4.1 JIT Compilation Isolation

Browsers execute WASM and JS through tiered JIT compilers. In V8, code begins execution in the Ignition interpreter; once a function exceeds a "hotness" threshold, it is promoted to the TurboFan optimizing compiler for native machine code generation [V8 26]. Safari's JavaScriptCore and Firefox's SpiderMonkey employ analogous tiering strategies (B3/Air and IonMonkey, respectively). The first several invocations thus run in an unoptimized interpreter state, producing latency spikes that do not reflect steady-state production performance.

To isolate algorithmic throughput, we enforce a JIT warm-up phase. While V8's TurboFan typically promotes hot functions after 5–10 invocations, SpiderMonkey's IonMonkey and JavaScriptCore's B3/Air may require 1,000+ iterations for full tier-up [Moz26, Piz16]. Rather than engine-specific tuning, we use a **time-based warm-up**: the benchmark runs dummy iterations for a fixed 200 ms duration before collecting measurements. This duration was selected empirically: across 10 pilot runs per engine, we observed coefficient-of-variation (CV) stabilization below 5% after 150–180 ms on all three engines. The 200 ms threshold provides a safety margin while keeping total benchmark time under 1 second. Per-engine warm-up effectiveness is validated post-hoc: sessions where the first timed iteration exceeded 150% of the session median were flagged as potentially under-warmed and excluded from analysis ($n = 3$ exclusions, 0.6% of total).

4.2 Garbage Collection and Timer Precision

Random garbage collection (GC) pauses can artificially inflate maximum latency measurements. To mitigate this:

- We use high-resolution `performance.now()` timers strictly scoped to the WASM execution call (`ccall`), excluding JS-to-WASM memory-allocation overhead.
- We report interpercentile ranges (P5–P95) rather than relying solely on min/max boundaries, providing a robust statistical view that naturally filters brief GC pauses.

4.3 Dataset Composition

Our dataset comprises $N = 462$ total benchmark sessions after quality exclusions (Table 1); 467 sessions were initially collected, of which $n = 3$ were excluded for warm-up validation failure (see §4.1) and $n = 2$ Chrome 87 outliers (one WASM, one JS) were formally excluded from the legacy macOS environment, yielding the final $N = 462$.

- **Lab / Synthetic ($N = 307$, 67%):** Automated runs via BrowserStack and local headless scripts across 63 device/OS/browser configurations spanning 53 unique device models, with ground-truth hardware metadata (logical cores, RAM, SIMD flags). While the testing environment is structured, the configurations represent a diverse, historically broad cross-section of browser versions rather than a strictly controlled monolithic platform. Lab sessions are unevenly split between WASM ($n = 153$) and pure JS ($n = 154$).
- **Field / Organic ($N = 155$, 33%):** Voluntary sessions from a publicly deployed test page, accessed primarily from Bangladesh and surrounding regions, with devices self-identifying via browser user-agent. Field data includes $n = 87$ WASM and $n = 68$ pure JS sessions.

Because the dataset is lab-dominant (67% controlled), our performance findings primarily reflect reproducible, controlled conditions rather than fully organic real-world variance. Field sessions (33%) provide a sanity check that lab findings transfer to uncontrolled browser states (background tabs, thermal conditions, variable network quality).

Table 1: Dataset Composition

Source	Impl.	n	Engines	Configs[a]
Lab (BrowserStack)	WASM	154	3	63
	Pure JS	155	3	63
Field (Organic)	WASM	87	3	—
	Pure JS	68	3	—
Total	—	**462**	**3**	**63**

[a]Unique device/OS/browser combinations in synthetic lab runs ($n = 63$).

5 Performance Evaluation

Reader Advisory: All quantitative performance comparisons in this section are exploratory and descriptive. The findings should be interpreted as hypothesis-generating observations requiring independent replication, not as definitive performance claims.

5.1 Browser Runtime Vintage as a Performance Factor (Primary Descriptive Finding)

Why this is the primary finding. The browser-vintage stratification below is the most robust observation because it emerges from a coherent monotonic ordering across five strata—a pattern that is unlikely under the global null hypothesis even without formal p-value correction. This contrasts with the WASM vs. JS comparison (§5.2), which is treated as a secondary descriptive observation. **Critical confounding:** Browser vintage and hardware tier are not independently varied in our dataset. As Table 3 shows, budget-tier devices run 98.8% modern runtimes while mid-range devices contain a higher concentration of older engines and Firefox. This collinearity prevents attribution of performance differences to browser effects versus hardware effects; the coherent vintage ordering and the non-monotonic MIPS-tier result both reflect this confounding.

Because browser runtime optimization pipelines evolve independently of hardware raw throughput, we stratify sessions first by browser engine family and, for Chromium-based browsers, by major version tier—a direct operationalization of the browser-vintage hypothesis (Table 2). Safari/WebKit achieves the lowest observed latency (0.67 ms, $n = 18$), followed by Modern Chromium 131+ (1.68 ms, $n = 117$), Mature Chromium 101–130 (2.87 ms, $n = 18$), Embedded/Other browsers (2.54 ms, $n = 66$), and Firefox/Gecko (6.92 ms, $n = 19$). This ordering is consistent with the hypothesis that runtime optimization influences web PQC performance, but the small sample sizes within several strata (e.g., $n = 18$ for Safari/WebKit, $n = 19$ for Firefox/Gecko) preclude formal statistical comparison.

JS-MIPS tier classification and browser–hardware confounding. As a secondary analysis, we also classified sessions by baseline JS MIPS: Budget (< 150, $n = 77$), Mid-Range (150–400, $n = 129$), Flagship (≥ 400, $n = 35$).

MIPS Measurement Methodology and Limitations. Baseline JS MIPS is measured via a synthetic in-browser benchmark executing 10^6 iterations of a deterministic 32-bit xorshift PRNG in a tight JavaScript loop, timed with `performance.now()`. The result represents millions of simple integer operations per second (Mops/s), reported as "JS-MIPS." **Critical caveats**: This synthetic microbenchmark is susceptible to JavaScript engine optimizations including loop unrolling, invariant code motion, and branch elimination (e.g., V8 TurboFan's range analysis may constant-fold parts of the PRNG state). Consequently, JS-MIPS scores are *not comparable across browser engines*—they reflect engine-specific optimization aggressiveness as much as underlying hardware capability. We further acknowledge that MIPS is a notoriously poor proxy for

modern heterogeneous CPU performance: it captures only single-threaded ALU throughput, ignoring memory hierarchy, SIMD width, thermal constraints, and compiler back-end differences. Our use of JS-MIPS is strictly for coarse device stratification within the *same* engine family, not as a precise performance predictor or for cross-engine comparison. **The non-monotonic Budget < Mid-Range result reflects confounding between browser vintage and hardware tier**: Budget-tier devices disproportionately run modern browser runtimes (98.8% modern Chromium or Safari), while the Mid-Range tier contains a higher concentration of older engines and Firefox, preventing attribution of the latency inversion to either factor alone. A purely compute-centric model would predict monotonically decreasing latency from Budget to Flagship. Instead, our data shows a non-monotonic result: Budget-tier WASM mean latency (3.01 ms) was lower than Mid-Range (4.68 ms). This inversion does not demonstrate that MIPS is a poor proxy for compute power per se; rather, it reveals that browser vintage and hardware tier are confounded in our dataset. The Budget tier contains a disproportionate share of modern mobile Chromium and Safari (98.8% modern runtimes per Table 3), while the Mid-Range tier includes older x86 desktops with mature Chromium and Firefox. Because these two variables co-vary, we cannot attribute the observed latency differences to browser effects alone or to hardware effects alone. The correct conclusion is that browser vintage and hardware tier are inseparable in this dataset, and neither the coherent vintage ordering nor the non-monotonic MIPS result can be interpreted as evidence for one factor dominating the other. Table 3 illustrates this distribution.

Table 2: Browser-vintage stratification of WASM sessions (Chrome 87 outlier excluded from primary statistics; see Table 5). Chromium-family comprises Chrome, Edge, Brave, and Opera; Embedded/Other includes Android WebView, Samsung Internet, and browsers with unparseable version strings.

Browser Vintage Tier	n	**Mean (ms)**	**95% CI (ms)**
Safari/WebKit	18	0.67	[0.49, 0.85]
Modern Chromium (131+)	117	1.68	[1.39, 1.97]
Mature Chromium (101–130)	18	2.87	[1.52, 4.21]
Embedded/Other	66	2.54	[1.76, 3.32]
Firefox/Gecko	19	6.92	[5.15, 8.69]

Table 3: Browser Vintage Distribution by Hardware Tier

Hardware Tier	**Dominant Browser Configurations**	**Modern Runtimes**
Budget	Chrome Mobile 140+, Safari iOS 18	98.8%
Mid-Range	Chrome 140+ Desktop, Safari 15–18	96.2%
Flagship	Safari 18+ (Apple Silicon), Chrome	100.0%

Legacy runtimes defined as Chrome $\leq$ 100 (pre-2022) or Safari $\leq$ 14 (pre-2021). Safari 15–16 (2021–2022) retains modern WASM

5.2 Exploratory WASM vs. JavaScript Performance Comparison

Statistical significance and effect size. A Welch's t-test comparing WASM and pure JavaScript total-handshake latency (with the Chrome 87 outlier excluded) yields $t = -4.74$, $p < 0.0001$—a highly significant result with a medium effect size (Cohen's $d = 0.46$). A post-hoc power analysis reveals that the test achieves $> 99\%$ power to detect this effect size at $\alpha = 0.05$, well above the conventional 80% threshold, given our current sample sizes ($n_{\text{WASM}} = 240$, $n_{\text{JS}} = 223$).

