

Deploying Native-Rust KpqC: Pluggable Post-Quantum KEMs and Signatures across TLS, Space Links, and the OQS Ecosystem

Yulim Hyoung¹[0009-0009-7530-6497], Doyun Park¹[0009-0002-2201-6161],
Hyunji Kim¹[0000-0001-9828-3894], and Hwajeong Seo¹[0000-0003-0069-9061]

Hansung University, Seoul, South Korea
yulim4hyoung@gmail.com rhcp030418@gmail.com
khj1594012@gmail.com hwajeong84@gmail.com

Abstract. Post-quantum migration is no longer only an algorithm problem but a deployment problem: standardized schemes must be dropped into real protocol stacks, on real (often resource-constrained) targets, without giving up memory safety. The Korean Post-Quantum Cryptography (KpqC) competition standardized two KEMs (NTRU+, SMAUG-T) and two signature schemes (HAETAE, AIMer), distributed as memory-unsafe C. Building on our native-Rust KpqC implementation, we expose the schemes through a single *pluggable provider* and evaluate them across three deployment front-ends: (i) a rustls TLS 1.3 key-exchange group, (ii) an Open Quantum Safe (OQS) compatible C ABI that lets the native-Rust code stand in for the C reference behind the liboqs API, and (iii) an analytical CCSDS/SDLS satellite constrained-link model. All three are demonstrated end-to-end: a complete TLS 1.3 handshake using only a KpqC KEM for key establishment, a C program that drives the Rust KEMs and signatures through the OQS-shaped ABI, and a fragmentation/latency analysis over illustrative LEO/GEO link profiles. This work *extends* our earlier C/OQS-OpenSSL TLS benchmark of KpqC (Sim et al.) with a memory-safe Rust implementation, an explicit memory-footprint axis, and deployment breadth beyond localhost/LAN. On a constrained space link, the compact SMAUG-T key exchanges (SMAUG-TiMER 1280 B, SMAUG-T1 1344 B) are smaller and faster than ML-KEM-768 (2272 B). All seven KpqC KEM parameter sets and four signatures (AIMer and HAETAE modes 2/3/5) are demonstrated across the three front-ends. We further give NTRU+ a full memory-optimized build for all three sets, byte-identical and KAT-verified, that cuts peak stack by up to 25% at no measurable time cost ($\approx 0.99\times$ handshake latency).

Keywords: Post-quantum cryptography · KpqC · Rust · Memory safety · TLS · rustls · Open Quantum Safe · Satellite communication.

1 Introduction

Large-scale quantum computers would break the integer-factoring and discrete-logarithm assumptions that RSA and elliptic-curve cryptography rest on, and

with them the key exchange and authentication of protocols such as TLS. In response, NIST standardized post-quantum algorithms including ML-KEM [10] and ML-DSA [11], and several countries have pursued national PQC standards of their own. As one such effort, the Korean Post-Quantum Cryptography (KpqC) competition selected two key-encapsulation mechanisms, NTRU+ [7] and SMAUG-T [4], and two digital-signature schemes, HAETAE [3] and AIMer [5]. Like most PQC reference code, the selected schemes are distributed only as C (the Kpq-Clean suite).

With the algorithms fixed, the open problem shifts from *which* scheme to *how to deploy* it: standardized primitives must drop into real protocol stacks, on real and often resource-constrained targets, without surrendering memory safety. Memory safety is not incidental here: roughly 37% of reported vulnerabilities in widely used C/C++ cryptographic libraries are memory-safety defects rather than cryptographic flaws [2], and the U.S. Office of the National Cyber Director recommends migrating to memory-safe languages, citing Rust [13]. Rust’s ownership-and-borrowing type system enforces spatial and temporal memory safety at compile time with no runtime cost [8], and a growing body of cryptographic code is written in it [15,6]. A native-Rust KpqC implementation that plugs directly into deployment stacks therefore addresses the language-safety and the deployment questions at once.

The deployment axis that matters most for KpqC is memory. Post-quantum keys, ciphertexts, and signatures are far larger than their classical counterparts, so on RAM-constrained embedded and spaceborne targets the working-set size, not raw speed, is often the binding constraint; on bandwidth-constrained links (for example a satellite space link) the on-wire size of a key exchange drives packet fragmentation and handshake latency. These are precisely the quantities a deployment study must measure, and precisely the ones left unquantified so far.