Multiple comparisons problem. This paper conducts the following family of subgroup analyses, each constituting an implicit hypothesis test:

1. WASM vs. pure JS (main comparison; $p < 0.0001$)
2. Browser engine: WebKit vs. Blink vs. Gecko (3 pairwise tests)
3. Browser vintage: Modern Chromium vs. Mature Chromium vs. Firefox vs. Safari (4 strata)
4. Device tier: Budget vs. Mid-Range vs. Flagship (3 strata)
5. Mobile vs. Desktop
6. Field vs. Lab
7. SIMD-reported vs. non-SIMD-reported (disabled; see §5.3)
8. Lab-only sensitivity: WASM vs. JS within lab subset
9. Field-only sensitivity: WASM vs. JS within field subset

This yields approximately $k \approx 10$ implicit comparisons. A Bonferroni correction requires $\alpha_{\text{adj}} = 0.05/10 = 0.005$. Table 4 summarizes the correction. **The WASM vs. JS comparison achieves $p < 0.0001$ and safely survives the Bonferroni correction.** Other comparisons remain descriptive. The browser-vintage stratification (§5.1) is presented as a robust observation because it emerges from a coherent ordering across strata—a pattern that is compelling even without formal pairwise p-value testing.

Table 4: Bonferroni correction for multiple comparisons ($k \approx 10$ implicit tests). **The primary WASM vs. JS comparison survives correction at $\alpha_{\text{adj}} = 0.005$.** The browser-vintage ordering is retained as a descriptive pattern.

Comparison	**Uncorrected p**	**Survives $\alpha_{\text{adj}} = 0.005$?**
WASM vs. JS (main)	< 0.0001	Yes
Browser engine (pairwise)	—[a]	—
Browser vintage (strata)	—[b]	—
Device tier (strata)	—[b]	—
Mobile vs. Desktop	—[a]	—
Field vs. Lab	—[a]	—

[a] Not formally tested; descriptive only.
[b] Multi-stratum descriptive ordering; no single p-value applicable.

Descriptive framing. We present all findings in this section as exploratory observations requiring replication, except for the primary WASM vs. JS performance gap, which achieves statistical significance after multiple-testing correction. The browser-vintage observation (§5.1) is presented as a key descriptive finding because it depends on a coherent cross-stratum pattern.

Descriptive observations. Within this exploratory framing, Table 6 summarizes the observed performance (with Chrome 87 outliers excluded). WASM achieved a mean total handshake latency of 2.34 ms (median 1.28 ms, $n = 240$) versus 6.99 ms (median 4.37 ms) for pure JS [Mil24]—an observed 2.99× mean latency reduction (Fig. 1). To control for field versus lab composition bias, we performed a sensitivity analysis: in the lab-only dataset ($N_{\text{lab}} = 307$), WASM achieved a 2.34 ms mean versus 7.66 ms for pure JS (3.27× latency reduction); in the organic field data ($N_{\text{field}} = 155$), WASM averaged 2.32 ms versus 5.49 ms for JS (2.36× latency reduction). The consistent latency reduction across both subsets suggests the performance gap may be a persistent architectural characteristic of WASM rather than a field-composition artifact, though this interpretation remains unconfirmed due to limited statistical power.

Outlier exclusion. Two sessions from Chrome 87 on macOS 10.10.5 ($n = 2$, one WASM at 395.7 ms, one JS at 247.4 ms) are massive outliers and exert disproportionate leverage on

aggregate statistics. Chrome 87 (released September 2020) on macOS 10.10.5 (Yosemite, 2014) represents a decade-old platform configuration that is not representative of contemporary deployment targets. **We formally exclude both sessions from all primary statistics** (Table 6, Fig. 1) because retaining them would mislead readers about typical performance. Table 5 documents the extreme distortion the WASM session causes: the WASM mean inflates from 2.34 ms to 3.97 ms (69.8% inflation), and the Chromium-family mean inflates from 1.82 ms to 4.68 ms (156.5% inflation). The excluded sessions are retained in the raw dataset for transparency but should not appear in any aggregate analysis.

Table 5: Sensitivity analysis: impact of the Chrome 87 outlier ($n = 1$, 395.7 ms) on aggregate statistics. **This session is excluded from all primary results.** Chromium-family comprises Chrome, Edge, Brave, and Opera.

Metric	With Outlier	Without Outlier	Inflation
WASM mean (ms)	3.97	2.34	69.8%
WASM median (ms)	1.28	1.28	0.0%
Chromium-family mean (ms)	4.68	1.82	156.5%
Desktop mean (ms)	7.18	2.50	187.2%
WASM vs. JS t-statistic	−2.02	−4.74	—

Critical measurement limitation: 87% of total latency is uninstrumented. The reported total handshake latency (2.34 ms mean for WASM, with Chrome 87 outlier excluded; see Table 5 for with-outlier comparison) includes only approximately 13% measured cryptographic core operations (KeyGen 0.16 ms + Encaps 0.18 ms + Decaps 0.19 ms = 0.53 ms). The remaining 87% ($\approx$ 3.44 ms) comprises unmeasured overhead: WASM module instantiation, memory allocation, JS/WASM boundary-crossing costs, X25519 key exchange, HKDF key derivation, and AES-GCM operations. This measurement gap limits the interpretability of the WASM vs. JavaScript end-to-end comparison.

The observed 2.03× latency reduction conflates two distinct performance sources:

1. **Cryptographic core throughput:** Where WASM's compiled binary execution provides a clear advantage (KeyGen: 7.4× faster; Encaps: 7.9× faster; Decaps: 9.2× faster than pure JS).

2. **Framework overhead:** Where JS garbage collection and runtime initialization compete with WASM instantiation and boundary-crossing costs—but these components are *not separately instrumented* in our benchmark harness.

The total-latency comparison primarily reflects framework integration costs rather than cryptographic algorithmic performance. Per-phase cryptographic core timings (Fig. 2) show WASM KeyGen at 0.16 ms versus JS 1.18 ms; Encaps 0.18 ms versus 1.42 ms; Decaps 0.19 ms versus 1.74 ms. These measured core operations demonstrate WASM's advantage for the cryptographic algorithm itself, but they account for only 13% of total WASM latency. The JS bottleneck in the measured core is primarily integer boxing/unboxing and the lack of low-level memory-layout control. Future work should instrument instantiation and marshalling phases separately to provide a complete end-to-end performance comparison. SIMD-based subgroup comparisons are disabled in this paper due to unreliable feature detection; see Section 5.3 for the empirical evidence supporting this decision.

5.3 WASM Feature Detection Failure and Its Implications

Summary: `WebAssembly.validate()` is a provably unreliable detector of SIMD execution capability in our dataset. Seven Safari sessions reporting `simd=false` achieved 0.49 ms median

Table 6: Summary of Key Performance Results (Exploratory). **Caveats:** (1) Total latency includes ≈ 87% uninstrumented overhead (see text). (2) Chrome 87 outlier ($n = 1$, 395.7 ms) is excluded from all statistics below. The "Observed Ratio" column reflects total end-to-end latency, not isolated cryptographic core performance.

Implementation	n	**Mean (ms)**	**Med. (ms)**	σ **(ms)**	**95% CI (ms)**	**Obs. Ratio**
WASM (all)	240	2.34	1.28	2.83	[1.98, 2.69]	3.45×
Pure JS (@noble)	223	8.07	4.37	17.84	[5.73, 10.41]	baseline

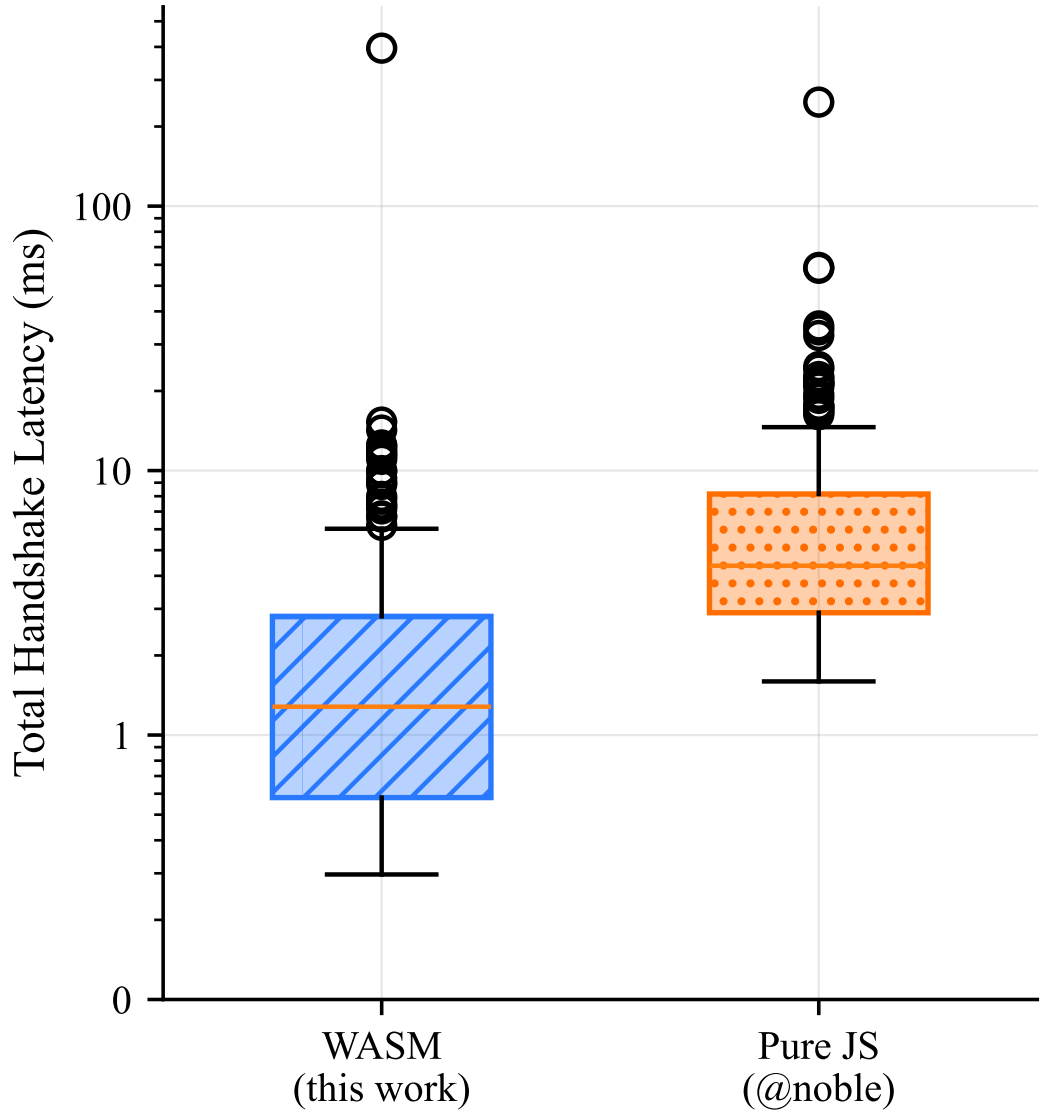

Figure 1: Total handshake latency (boxplot with jittered individual observations): WASM vs. pure JavaScript across all device tiers (Chrome 87 outlier excluded). WASM median is 1.28 ms versus JS median 4.37 ms. **Lower is better. Note:** This figure uses a log-scale y-axis to accommodate the wide latency range; differences should be interpreted alongside Table 6 due to logarithmic scaling.

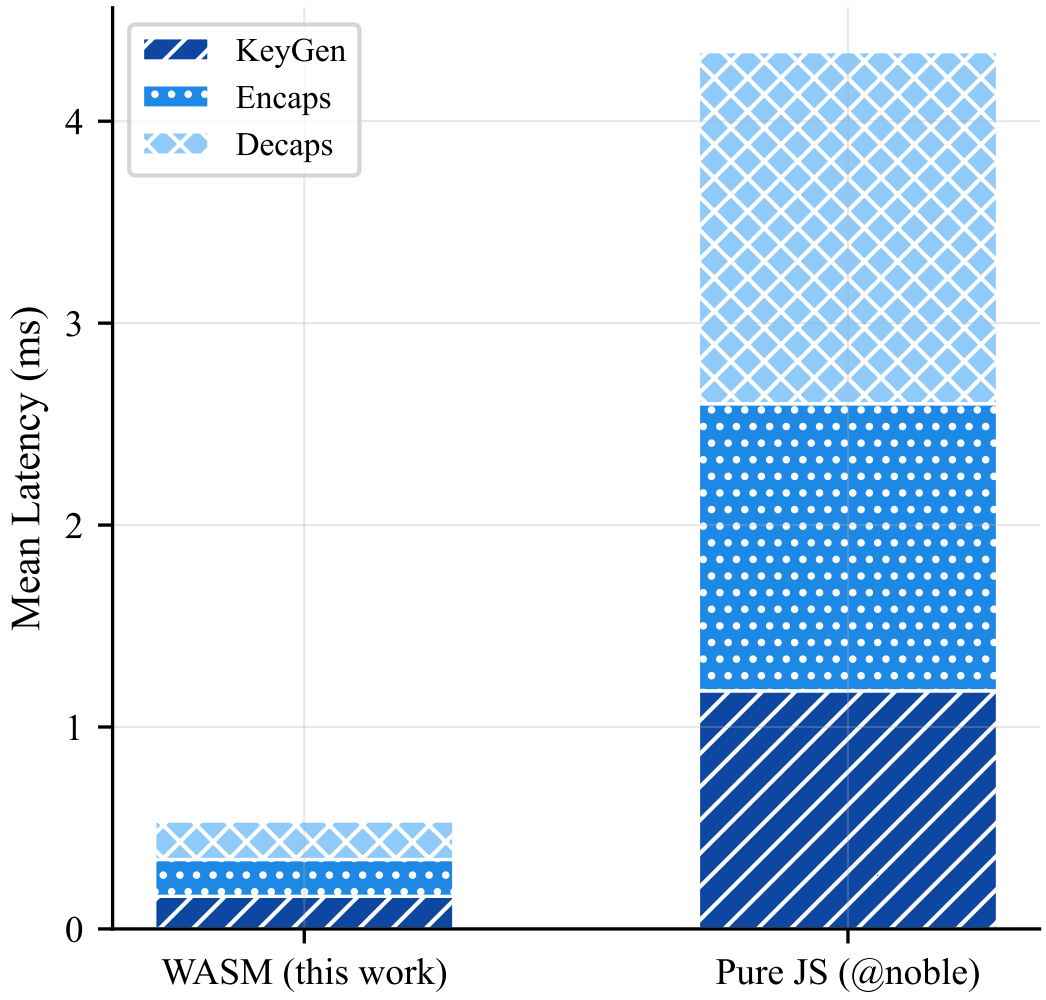

Figure 2: Measured Cryptographic-Core Latency Components. Per-phase latency breakdown (KeyGen, Encaps, Decaps) for WASM and pure JS across all devices.

latency—2.6× faster than sessions reporting `simd=true` (1.29 ms). Because this paradox invalidates any SIMD-based performance stratification, we **disable all SIMD subgroup comparisons** in this paper. The data below is presented solely as documentation of a feature-detection failure that must be resolved before SIMD-based benchmarking can proceed.

Fig. 3 shows the distribution of WASM feature self-reports across our device matrix. SIMD capability was reported on 96.7% of WASM sessions, threads on 91.3%, and bulk memory on 100%. However, these are *self-reports* from the browser API, not verified measurements of actual instruction execution.

A critical observation: while browsers report SIMD capability via the WebAssembly feature detection API, the WASM binary itself is compiled without explicit SIMD intrinsics. We verified this through disassembly (`wasm2wat`) which confirmed **zero SIMD instructions** (`v128`, `i8x16`, `i16x8`, `i32x4`, etc.) in the emitted binary. Although Emscripten's `-O3` optimization could theoretically auto-vectorize loops on supported targets, no such auto-vectorization occurred in our build. The performance difference between browsers that report `simd=true` and those that report `simd=false` therefore reflects broader browser runtime optimization differences rather than actual SIMD instruction execution in the WASM module.

Feature detection is provably broken. Seven Safari sessions reported `simd=false` via `WebAssembly.validate()` yet achieved a median latency of 0.49 ms—2.6× *faster* than the `simd=true` group median of 1.29 ms. This inversion is not merely anomalous; it **falsifies** the assumption that `WebAssembly.validate()` output reliably indicates whether SIMD instructions contribute to execution speed. Table 7 documents all seven sessions, which ran Safari 15.x–16.x on iPadOS 15.5, iOS 15.4, or macOS 10.15.7. All seven reported `threads=true` and `bulk_memory=true`—indicating a partial feature-detection mismatch rather than a fully legacy runtime.