Gap. Our group has already shown KpqC *can* run in TLS by integrating the four schemes into OQS-enabled OpenSSL (C) and benchmarking certificate size and TLS 1.3 handshake latency on localhost and LAN [17]. That work leaves open exactly the axes a deployment story needs: it is C (not memory-safe Rust), it does not measure the memory footprint, it does not use the pure-Rust TLS stack [18], and it does not consider resource-constrained or space links. This paper fills those gaps, building on a native-Rust KpqC implementation.

Contributions.

- **A pluggable provider over native-Rust KpqC.** One object-safe interface exposes the KpqC KEMs and signatures uniformly, spanning all seven KEM parameter sets (NTRU+ 768/864/1152, SMAUG-T1/T3/T5/TiMER) and four signatures (AIMer and HAETAE modes 2/3/5), so that multiple deployment front-ends dispatch them without touching per-scheme code (§3.1).
- **KpqC in rustls TLS 1.3.** We implement the KEMs as a rustls key-exchange group and complete a full TLS 1.3 handshake using *only* a KpqC

KEM for key establishment (§3.2). To our knowledge this is the first native-Rust / rustls TLS integration of KpqC, distinct from the C/OpenSSL integration of [17].

- **An OQS-compatible native-Rust backend.** A C ABI in the `OQS_KEM/OQS_SIG` shape lets the memory-safe Rust code be linked in place of the C reference behind the `liboqs` [14] API (§3.3).
- **A full memory-optimized NTRU+ build.** For all three NTRU+ sets we add a buffer-reuse keypair and a streamed-hash encapsulation (beyond the decapsulation-only prior work), byte-identical to the speed path and KAT-verified; on the canonical machine it cuts peak stack by up to 25% at no measurable time cost ($\approx 0.99\times$ handshake latency, §4.2).
- **A satellite constrained-link deployment analysis.** A CCSDS/SDLS fragmentation model ties the memory-optimized build to space targets and compares KpqC against ML-KEM-768 and X25519 (§4.3).

All integrations here are experimental proofs-of-concept (KpqC has no IANA TLS code point, so the TLS work uses private-use code points and interoperates only between our own endpoints), and we frame them as deployability evidence and as an extension of [17], not as a production or standardized integration.

2 Background

2.1 KpqC Algorithms

The KpqC algorithms comprise key-encapsulation mechanisms (NTRU+ [7] and SMAUG-T [4]) and digital signature schemes (HAETAE [3] and AIMER [5]). They are based on different cryptographic constructions. NTRU+ and SMAUG-T are lattice-based KEMs, HAETAE is a lattice-based Fiat–Shamir signature, and AIMER is an MPC-in-the-Head signature based on a symmetric primitive. These constructions lead to different core implementation routines. NTRU+ uses NTT-based polynomial arithmetic, while SMAUG-T uses Toom–Cook and Karatsuba multiplication. For the signature schemes, HAETAE uses sampling and SHAKE-based expansion, whereas AIMER uses carry-less multiplication over $\text{GF}(2^\lambda)$.

2.2 Implementation Security and Memory Safety in Rust

Memory-safety bugs (e.g., buffer overflows, use-after-free errors, and out-of-bounds reads) occur when code accesses memory outside a valid object’s bounds or lifetime. C provides low-level control and flexibility, but it does not enforce bounds or lifetime safety by default [12]. Rust reduces these risks in safe code through ownership (i.e., each value has a single owner and is dropped when the owner goes out of scope), borrowing (i.e., references may not outlive their referent or alias mutably), and checked indexing [8].

2.3 Cryptographic Algorithm Implementations in Rust

Cryptographic implementations in Rust. Rust has increasingly been used to implement cryptographic algorithms. RustCrypto [15] provides pure-Rust implementations of common cryptographic primitives, rustls [18] is a Rust-based TLS stack, and libcrux [6] develops high-assurance Rust implementations with portable and SIMD backends. These projects show that Rust is a practical language for cryptographic implementations.

KpqC implementations. KpqClean [9] provides the current C implementation and benchmarking framework for the selected KpqC algorithms. Our work builds on this line by providing native Rust implementations of the selected KpqC algorithms with speed and memory optimizations.

2.4 Deploying Post-Quantum Cryptography

Deployment infrastructure. Two pieces of infrastructure carry PQC into real protocol stacks. On the pure-Rust side, rustls [18] exposes a `CryptoProvider` whose key-exchange groups are pluggable, so a new KEM can be added as a `SupportedKxGroup` without forking the TLS state machine; recent rustls ships a hybrid ML-KEM key exchange through this mechanism. On the C side, the Open Quantum Safe project’s liboqs [14] is the de facto vehicle for prototyping PQC in existing protocols: it exposes a uniform `OQS_KEM`/`OQS_SIG` C API and feeds an OpenSSL provider (OQS-OpenSSL). We target both extension points: a rustls key-exchange group in native Rust, and an OQS-shaped C ABI to sit behind liboqs.