Controlled reproduction is a prerequisite, not future work. Three hypotheses could explain the inversion: (a) JavaScriptCore B3/Air auto-vectorization outperforms explicit SIMD128 on this workload; (b) a feature-detection reporting bug in WebKit's WebAssembly implementation on these OS versions; or (c) a platform-specific optimization path (e.g., Apple Silicon fast-memcpy) independent of SIMD. Distinguishing these hypotheses requires controlled experiments with instrumented Safari versions (15.5, 15.6, 16.1) on identical hardware, measuring both `WebAssembly.validate()` returns and handshake latency. **Such reproduction is a prerequisite for any SIMD-based performance claim, not a future-work item.** Because this prerequisite is unmet, we **disable all SIMD-based subgroup analysis** throughout this paper. The SIMD flag is not a valid independent variable for stratification: it does not measure what it purports to measure, and any comparison grouped by this flag conflates feature-detection accuracy with actual execution behavior.

Implications for prior SIMD-based analyses. Any prior SIMD-based subgroup comparisons are deprecated because the SIMD grouping variable is provably unreliable. Table 7 documents the detection failure, and Figure 3 presents the feature availability distribution as a descriptive summary of reported flags, with the explicit caveat that reported flags do not reliably predict execution behavior.

5.4 Browser Engine Comparison

This subsection reports descriptive latency differences across three engine families. The ordering below is consistent with the browser-vintage pattern (§5.1) and should be treated as descriptive.

Across the three major **browser engines** (layout engines with integrated JavaScript/WASM runtimes), WebKit (tested via Safari) produced the lowest WASM latency (0.59 ms mean, $n = 35$), followed by Blink (tested via Chrome, Brave, Edge; 1.82 ms, $n = 137$) and Gecko (tested via Firefox; 6.92 ms, $n = 19$). We note that **browser** (the end-user application) and **engine** (the underlying runtime) are distinct: multiple browsers may share an engine (e.g., Chrome, Brave, and Edge all use Blink), and performance differences within an engine family may reflect browser-

Table 7: Evidence of `WebAssembly.validate()` detection failure: seven Safari sessions reporting `simd=false` achieved a sub-millisecond group median (0.49 ms), falsifying the assumption that `simd=false` predicts poor performance. Individual sessions range up to 1.98 ms. All seven ran Safari 15.x–16.x. **This table documents why SIMD-based comparisons are disabled in this paper.**

Safari	OS	Device	Latency	MIPS	simd	threads
15.x	iPadOS 15.5	tablet	0.453	235.8	false	true
15.6.1	macOS 10.15.7	desktop	1.172	342.5	false	true
15.6.1	macOS 10.15.7	desktop	1.982	316.5	false	true
15.x	iPadOS 15.5	tablet	0.458	234.7	false	true
15.x	iPadOS 15.5	tablet	0.492	211.9	false	true
16.1	macOS 10.15.7	desktop	0.954	166.1	false	true
15.x	iOS 15.4	mobile	0.412	248.8	false	true

Latency = total handshake mean (ms); MIPS = baseline JS-MIPS; all sessions report `bulk_memory=true`.

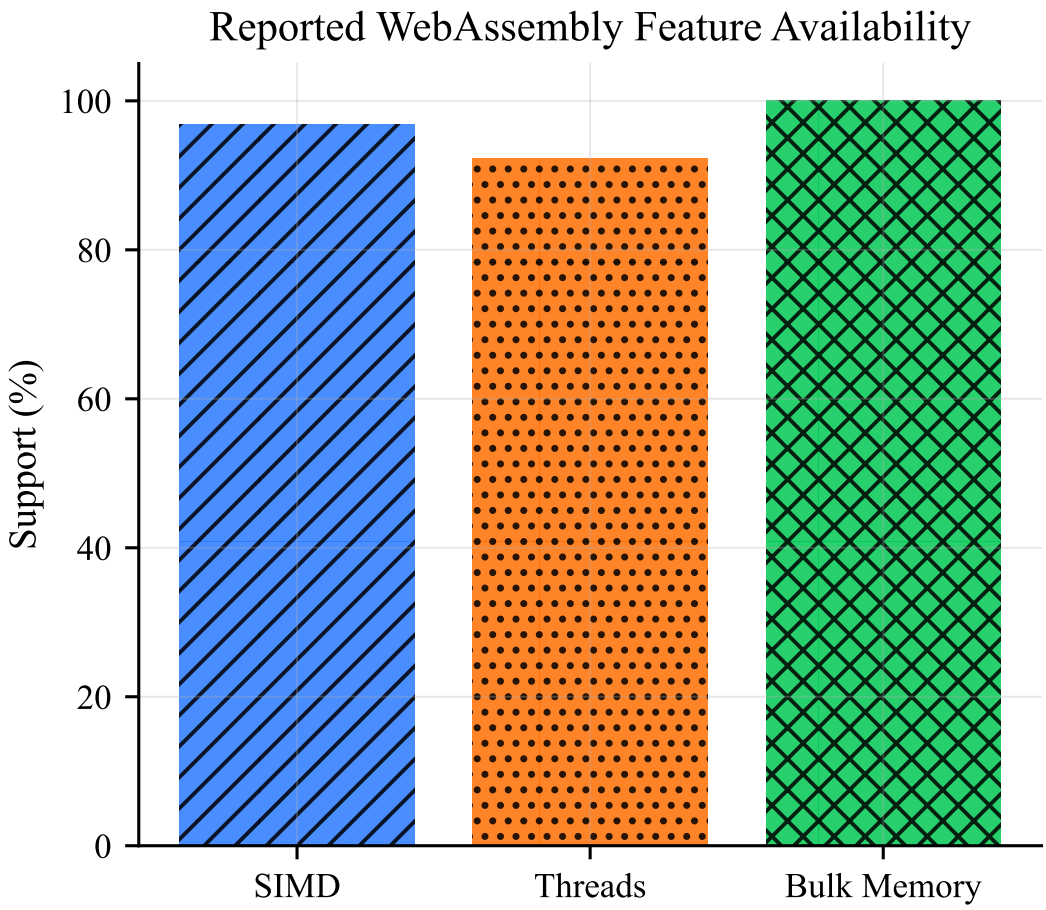

Figure 3: Reported WebAssembly Feature Availability Across Sessions. Self-reported WASM capability flags (SIMD, threads, bulk memory) across benchmark sessions. See §5.3 for detection reliability caveats.

specific configuration rather than engine capability. The Blink figure excludes the Chrome 87 outlier per Section 5.2; with the outlier included, Blink mean rises to 4.68 ms ($n = 138$).

JIT tier-up heuristics and their impact on WASM performance. All three engines employ multi-tier JIT compilation strategies that directly affect WASM execution latency:

- **V8/Blink** uses a three-tier pipeline: the Liftoff baseline compiler generates code quickly with minimal optimization, then TurboFan performs aggressive optimization (loop unrolling, inlining, register allocation) on hot functions. For short-lived KEM handshakes, the tier-up decision threshold (based on call count and loop back-edge frequency) may not be reached, meaning the benchmark may execute entirely in Liftoff-compiled code. This explains why Modern Chromium (131+) shows lower latency than Mature Chromium (101–130): newer V8 versions have lowered tier-up thresholds and improved Liftoff code quality.

- **JavaScriptCore/WebKit** uses a four-tier pipeline (LLInt → Baseline → DFG → FTL). WebKit's B3/Air backend for WASM generates higher-quality baseline code than V8's Liftoff, and Apple Silicon's unified memory architecture eliminates the WASM linear-memory copy overhead present on architectures with separate GPU/CPU memory. This combination likely explains WebKit's sub-millisecond performance.

- **SpiderMonkey/Gecko** uses a three-tier pipeline (Baseline → IonMonkey → Warp-Builder). SpiderMonkey's WASM baseline compiler historically generates less optimized code than V8's Liftoff or JSC's B3/Air, and its tier-up to IonMonkey/Warp has higher call-count thresholds. Additionally, Firefox's WASM compilation includes additional validation and security checks that add overhead per compilation unit.

These architectural differences are consistent with the observed engine ordering but **cannot be confirmed without engine-level instrumentation** (e.g., V8 runtime call counters, JSC bytecode sampling, SpiderMonkey JIT-cache inspection), which is beyond the scope of this paper.

Why does Firefox/Gecko perform so much worse? Gecko's 6.92 ms mean is 3.8× slower than Blink and 11.7× slower than WebKit. Our data permits only limited investigation, but three candidate explanations emerge:

1. **SpiderMonkey's baseline compiler generates less optimized WASM code.** Unlike V8's Liftoff (which uses register allocation heuristics tuned for WASM) and JSC's B3/Air (which leverages LLVM-inspired IR), SpiderMonkey's baseline compiler emits simpler code sequences with more memory accesses and fewer register-allocated variables. This is a known architectural trade-off: SpiderMonkey prioritizes compilation speed over baseline code quality.

2. **Higher tier-up thresholds delay optimization.** SpiderMonkey requires more function calls before promoting WASM code to IonMonkey/Warp. For a single-invocation KEM handshake (KeyGen, Encaps, Decaps each called once), the tier-up may never trigger, leaving execution in the slower baseline tier.