Deploying KpqC and related work. Our group has integrated the four KpqC schemes into OQS-enabled OpenSSL and benchmarked certificate size and TLS 1.3 handshake latency on localhost and LAN [17], using a modified liboqs. That study is C-based, does not measure the memory footprint, and does not use the pure-Rust TLS stack or consider resource-constrained links; this paper extends it along exactly those axes. Beyond TLS, post-quantum cryptography has been carried into other deployment stacks: into the DDS middleware of ROS 2 for robotics, evaluated on constrained hardware [16], and into space-system protocols such as CCSDS, where recent surveys map the migration for bandwidth- and memory-constrained links [1]. Notably, these lines integrate only the *standardized* ML-KEM/ML-DSA, and in C; the Korean KpqC schemes have not been brought into them. Placing native-Rust KpqC into a TLS stack, an OQS-compatible backend, and a constrained-link model is, to our knowledge, new.

3 Design

We reuse the KAT-verified native-Rust KpqC implementations unchanged and add a thin adapter layer; no cryptographic behavior is introduced here, only shape adaptation, so byte-level KAT parity is preserved.

Listing 1.1. The object-safe Kem trait (the Sig trait is analogous).

```
pub trait Kem {
    fn name(&self) -> &'static str;
    fn public_key_len(&self) -> usize;
    fn ciphertext_len(&self) -> usize;
    fn shared_secret_len(&self) -> usize;
    fn keypair(&self, rng: &mut dyn Rng) -> (Vec<u8>, Vec<u8>);
    fn encapsulate(&self, rng: &mut dyn Rng, pk: &[u8]) -> (Vec<u8>, Vec<u8>);
    fn decapsulate(&self, sk: &[u8], ct: &[u8]) -> Vec<u8>;
}
```

3.1 Pluggable provider

Each KpqC crate exposes its own `keypair/encapsulate/decapsulate` (KEM) or `keypair/sign/verify` (signature) with per-parameter-set fixed-size array types. We wrap these behind two object-safe traits, `Kem` and `Sig` (`kpqc-provider`), that operate on byte slices so a registry can hold `Box<dyn Kem>` / `Box<dyn Sig>`. A local newtype bridges `&mut dyn Rng` to the generic `Rng`-bounded crate functions (orphan-rule workaround). The provider is `forbid(unsafe_code)`. The KEM trait is shown in Listing 1.1; the signature trait is analogous. The two schemes' failure conventions are absorbed behind the uniform `decapsulate`: NTRU+'s explicit rejection (a status flag plus a zeroed secret) and SMAUG-T's implicit rejection (a pseudo-random secret) both surface simply as the returned shared secret, and a caller needing NTRU+'s flag uses the crate API directly. The registry covers all seven KEM parameter sets and four signatures; the memory-optimized builds (§4.6) are exposed as additional entries.

3.2 TLS front-end (rustls)

A KEM is exposed as a rustls `SupportedKxGroup`. Because a KEM has a data dependency between the two key shares (unlike Diffie-Hellman), the client and server play different roles: the client's `start` runs key generation and sends the encapsulation key as its `key_share`; the server's `start_and_complete` encapsulates to it, yielding the ciphertext (server `key_share`) and the shared secret together; the client's `complete` decapsulates. Each KEM parameter set (and each memory-optimized build) is assigned a distinct private-use TLS Supported Groups code point (in the IANA `0xFE00-0xFEFF` range). Server authentication still uses a classical certificate (KEM-first; KpqC signatures in TLS, which need X.509 object identifiers and certificate tooling, are future work). Because a build registers *only* KpqC key-exchange groups, a completed TLS 1.3 handshake is itself evidence that a KpqC KEM performed the key establishment (§4.1).

3.3 OQS-compatible C ABI

`kpqc-capi` exports `extern "C"` functions matching the OQS function-pointer shapes (`keypair(pk,sk)`, `encaps(ct,ss,pk)`, `decaps(ss,ct,sk)` for KEMs and `keypair/sign/verify`

for signatures), returning 0 on success. This lets the native-Rust implementations be linked in place of the C reference behind the liboqs API. FFI is the one justified use of `unsafe`: each entry validates non-null pointers and treats them as slices of the algorithm’s fixed length. A liboqs fork already registers the KpqC schemes as C entries, so the drop-in target exists; that fork currently builds on a single platform, so we verify ABI conformance independently by linking the static library into a C test program on Linux (§4.1), leaving the end-to-end fork build as an integration step rather than a re-implementation.

3.4 Satellite constrained-link model

An analytical CCSDS/SDLS model (not a packet-level simulator) maps the on-wire key-exchange bytes (public key one way, ciphertext back) onto a link profile (usable frame payload, data rate, one-way delay) to compute frame-fragmentation counts and the 1-RTT key-exchange leg time. Certificate bytes are classical and identical across KEMs, so they are omitted to isolate the KEM’s contribution.

3.5 Memory-safety discipline

The provider and the rustls front-end are `forbid(unsafe_code)` (zero unsafe sites); only the C-ABI shim uses `unsafe`, confined to the FFI boundary and audited (Table 5).

4 Evaluation

Setup. All timing and stack figures are measured on the same canonical machine as the prior work: an 11th-generation Intel Core i5-1135G7 laptop running native Ubuntu Linux, `rustc/cargo` 1.96.0, with the CPU governor and energy preference set to `performance` and the workload pinned to one core (`taskset -c 3`); no thermal throttling occurred during the run. Latencies are wall-clock medians of $n=1000$ per parameter set. Correctness rests on the underlying crates’ byte-level KAT parity with KpqClean (inherited, unchanged), re-verified on this machine. The implementation and the measurement programs are public at <https://github.com/yulim4hyoung/kpqc-rust>; every table in this paper is reproducible from the tag `cisc-2026-09`.

4.1 Functional correctness across the three front-ends

- **Provider round trips.** For every exposed KEM, encapsulate and decapsulate agree on the shared secret; for every signature, a valid signature verifies and a tampered signature / wrong message is rejected (`cargo test -p kpqc-provider`).

Table 1. KpqC KEM buffer sizes and TLS/space-link on-wire bytes (public key + ciphertext) across all parameter sets. Reference rows spec-fixed.

Scheme	pk	sk	ct	ss	on-wire (pk+ct)
SMAUG-T1	672	832	672	32	1344
SMAUG-T3	1088	1312	992	32	2080
SMAUG-T5	1440	1728	1376	32	2816
SMAUG-TiMER	672	832	608	32	1280
NTRU+KEM768	1152	2336	1152	32	2304
NTRU+KEM864	1296	2624	1296	32	2592
NTRU+KEM1152	1728	3488	1728	32	3456
X25519 (ref)	32	–	32	32	64
ML-KEM-768 (ref)	1184	–	1088	32	2272

- **TLS 1.3 handshake.** With a KpqC KEM as the *only* key-exchange group, a full in-memory TLS 1.3 handshake completes and application data flows, for both SMAUG-T1 and NTRU+768, so the KEM demonstrably performed key establishment.
- **OQS C ABI.** A C program links the native-Rust static library and drives keypair/encaps/decaps (KEMs) and keypair/sign/verify (AiMer-128f) through the OQS-shaped ABI; all round trips pass. This verifies ABI conformance on Linux independently of the macOS-only liboqs fork.

4.2 On-wire cost and TLS handshake

Table 1 reports KpqC KEM buffer sizes and the on-wire bytes (public key + ciphertext) that a TLS `key_share` exchange or a space-link handshake carries, for all seven parameter sets. Table 2 reports the TLS 1.3 handshake latency with each KEM as the sole key-exchange group (median of 1000). The compute-only column (0.114–0.190 ms) isolates the KEM+TLS work; the loopback column is a *real* handshake over a 127.0.0.1 TCP socket (0.25–0.29 ms). Two observations follow. First, the speed-vs-memory trade-off: the memory-optimized SMAUG-T build, byte-identical to the speed build, raises the compute time by 2.1–3.6×, the cycle cost paid for its few-KiB footprint (cross-reference Table 8). The NTRU+full memory build is instead essentially free ($\approx 0.99\times$): buffer reuse lowers memory traffic without changing the computation. This build is new here (prior KpqC memory work covered only NTRU+ decapsulation); for all three parameter sets we add a 4-polynomial-reuse keypair and a streamed-hash encapsulation, every path byte-identical to the speed build and KAT-verified. Second, over a real socket the KEM choice is nearly latency-invisible: socket round-trips dominate, so on a fast link the parameter set barely matters, whereas on the constrained satellite link (§4.3) the on-wire size is decisive.