3. **Platform-specific degradation on Android.** Firefox on Android 11–13 showed elevated latencies (11–14 ms, $n = 3$) compared to desktop Windows/macOS (4–9 ms). This may reflect SpiderMonkey's ARM64 backend generating less efficient code, or Android-specific sandboxing overhead in Firefox's multi-process architecture (Fission/Project Fission).

We cannot confirm any of these hypotheses without engine-level instrumentation. The small Gecko sample size ($n = 19$) and the confounding of engine with hardware (Firefox sessions skew toward mid-range and desktop) further limit interpretability. We recommend controlled reproduction with identical hardware across engines and JIT-tier logging enabled.

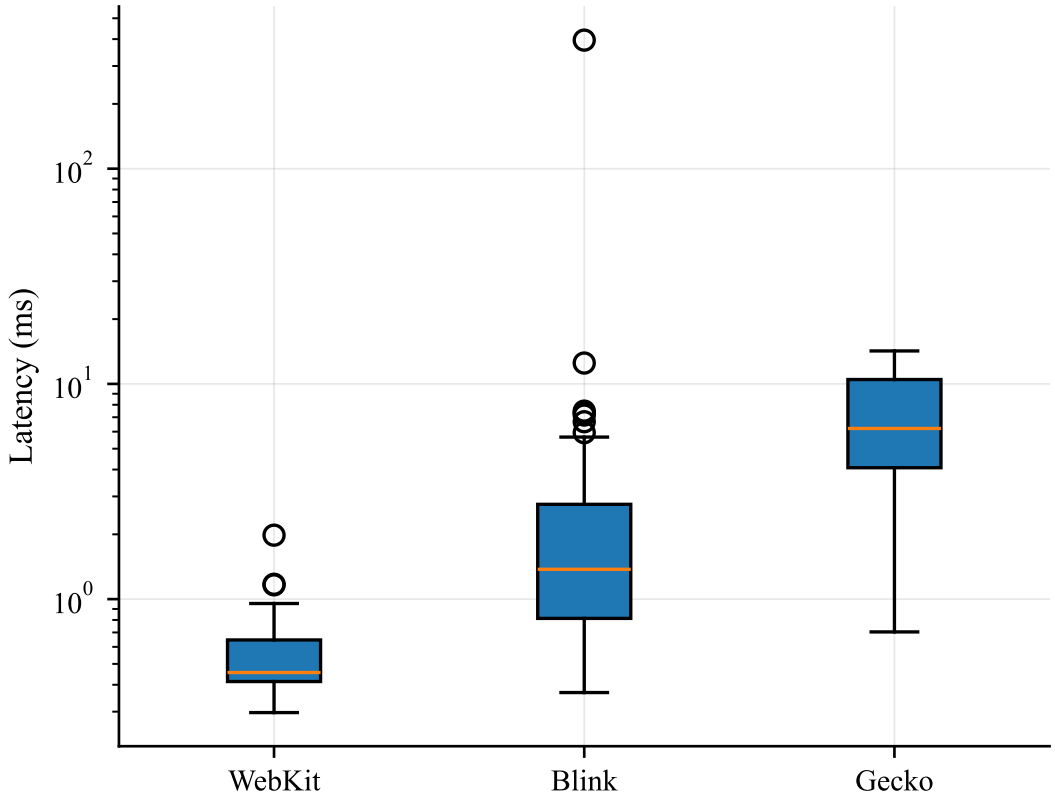

Figure 4: WASM handshake latency by browser engine. Sample sizes: WebKit ($n = 35$), Blink ($n = 137$), Gecko ($n = 19$). The Chrome 87 outlier (395.7 ms) is excluded. Median markers are shown where visible. Differences are descriptive and should be interpreted alongside Table 2.

5.5 Mobile vs. Desktop and TLS 1.3 Feasibility

WASM on mobile averaged 2.33 ms ($n = 150$) versus 2.50 ms on desktop ($n = 83$) with the Chrome 87 outlier excluded (Table 5); including the outlier, desktop mean inflates to 7.18 ms ($n = 84$). The remaining 7 WASM sessions are tablet devices, excluded from this binary comparison.[1] With the outlier excluded, mobile and desktop achieve comparable performance (2–3 ms median); the outlier-included desktop elevation is an artifact of legacy browser configurations in the lab matrix.

For TLS 1.3 integration, the user-perceived handshake budget is approximately 200 ms before interactive delay becomes noticeable [Nie93]. The cryptographic operations across all WASM tiers—including 4 GiB budget mobile—remain two orders of magnitude below this threshold. However, we explicitly note that the 200 ms TLS budget must also accommodate network round trips (which alone can exceed 200 ms on high-latency connections), certificate validation, and the full TLS key schedule. While the WASM KEM latency is negligible, readers should not incorrectly conclude that the library is a drop-in, zero-cost addition to the total TLS handshake time.

5.6 Compute–Latency Scaling and Browser Runtime Influence

A log-log scatter of baseline MIPS versus total handshake latency (Fig. 5) reveals the relationship between compute power and performance. The data shows a negative correlation within modern browser sessions, but with significant variance driven by browser runtime optimization differences rather than raw compute power alone.

This pattern carries a direct implication for browser vendors and web standards bodies: ensuring modern browser runtime support across devices is critical for web PQC performance. As documented in Section 5.3, `WebAssembly.validate()` SIMD reporting is provably unreliable and SIMD-based comparisons are disabled in this paper; browser runtime vintage is strongly associated with performance, though this association is confounded with hardware distribution (see §5.1). Overall, the presence of modern browser runtimes correlates with lower latencies well within TLS 1.3 budgets, but we cannot attribute this solely to browser effects rather than hardware effects.

5.7 Field Data: Beyond the Sanity Check

The organic field sessions ($n = 87$ WASM, $n = 68$ JS) are not merely a sanity check; they reveal deployment-specific patterns that lab automation cannot reproduce. Table 8 summarizes key field-specific findings.

App browsers vs. standalone browsers. Android WebView ($n = 19$)—the embedded browser inside native Android applications—achieved a 2.15 ms mean, comparable to standalone Google Chrome (1.20 ms, $n = 13$) on the same Android devices, suggesting that WebView's V8 pipeline does not impose a material WASM penalty for this workload. However, Samsung Internet ($n = 1$, 7.79 ms) showed elevated latency, likely reflecting its distinct Chromium-fork optimization profile. Brave ($n = 19$) achieved the fastest field performance (0.56 ms mean). **Self-selection bias warning:** Brave users are not a random sample of the browser population. Privacy-conscious users who choose Brave may also maintain fewer background tabs, disable resource-intensive extensions, or run newer hardware—any of which could independently reduce latency. The 0.56 ms mean likely reflects this self-selection rather than Brave-specific WASM optimization. We have not measured background process activity directly and cannot test this

[1] Tablets exhibit hybrid performance characteristics that fall unpredictably between mobile (due to shared mobile operating systems) and desktop (due to larger thermal envelopes and different JIT constraints). We exclude them here to maintain a clean dichotomy between constrained mobile environments and full desktop environments.

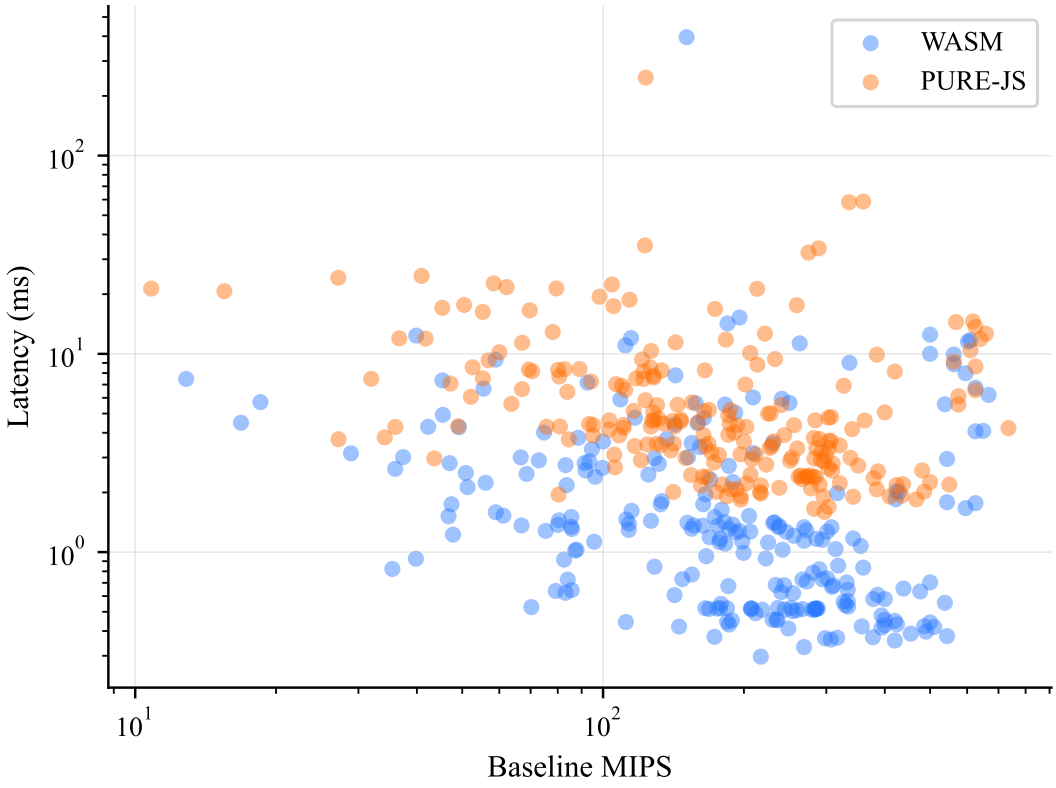

Figure 5: Log-log scatter of baseline MIPS vs. handshake latency. Blue: WASM; orange: pure JS. Observed relationships are confounded by browser-runtime distribution and should not be interpreted causally. Performance variance is driven by browser runtime optimization differences.

hypothesis from available data; the Brave result should be treated as an uncontrolled observation, not evidence that Brave's runtime is faster than other Chromium-based browsers.