4.3 Satellite constrained-link deployment

Table 3 applies the CCSDS/SDLS model to a LEO TT&C profile. The headline: on a 256 B-frame link, the compact SMAUG-T sets are *smaller and faster than*

Table 2. TLS 1.3 handshake latency with a KpqC KEM as the sole key-exchange group (server auth: classical certificate), median of $n=1000$ on the canonical i5-1135G7. *compute* = in-memory, no network; *loopback* = real handshake over a 127.0.0.1 TCP socket. Over a real socket the KEM choice is nearly latency-invisible (0.25–0.29 ms): socket round-trips dominate the KEM compute difference. The SMAUG-T memory build costs 2.1–3.6 $\times$ the handshake compute time, whereas the NTRU+ full memory build (buffer reuse, byte-identical, all three sets) is essentially free ($\approx 0.99\times$): it reduces memory traffic without changing the computation.

	KEM key exchange compute (ms)	mem/speed	loopback (ms)
SMAUG-T1	0.126	2.11 $\times$	0.249
SMAUG-T3	0.150	2.89 $\times$	0.274
SMAUG-T5	0.179	3.58 $\times$	0.294
SMAUG-TiMER	0.114	2.27 $\times$	0.245
NTRU+KEM768	0.149	0.99 $\times$	0.250
NTRU+KEM864	0.166	0.99 $\times$	0.259
NTRU+KEM1152	0.190	0.99 $\times$	0.270

ML-KEM-768 (2272 B, 5+5 frames, 304 ms): SMAUG-TiMER (1280 B, 3+3 frames, 180 ms) and SMAUG-T1 (1344 B, 3+3, 188 ms), while the larger sets (SMAUG-T5, NTRU+1152) trade extra frames for higher security. NTRU+768 is comparable to ML-KEM-768. Two further link profiles confirm where size actually matters. On a wider, faster LEO S-band link (1115 B frames, 2 Mbps, 10 ms one-way) every KEM fits in one or two frames and the key-exchange leg collapses to 25–29 ms, so the parameter set is nearly irrelevant; on a GEO link (512 B frames, 256 kbps, 125 ms one-way) the ~ 250 ms round-trip propagation dominates and all schemes land within 290–321 ms. Only the tightest 256 B TT&C uplink makes fragmentation and on-wire size decisive: there SMAUG-TiMER and SMAUG-T1 save two frames and ~ 120 ms over ML-KEM-768. The on-board binding constraint, meanwhile, is memory, not link time: the memory-optimized native-Rust build keeps the KEM working set to a few KiB (Tables 7 and 8), which is what makes these schemes viable on RAM-constrained spacecraft. (The link constants are illustrative order-of-magnitude values; the contribution is the model, not the constants.)

4.4 Signatures

Table 4 lists the signature sizes. The provider exposes AIMer-128f and HAETAE modes 2/3/5; all four sign/verify round-trip (a valid signature verifies, a tampered signature or wrong message is rejected). HAETAE’s lattice Fiat–Shamir signatures (1474–2948 B) are markedly smaller than AIMer’s MPC-in-the-Head signature (5888 B). Both AIMer-128f and all three HAETAE modes are additionally driven through the OQS C ABI; the remaining AIMer parameter sets are a mechanical addition.

Table 3. Satellite constrained-link model, LEO TT&C profile (256 B frame, 64 kbps, 10 ms one-way). Frames $\uparrow/\downarrow$ = fragments for the client/server key_share; kx leg = 1-RTT key-exchange time. Illustrative link constants.

Scheme	on-wire (B)	frames $\uparrow$	frames $\downarrow$	kx leg (ms)
SMAUG-T1	1344	3	3	188.0
SMAUG-T3	2080	5	4	280.0
SMAUG-T5	2816	6	6	372.0
SMAUG-TiMER	1280	3	3	180.0
NTRU+KEM768	2304	5	5	308.0
NTRU+KEM864	2592	6	6	344.0
NTRU+KEM1152	3456	7	7	452.0
X25519 (ref)	64	1	1	28.0
ML-KEM-768 (ref)	2272	5	5	304.0

Table 4. KpqC signature sizes exposed through the provider / C ABI. HAETAE (lattice Fiat–Shamir) signatures are markedly smaller than AIMer’s (MPC-in-the-Head).

Scheme	pk	sk	signature
AIMer-128f	32	48	5888
HAETAE-2	992	1408	1474
HAETAE-3	1472	2112	2349
HAETAE-5	2080	2752	2948

4.5 Memory safety

By the same lexical count as the underlying implementation (Table 6), the provider and the rustls front-end contain zero `unsafe` lines (both `forbid(unsafe_code)`); all 174 `unsafe` lines across 34 sites live in the C-ABI shim, where FFI is unavoidable, and each site is guarded by a null-pointer check before it forms a slice (Table 5). This continues the same discipline: safe by construction everywhere except a small, auditable boundary: here the FFI shim, there the compile-time-gated SIMD kernels.

4.6 Memory footprint

Table 7 reports the NTRU+ peak stack for the speed and full memory builds across all three sets and all three operations (paint-and-scan). The full memory build, new here, reduces peak stack by 19.6–25.0% for keypair, 13.5–17.7% for encapsulation, and $\approx 15\%$ for decapsulation, at negligible time cost ($\approx 0.99\times$, Table 2): a rare case where the memory-optimized build is strictly better on the time–space plane. The larger prior speed-vs-memory footprint results for the other schemes are carried in Table 8, and the space-link discussion in §4.3 builds on both.

Table 5. Amount of **unsafe** in the deployment adapter crates (same lexical definition as Table 6). All **unsafe** is confined to the FFI boundary of the C-ABI shim, each site guarded by a null-pointer check; the provider and the rustls front-end forbid it entirely.

Crate	Crate-level policy	unsafe			
		Lines	lines	Share	Sites
kpqc-provider	forbid(unsafe_code)	514	0	0.0%	0
kpqc-rustls	forbid(unsafe_code)	164	0	0.0%	0
kpqc-capi	FFI (audited)	320	174	54.4%	34
Total		998	174	17.4%	34

Table 6. Amount of **unsafe** code per crate (non-blank source lines excluding test modules). *unsafe* counts every line lexically inside an **unsafe** block or **unsafe fn**. The counted regions are confined to compile-time-gated AVX2 or PCLMULQDQ acceleration paths and their call sites; compiled portable workspace configurations contain no project-authored **unsafe** code.

Crate	Crate-level policy	unsafe			
		Lines	lines	Share	Sites
kpqc-common	forbid	135	0	0.0%	0
kpqc-smaug-t	deny	2,532	190	7.5%	13
kpqc-smaug-t3-native	deny	2,231	44	2.0%	6
kpqc-smaug-t5-native	deny	2,267	44	1.9%	6
kpqc-smaug-timer-native	deny	2,292	44	1.9%	6
kpqc-ntru-plus	forbid (portable) / deny (AVX2)	4,937	923	18.7%	28
kpqc-haetae-native	forbid (portable) / deny (AVX2)	5,006	43	0.9%	6
kpqc-aimer-native	deny	7,949	282	3.5%	26
Total		27,349	1,570	5.7%	91

5 Discussion and Conclusion

Deployability. A single pluggable provider let the same native-Rust KpqC implementations (all seven KEM parameter sets and four signatures) appear in three deployment front-ends: a rustls TLS 1.3 key exchange, an OQS-compatible C ABI, and a satellite constrained-link model, each demonstrated end-to-end and re-verified on the canonical machine. The memory-optimized build and SMAUG-T’s compact on-wire key exchange make KpqC attractive precisely where memory and bandwidth are scarce, and the new NTRU+ full memory build delivers up to a 25% peak-stack reduction at essentially no time cost.

Limitations (stated honestly). All integrations are experimental proofs-of-concept: KpqC has no IANA TLS code point, so the TLS work uses private-use code points and interoperates only between our own endpoints. The satellite study is at simulation level (no radiation-hardened hardware); its link constants are illustrative. KpqC signatures *in TLS* (X.509 certificate authentication) and a full ROS2/DDS-Security path are future work. The C ABI itself is verified on

Table 7. NTRU+ peak stack, speed vs the full memory-optimized build (paint-and-scan on the canonical i5-1135G7, the same machine and toolchain as the SMAUG-T stack figures; byte-identical, KAT-verified). Reduction is the stable figure; the handshake-time cost is $\approx 0.99\times$ (Table 2).

Scheme	operation	speed (KiB)	mem (KiB)	reduction
NTRU+KEM768	keypair	14.98	12.04	19.6%
	encapsulate	9.12	7.88	13.5%
	decapsulate	19.89	16.89	15.1%
NTRU+KEM864	keypair	16.82	13.21	21.5%
	encapsulate	10.46	8.94	14.5%
	decapsulate	22.33	18.96	15.1%
NTRU+KEM1152	keypair	22.29	16.73	25.0%
	encapsulate	13.51	11.12	17.7%
	decapsulate	29.60	25.10	15.2%

Linux by linking it into a C test program; building the end-to-end liboqs fork on every platform is a deployment step, not re-implemented here.