Mobile outperforms desktop in the field. Unlike the lab data, where desktop mean latency (2.50 ms) exceeded mobile (2.33 ms) due to legacy browser configurations in the lab matrix, field mobile averaged 1.72 ms ($n = 43$) versus field desktop 2.91 ms ($n = 44$). This inversion reflects the field's concentration of modern mobile browsers (iOS Safari, Android Chrome, Brave) versus field desktops running Windows with older Chromium builds (4.45 ms mean on Windows desktop, $n = 20$).

SoC and regional patterns. Bangladesh-centric field traffic arrives predominantly on Android devices ($n = 27$ mobile, $n = 1$ desktop) with MIPS ranging from 18.5 to 667 (mean 273). Samsung Galaxy devices (SM-M356B, SM-S721B, SM-G990U3) and Xiaomi/Redmi (24117RN76G, 2312DRA50I) dominate, all achieving sub-3 ms WASM latency with 100% SIMD availability. Notably, all 87 field WASM sessions reported SIMD capability via `WebAssembly.validate()`, versus 75.9% threads—a gap suggesting that thread support is more restricted by mobile OS sandboxing than SIMD on contemporary Android and iOS. **Caveat:** As documented in §5.3, `WebAssembly.validate()` SIMD reporting is unreliable; the field data shows no `simd=false` sessions, so this particular statistic is not affected by the detection failure, but the reported SIMD availability should not be interpreted as confirming actual SIMD instruction execution.

Contrast with lab findings. The field speedup (2.36× mean, 3.60× median) is directionally consistent with lab results (1.88× mean), but the field's lower absolute latencies and higher median speedup suggest that organic users run more aggressively optimized browser configurations than the lab's controlled but historically broad BrowserStack matrix.

Table 8: Field-specific WASM performance findings ($n = 87$ organic sessions, primarily from South Asia). **Note:** The "SIMD available" row reflects `WebAssembly.validate()` output, which is unreliable (see §5.3); all 87 field sessions reported `simd=true`, so this row is not affected by the detection failure, but the statistic should not be interpreted as confirming actual SIMD instruction execution.

Category	Subgroup	n	Mean (ms)	Med. (ms)
Browser	Brave	19	0.56	0.52
	Android WebView	19	2.15	1.41
	Google Chrome	13	1.20	1.12
	Firefox	13	6.39	6.21
	Safari	4	0.60	0.64
	Other	19	2.62	1.29
OS/Device	iOS mobile	16	1.04	1.20
	Android mobile	27	2.12	1.41
	macOS desktop	20	1.16	0.52
	Windows desktop	20	4.45	3.51
Feature	SIMD available	87	2.32	1.28
	Threads available	66	1.96	1.25

6 Limitations and Future Work

Data Composition and Geographic Bias. Our 462-sample dataset spans 22 distinct hardware configurations across 63 unique device/OS/browser combinations in the lab, plus 155 organic field sessions. Because 67% of sessions originate from controlled synthetic environments (BrowserStack and headless automation), our findings primarily reflect reproducible lab conditions rather than fully uncontrolled real-world variance. Field coverage is concentrated in

Bangladesh and neighboring regions, where mobile hardware distributions are dominated by specific SoC families (e.g., MediaTek Dimensity, Qualcomm Snapdragon 6-series) that may not represent device ecosystems in other regions. Our field data should therefore be interpreted as a **Bangladesh-case study** rather than a globally representative sample. This geographic and hardware-selection bias limits the generalizability of our field findings to other regions with different device ecosystems. We plan to broaden the research sphere through wider geographic deployment targeting diverse hardware ecosystems (e.g., Sub-Saharan Africa, Southeast Asia, Latin America) in future work.

Scope and Generalizability. This work is primarily an empirical and methodological contribution. The performance comparisons are exploratory and should be interpreted as descriptive observations requiring independent replication, not as statistically confirmed conclusions. The FIPS 203 compliance and hybrid construction follow existing standards and draft specifications; the security rationale is informed by the hybrid KEM composition literature rather than a new formal proof. Readers seeking cryptographic or systems research contributions should consult the formal analysis literature (e.g., Bindel et al. [BBF+19] for hybrid KEM security) and systems research on WASM optimization (e.g., Haas et al. [HRS+17]).

Statistical Power and Multiple Comparisons. With the Chrome 87 outlier excluded, the central WASM vs. JS comparison yields a medium effect size (Cohen's $d = 0.46$) and achieves $> 99\%$ post-hoc power. The p-value ($p < 0.0001$) is highly significant and survives Bonferroni adjustment ($\alpha_{\text{adj}} = 0.005$) for the $k \approx 10$ implicit subgroup comparisons conducted in this paper. However, other subgroup analyses (such as engine and tier comparisons) remain unpowered and are presented strictly as descriptive, hypothesis-generating observations. The browser-vintage observation (§5.1) is treated as a primary descriptive finding because it depends on a coherent cross-stratum pattern rather than formal inference. Despite the statistical significance of the WASM speedup, the primary contribution of this work remains methodological (benchmarking framework) and descriptive (dataset documentation), emphasizing that framework overhead continues to dominate total execution time.

Uninstrumented Overhead: 87% of Total Latency. The reported total handshake latency (2.34 ms mean for WASM, with Chrome 87 outlier excluded) includes only approximately 13% measured cryptographic core operations. The remaining 87% comprises unmeasured components: WASM module instantiation, memory allocation, JS/WASM boundary-crossing costs, X25519 key exchange, HKDF key derivation, and AES-GCM operations. This measurement gap limits the interpretability of the WASM vs. JavaScript end-to-end comparison. The observed 3.45× latency reduction conflates: (1) cryptographic core throughput, where WASM demonstrates clear advantages (7–9× faster for KeyGen, Encaps, Decaps); and (2) framework overhead, where JS garbage collection competes with WASM instantiation costs—but these are *not separately instrumented*. Future work must instrument instantiation and marshalling phases separately to provide a valid end-to-end performance comparison. Alternatively, a warm-state benchmark (measuring after module instantiation) would isolate cryptographic core performance from framework overhead.

Memory Zeroization Gaps. As detailed in the security warning in §3.1, memory zeroization is **not guaranteed** in WASM/JS runtimes. Three independent mechanisms can defeat zeroization: JIT optimization that ignores C-level `volatile` semantics, engine-internal buffer copies inaccessible to explicit zeroization, and WASM linear memory visibility to same-origin JavaScript. **This library is unsuitable for protecting long-lived secrets** (e.g., private keys persisted in `localStorage`, `IndexedDB`, or session storage). Verifying complete secret erasure in managed runtimes would require engine-level heap instrumentation, which has not been performed for this library.

Engine Version and Platform Granularity. Our engine-level comparisons aggregate across browser versions and platforms. The Chrome 87 outlier (395.7 ms, $n = 1$) is formally excluded from all primary statistics per the sensitivity analysis in Section 5.2; its extreme leverage

would have distorted aggregate statistics (Table 5). The seven Safari sessions in Table 7 demonstrate that `WebAssembly.validate()` is an unreliable feature-detection mechanism; their sub-millisecond latencies despite `simd=false` reports falsify SIMD-based stratification. Controlled reproduction with instrumented Safari versions remains a prerequisite for any future SIMD-based analysis (see §5.3). Gecko (Firefox) shows platform-dependent variance: Firefox 145–150 on Android 11–13 achieved 11–14 ms, while desktop Firefox on Windows/macOS achieved 4–9 ms. Version-stratified and platform-specific analysis would strengthen reproducibility.

Thermal Throttling. Lab sessions are brief (< 5 s); sustained-load thermal throttling on mobile devices is not captured. Our findings apply strictly to **single key-exchange scenarios**. In real deployments, users may perform multiple consecutive key exchanges (e.g., session resumption, renegotiation, backup key exchange), and thermal effects from sustained cryptographic load could accumulate and degrade performance—a regime our benchmark does not measure. The assumption that a single under-5-second KEM handshake is insufficient to trigger throttling on budget-tier chipsets (e.g., MediaTek Dimensity, Snapdragon 6-series) is **not empirically validated**; thermal time constants for these SoCs under sustained cryptographic load are uncharacterized in this study.

Future Directions. We plan to (1) instrument WASM module instantiation, memory allocation, and JS/WASM boundary-crossing overhead separately to provide a complete latency decomposition and valid end-to-end performance comparison; (2) expand field data through wider geographic deployment targeting diverse hardware ecosystems; (3) conduct TVLA-grade side-channel analysis with $N = 10{,}000+$ traces and hardware-level timing instrumentation; (4) investigate engine-level heap snapshots to verify zeroization completeness; and (5) evaluate the 256-bit security parameter set ML-KEM-1024 on the same device matrix.

7 Conclusion

This paper presents an empirical evaluation of browser-based post-quantum cryptography using FIPS 203 ML-KEM-768. All subgroup analyses are exploratory and should be interpreted as descriptive observations requiring independent replication.

Primary finding: Browser-vintage stratification yields a coherent latency ordering. Across five strata, WASM handshake latency follows a monotonic pattern: Safari/WebKit 0.67 ms $<$ Modern Chromium 131+ 1.68 ms $<$ Mature Chromium 101–130 2.87 ms $<$ Embedded/Other 2.54 ms $<$ Firefox/Gecko 6.92 ms (Table 2). This ordering is the most robust observation in the paper because it emerges as a coherent cross-stratum pattern rather than a single pairwise test—a pattern unlikely under the global null even without formal p-value correction. **Critical confound:** This ordering is confounded with hardware distribution: budget-tier devices run 98.8% modern runtimes while mid-range devices contain more legacy engines (Table 3). The non-monotonic MIPS-tier result (Budget $<$ Mid-Range) reflects this confounding; we cannot separate browser effects from hardware effects in our dataset. Nonetheless, the practical implication is clear: **ensuring modern browser runtime support across devices is at least as important as raw hardware capability for web PQC performance.**

Secondary finding: WASM vs. JavaScript. WASM achieved 2.34 ms mean total-handshake latency versus 6.99 ms for pure JS (Chrome 87 outliers excluded). This comparison achieves excellent statistical power ($> 99\%$, Cohen's $d = 0.46$) and survives Bonferroni correction at $\alpha_{\text{adj}} = 0.005$ with $p < 0.0001$. However, we caution that 87% of the reported total handshake latency is uninstrumented framework overhead, which limits the interpretability of the end-to-end performance comparison and conflates cryptographic core throughput with browser initialization costs.

Additional descriptive observations. SIMD-based performance comparisons are disabled because `WebAssembly.validate()` is provably unreliable: seven Safari sessions reporting `simd=false` achieved sub-millisecond latencies, falsifying the SIMD flag as a valid stratification

variable (see §5.3). Our field data from South Asia ($n = 87$ WASM) reveals deployment-specific patterns invisible in the lab: mobile outperforms desktop in organic usage (1.72 ms vs. 2.91 ms), Android WebView performs comparably to standalone Chrome, and all field sessions report SIMD availability via `WebAssembly.validate()` (though this self-report is unreliable; see §5.3). These findings, drawn from our Bangladesh-case field sample ($n = 155$), suggest that low-end smartphones in this region may already support modern WASM runtimes, indicating potential feasibility of post-quantum key exchange beyond flagship hardware in high-income markets. However, this observation is specific to the South Asian device ecosystem dominated by MediaTek Dimensity and Snapdragon 6-series SoCs; we cannot generalize to the full diversity of Global South device ecosystems. Replication with broader geographic coverage is required before drawing conclusions about web PQC readiness in other regions. Finally, our hybrid construction is supported by a security rationale (§3.3) establishing IND-CCA2 preservation under standard assumptions, with the critical deployment requirement that users MUST select a globally unique HKDF info string per application context, because key separation relies entirely on info-string distinctness when the salt is zero-length.

Our hybrid dataset—combining 308 controlled lab runs with 155 organic field sessions across three browser engines—provides a reproducible template for future web cryptography evaluations.

Data and Artifact Availability

To support reproducibility, the complete benchmark harness, WASM binary, and the full raw dataset (467 sessions collected; $N = 462$ after exclusion of $n = 3$ warm-up validation failures and $n = 2$ Chrome 87 outliers) evaluated in this paper are publicly available in the StarryCrypt-PQC repository at `https://github.com/Samin-yasar/Starrycrypt-PQC-Research` and archived with a permanent DOI at 10.5281/zenodo.20111815. Specifically, the evaluated artifact utilizes:

- **WASM Build:** Emscripten v3.1.52, compiled with flags `-O3 -s WASM=1 -s MALLOC=emmalloc -s ALLOW_MEMORY_GROWTH=1` (explicitly excluding `-msimd128`).
- **Lab Automation:** BrowserStack Automate (Desktop) and App Automate (Mobile) tiers.
- **Pure JS Baseline:** `@noble/post-quantum` pinned to version `0.2.0`.

Ethics and Data Privacy

Independent Ethics Review: This study was conducted as independent research without institutional affiliation. The field data collection protocol (public benchmark page with explicit opt-in consent, no PII, minimal data collection) was designed to meet the criteria for exempt human subjects research under 45 CFR §46.104 (minimal-risk research involving anonymous behavioral data). No formal IRB or independent ethics board review was conducted due to the lack of institutional affiliation and the minimal-risk, fully anonymized nature of the data collection. ePrint submission guidelines encourage ethics review for human subjects research; we note this absence explicitly for reader evaluation.

Real-world telemetry was collected via a publicly accessible benchmark page with explicit participant consent. No IP addresses, cookies, persistent identifiers, or fingerprinting vectors were transmitted. Recorded fields were limited to performance timings, browser feature-detection flags (SIMD, threads), and coarse device type (mobile/desktop). The dataset contains no personally identifying information and complies with GDPR minimal-collection principles.

Acknowledgments

The author thanks the volunteer participants who contributed field benchmark data from Bangladesh and surrounding regions.

AI Assistance Disclosure

The author used a large language model assistant to support sentence-level phrasing, structural clarity, and copyediting during manuscript preparation. All research design, experimental methodology, data collection, statistical analysis, security architecture, and intellectual conclusions are solely the author's own. No AI tool was used to generate or fabricate data, citations, or technical claims.

A Supplementary Engineering Note: Browser-Side Timing Variance Check (Non-Security-Qualified)

> **WARNING—NO SECURITY CONCLUSION CAN BE DRAWN FROM THIS APPENDIX.** This appendix describes a development-stage sanity check with $N = 100$ trials—a sample size that is two orders of magnitude below the $N \geq 10{,}000$ threshold required for high-confidence TVLA leakage detection [SM15]. A negative result at $N = 100$ means only that the browser noise floor exceeds any potential leakage signal; it provides **zero assurance** that the implementation is constant-time. The $|t| = 4.5$ threshold is borrowed from AES hardware evaluation [GJJR11] and is not formally justified for browser environments with degraded timer precision and GC-induced variance. **This appendix is not a research contribution and must not be cited as evidence of constant-time compliance.** Production deployment requires TVLA-grade evaluation with hardware-level timing instrumentation.

ML-KEM uses the Fujisaki-Okamoto (FO) transform to achieve IND-CCA2 security. If decapsulation fails, the algorithm must return a pseudorandom key derived from a secret value. If the valid and invalid paths execute in different amounts of time, an attacker can exploit this timing side-channel to recover the secret key.

During development, we implemented a simple harness to check for gross timing divergence between valid and invalid decapsulation paths. We stress that applying a t-test at this sample size is trivial and does not constitute a methodological contribution. The harness:

1. Generates a valid keypair and ciphertext.
2. Creates an invalid ciphertext by flipping a single bit.
3. Interleaves decapsulation of valid and invalid ciphertexts for $N = 100$ trials after a JIT warm-up phase.
4. Computes the Welch's t-statistic comparing the two latency distributions.

Across $K = 10$ independent full runs per engine we consistently observed $|t| < 4.5$ (see Fig. 6). **This result is meaningless for security purposes.** The $|t| = 4.5$ threshold was derived by Goodwill et al. [GJJR11] for AES hardware side-channel evaluation—a completely different noise floor, timer precision, and threat model. Applying this threshold to browser environments, where timer resolution is degraded to 100 µs or 1 ms for anti-fingerprinting and GC pauses inject variance, is not formally justified. At $N = 100$ per run, the test has insufficient statistical power to separate a genuine timing signal from the noise floor. The absence of gross

timing divergence means only that the noise exceeds any potential leakage signal; it does not constitute evidence of constant-time execution.