Conclusion. This paper extends our C/OQS-OpenSSL TLS benchmark of KpqC [17] with a memory-safe Rust implementation, an explicit memory-footprint axis, and deployment scenarios beyond localhost/LAN. Native-Rust KpqC runs in a pure-Rust TLS 1.3 handshake, drops in behind the OQS C API, and, through its memory-optimized builds, lets a deployment choose the point on the time-space curve its target requires while keeping the memory-safety guarantees a C reference cannot offer. We hope this lowers the barrier to evaluating the Korean PQC portfolio in real systems.

Table 8. Peak stack and peak heap in bytes per operation (deterministic for a fixed build configuration; stack by paint-and-scan, heap by a tracking allocator for Rust and `valgrind -tool=massif` for C). C = KpqClean reference; speed / mem = our speed-optimized and memory-optimized builds. The last column reports the reduction in the sum of separately measured peak stack and peak live heap for mem relative to C; a negative value means mem uses more than C.

Scheme	Op	Stack			Heap			mem red. vs. C
		C	speed	mem	C	speed	mem	
SMAUG-T1	KeyGen	11,463	26,775	4,567	208	0	0	+61%
	Encap	13,319	22,639	4,959	4,568	0	0	+72%
	Decap	14,119	23,439	5,335	4,568	0	0	+71%
SMAUG-T3	KeyGen	15,559	27,975	4,663	208	0	0	+70%
	Encap	17,911	30,031	4,623	4,568	0	0	+79%
	Decap	19,031	37,935	5,031	4,568	0	0	+79%
SMAUG-T5	KeyGen	20,695	35,207	4,999	208	0	0	+76%
	Encap	23,607	37,711	4,943	4,568	0	0	+82%
	Decap	25,111	48,143	5,351	4,568	0	0	+82%
SMAUG-TiMER	KeyGen	11,383	21,767	4,055	1,032	0	0	+67%
	Encap	13,159	23,191	4,383	4,568	0	0	+75%
	Decap	13,895	28,911	5,719	4,568	0	0	+69%
NTRU+768	KeyGen	9,579	15,735	12,791	200	0	0	-31%
	Encap	9,871	9,847	8,695	4,568	0	0	+40%
	Decap	16,575	20,775	11,367	4,568	0	0	+46%
NTRU+864	KeyGen	10,699	17,623	13,991	200	0	0	-28%
	Encap	10,975	11,031	9,687	4,568	0	0	+38%
	Decap	18,511	23,287	12,711	4,568	0	0	+45%
NTRU+1152	KeyGen	14,091	23,223	17,591	200	0	0	-23%
	Encap	14,239	14,343	12,583	4,568	0	0	+33%
	Decap	24,303	30,711	16,599	4,568	0	0	+43%
HAETAE2	KeyGen	24,119	26,455	22,963	10,192	14,480	2,528	+26%
	Sign	83,599	71,959	82,327	6,448	50,872	10,786	-3%
	Verify	38,219	36,230	36,278	4,568	16,400	6,144	+1%
HAETAE3	KeyGen	41,495	35,639	34,211	25,376	33,956	3,712	+43%
	Sign	126,991	105,367	120,263	16,640	67,376	19,885	+2%
	Verify	61,035	47,494	51,622	4,568	36,900	9,216	+7%
HAETAE5	KeyGen	52,727	38,679	42,243	36,192	49,328	4,960	+47%
	Sign	160,111	128,663	136,871	12,896	81,792	21,284	+9%
	Verify	76,683	52,390	47,942	5,408	53,296	11,264	+28%
AIMer128f	KeyGen	8,943	9,807	9,807	208	4,128	4,128	-52%
	Sign	10,703	13,407	17,919	113,808	169,276	10,576	+77%
	Verify	9,947	10,542	13,398	6,176	75,762	8,224	-34%
AIMer128s	KeyGen	8,943	9,807	9,807	208	4,128	4,128	-52%
	Sign	10,431	29,791	37,119	909,456	1,317,308	66,896	+89%
	Verify	9,947	10,510	13,366	25,376	699,714	33,680	-33%
AIMer192f	KeyGen	19,335	19,807	19,807	208	9,264	9,264	-49%
	Sign	21,903	22,815	27,935	253,064	336,412	20,856	+82%
	Verify	20,507	20,534	23,302	11,920	146,586	18,480	-29%
AIMer192s	KeyGen	19,335	19,807	19,807	208	9,264	9,264	-49%
	Sign	21,311	44,351	56,671	2,005,832	2,611,996	104,376	+92%
	Verify	20,507	20,518	23,302	40,720	1,049,098	54,168	-27%
AIMer256f	KeyGen	50,199	50,671	50,671	208	24,640	24,640	-49%
	Sign	53,775	62,639	63,759	522,112	733,692	49,216	+80%
	Verify	51,563	59,974	63,782	27,904	307,202	49,216	-42%
AIMer256s	KeyGen	50,199	50,671	50,671	208	24,640	24,640	-49%
	Sign	52,735	108,751	63,727	4,078,976	5,682,428	170,176	+94%
	Verify	51,547	59,942	63,750	66,304	2,733,730	86,336	-27%