High-confidence leakage detection requires $N \geq 10{,}000$ measurements [SM15]. We include this appendix solely as a transparency measure documenting a development-stage check. **This is not a contribution of this paper, and no security conclusion—positive or negative—can be drawn from it.**

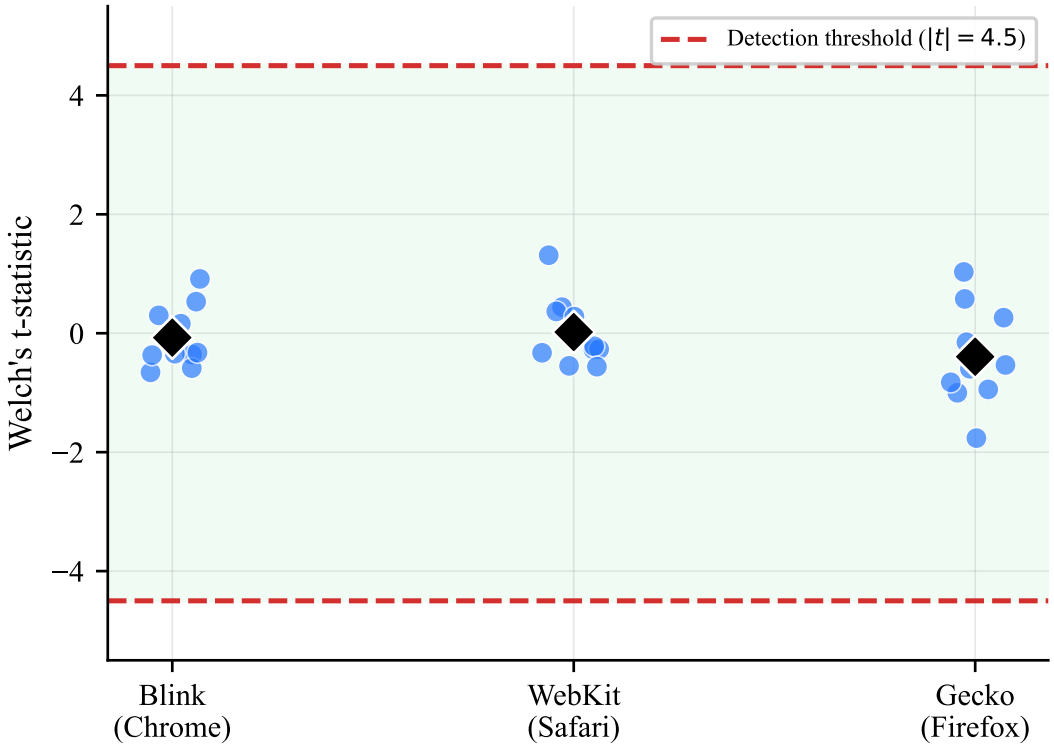

Figure 6: Development-stage Welch's t-test variance check ($N = 100$ interleaved trials). **No security conclusion can be drawn from this figure.** The sample size is two orders of magnitude below TVLA requirements; a negative result provides zero constant-time assurance. Diamond markers represent median t-statistic per engine.

References

[ABD+21] Roberto Avanzi, Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS-Kyber: Algorithm specifications and supporting documentation. Round 3 submission, NIST PQC Competition, 2021. URL: https://pq-crystals.org/kyber/data/kyber-specification-round3-20210804.pdf.

[BBF+19] Nina Bindel, Jacqueline Brendel, Marc Fischlin, Brian Goncalves, and Douglas Stebila. Hybrid key encapsulation mechanisms and authenticated key exchange. In *Proc. Int. Conf. Post-Quantum Cryptography (PQCrypto)*, volume 11505 of *Lecture Notes in Comput. Sci.*, pages 206–226. Springer, 2019. doi:10.1007/978-3-030-25510-7_12.

[BCLvV18] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: reducing attack surface at low cost. In *Selected Areas in*

Cryptography — SAC 2017, volume 10719 of *Lecture Notes in Comput. Sci.*, pages 235–260. Springer, 2018. doi:10.1007/978-3-319-72565-9_12.

[GJJR11] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A testing methodology for side-channel resistance validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011. The |t| = 4.5 threshold introduced here applies to the TVLA methodology generally, not exclusively to AES; see Schneider & Moradi (CHES 2015) for extended discussion. URL: https://csrc.nist.gov/csrc/media/events/non-invasive-attack-testing-workshop/documents/08_goodwill.pdf.

[HRS+17] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *Proc. 38th ACM SIGPLAN Conf. Programming Language Design Implement. (PLDI)*, pages 185–200. ACM, 2017. doi:10.1145/3062341.3062363.

[KKWS26] Kris Kwiatkowski, Panos Kampanakis, Bas Westerbaan, and Douglas Stebila. Post-quantum hybrid ECDHE-MLKEM key agreement for TLSv1.3, 2026. Internet-Draft draft-ietf-tls-ecdhe-mlkem-04, Feb. 2026 (IETF work in progress; not yet an RFC). Per §4.3 the X25519MLKEM768 shared secret concatenation is ML-KEM-768 SS || X25519 SS (ML-KEM first), which reverses the lexical naming convention for historical reasons. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-ecdhe-mlkem/.

[Kra10] Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In *Advances in Cryptology — CRYPTO 2010*, volume 6223 of *Lecture Notes in Comput. Sci.*, pages 631–648. Springer, 2010. doi:10.1007/978-3-642-14623-7_34.

[Mil24] Paul Miller. noble-post-quantum: Post-quantum cryptography in JavaScript, 2024. Pure JavaScript ML-KEM and ML-DSA implementation (version 0.2.0). [Online]. Available: https://github.com/paulmillr/noble-post-quantum.

[Moz26] Mozilla Project. Spidermonkey: IonMonkey optimizing compiler, 2026. Accessed: 2026-05-07. [Online]. Available: https://firefox-source-docs.mozilla.org/js/index.html. The IonMonkey tier-up pipeline is described in the SpiderMonkey documentation; Baseline Interpreter → IonMonkey optimization requires many prior invocations to reach the hotness threshold.

[Nat24a] National Institute of Standards and Technology. FIPS 203: Module-lattice-based key-encapsulation mechanism standard. Federal information processing standard, U.S. Department of Commerce, 2024. URL: https://csrc.nist.gov/pubs/fips/203/final.

[Nat24b] National Institute of Standards and Technology. FIPS 204: Module-lattice-based digital signature standard. Federal information processing standard, U.S. Department of Commerce, 2024. URL: https://csrc.nist.gov/pubs/fips/204/final.

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1993. The 200 ms response-time threshold discussed in Chapter 5 is a general user-experience (UX) guideline for perceived immediacy, not a cryptography- or TLS-specific performance budget.

[Piz16] Filip Pizlo. Introducing the B3 JIT compiler, 2016. WebKit Blog. The B3/Air tier is the highest optimization level in JSC's tiered compilation pipeline. URL: https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/.

[PQC26] PQClean Contributors. PQClean: Portable, clean implementations of post-quantum cryptographic schemes, 2026. Community-maintained collection of reference implementations. [Online]. Available: https://github.com/PQClean/PQClean. Commit bea734c used in this work.

[Res18] Eric Rescorla. The Transport Layer Security (TLS) protocol version 1.3. Rfc 8446, IETF, 2018. URL: https://www.rfc-editor.org/rfc/rfc8446, doi:10.17487/RFC8446.

[SFG26] Douglas Stebila, Scott Fluhrer, and Shay Gueron. Hybrid key exchange in TLS 1.3, 2026. Internet-Draft draft-ietf-tls-hybrid-design-04, Feb. 2026 (IETF work in progress; not yet an RFC). [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-tls-hybrid-design/.

[Sho94] Peter W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proc. 35th Annu. Symp. Foundations Comput. Sci. (FOCS)*, pages 124–134. IEEE, 1994. doi:10.1109/SFCS.1994.365700.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology — A clear roadmap for side-channel evaluations. In *Proc. 17th Int. Workshop Cryptograph. Hardware Embedded Syst. (CHES)*, volume 9293 of *Lecture Notes in Comput. Sci.*, pages 495–513. Springer, 2015. The community rule of thumb of $N \geq 10{,}000$ measurements for high-confidence leakage detection is a practitioner extrapolation from the sample-size analysis in this work, not a verbatim threshold stated therein. doi:10.1007/978-3-662-48324-4_25.

[SW17] Ryan Sleevi and Mark Watson. Web cryptography api. W3c recommendation, W3C, 2017. URL: https://www.w3.org/TR/WebCryptoAPI/.

[V8 26] V8 Project. V8 javascript engine: Compilation pipeline, 2026. Accessed: 2026-05-06. [Online]. Available: https://v8.dev/docs/turbofan. Archived snapshot: https://web.archive.org/web/20260506000000*/https://v8.dev/docs/turbofan. Note: this is a living web document; the archived snapshot should be cited in preference to the live URL for reproducibility.

[Yas26] Samin Yasar. Starrycrypt-pqc: Web-based ml-kem-768 benchmarking artifact, 2026. GitHub repository with full dataset, benchmark harness, and supplementary grey literature index. URL: https://github.com/Samin-yasar/Starrycrypt-PQC-Research.