References

1. Post-quantum cryptography for space systems: Algorithms, implementation, and design constraints—a systematic survey. *Acta Astronautica*,

- vol. 246 (2026), <https://www.sciencedirect.com/science/article/abs/pii/S0094576526002730>
2. Blessing, J., Specter, M.A., Weitzner, D.J.: You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries. arXiv:2107.04940 (2021)
3. Cheon, J.H., Choe, H., et al.: HAETAE: Shorter lattice-based fiat-shamir signatures (Feb 2026), <https://www.kpqc.or.kr/images/pdf2/HAETAE.pdf>, korean Post-Quantum Cryptography standardization document
4. Cheon, J.H., Choe, H., et al.: SMAUG-T: The key exchange algorithm based on Module-LWE and Module-LWR (Feb 2026), <https://www.kpqc.or.kr/images/pdf2/SMAUG-T.pdf>, korean Post-Quantum Cryptography standardization document
5. Cho, J., Ha, J., et al.: The AIMer Signature Scheme (Jan 2026), <https://www.kpqc.or.kr/images/pdf2/AIMer.pdf>, korean Post-Quantum Cryptography standardization document
6. Cryspen: libcrux: A formally verified cryptographic library. <https://github.com/cryspen/libcrux> (2024)
7. Kim, J., Park, J.H.: NTRU+: Compact construction of NTRU using simple encoding method (Jan 2026), <https://www.kpqc.or.kr/images/pdf2/NTRU+.pdf>, korean Post-Quantum Cryptography standardization document
8. Klabnik, S., Nichols, C.: The Rust Programming Language. No Starch Press, 2nd edn. (2023)
9. KpqC-CryptoCraft: KpqClean ver2: Clean, portable C reference implementations and benchmarks of the KpqC algorithms. https://github.com/kpqc-cryptocraft/KpqClean_ver2 (2024), accessed July 2026
10. National Institute of Standards and Technology: FIPS 203: Module-lattice-based key-encapsulation mechanism standard (ML-KEM). <https://doi.org/10.6028/NIST.FIPS.203> (2024)
11. National Institute of Standards and Technology: FIPS 204: Module-lattice-based digital signature standard (ML-DSA). <https://doi.org/10.6028/NIST.FIPS.204> (2024)
12. National Security Agency: Software memory safety. Cybersecurity Information Sheet (Nov 2022), https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF
13. Office of the National Cyber Director: Back to the building blocks: A path toward secure and measurable software. The White House (Feb 2024), <https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf>
14. Open Quantum Safe Project: liboqs: C library for quantum-safe cryptographic algorithms. <https://github.com/open-quantum-safe/liboqs> (2024)
15. RustCrypto Developers: RustCrypto: Cryptographic algorithms written in pure Rust. <https://github.com/RustCrypto> (2024)
16. Safdar, U., Qadir, S., Nazir, M.S.: Enhancing ROS 2 (robot operating system 2) security with standardized post-quantum cryptosystems. *International Journal of Information Security* **24**(5) (2025), <https://doi.org/10.1007/s10207-025-01133-w>
17. Sim, M., Song, G., Lee, M., Yoon, S., Baksi, A., Seo, H.: Integrating and benchmarking KpqC in TLS/X.509. *Electronics* **14**(18), 3717 (2025), also IACR ePrint 2025/1245, <https://eprint.iacr.org/2025/1245>
18. The Rustls Project: Rustls: A modern TLS library in Rust (2024), <https://github.com/rustls/rustls>