An Optimized Native Rust Implementation of the KpqC Algorithms

Yu-Lim Hyoung[1][0009−0009−7530−6497], Do-Yun Park[1][0009−0002−2201−6161], Hyun-Ji Kim[1][0000−0001−9828−3894], and Hwa-Jeong Seo[1][0000−0003−0069−9061]

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

Abstract. This paper presents a native Rust implementation of the Korean post-quantum cryptography algorithms (i.e., NTRU+, SMAUG-T, HAETAE, AIMer). We optimize the implementation in Rust for speed and memory usage, and compare our implementation with the C reference code (KpqClean) and unoptimized Rust implementation (baseline). For speed, we use AVX2 and PCLMULQDQ to accelerate NTT- and Toom–Cook-based polynomial multiplication, 4-way SHAKE, and carryless GF(2^λ) multiplication. Compared to the C reference, the speed-optimized version reduces the execution time by 32–76% across all target schemes.
For memory, we reduce large temporary buffers through on-the-fly approach, two-pass sampling, and transcript recomputation. Against the C reference, the memory-optimized implementation reduces the total memory usage by 1–94% across all target schemes.
Overall, these results show a speed–memory trade-off, letting an implementation be chosen according to whichever resource is more constrained.

Keywords: Post-quantum cryptography · KpqC · Rust Implementaion · Memory safety · SIMD acceleration · Memory optimization.

1 Introduction

The development of quantum computers and quantum algorithms threatens the integer-factoring and discrete-logarithm assumptions underlying RSA and elliptic-curve cryptography. This threat has motivated research into quantum-resistant alternatives, now known as post-quantum cryptography (PQC). NIST standardized post-quantum algorithms[1] including ML-KEM [1] and ML-DSA [2], and several other countries have pursued PQC standardization of their own. As one such effort, the Korean Post-Quantum Cryptography (KpqC) competition[2] selected the key-encapsulation mechanism (NTRU+ [3] and SMAUG-T [4]) and

[1] https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/call-for-proposals
[2] https://www.kpqc.or.kr/contents/03_exhibit/sub_03.html,https://github.com/kpqc-cryptocraft/KpqClean_ver2

the digital signature schemes (HAETAE [5] and AIMer [6]). These algorithms are distributed as C reference implementations.

However, C is not a memory-safe language, and roughly 37% of reported vulnerabilities in widely used C/C++ cryptographic libraries are memory-safety defects rather than cryptographic flaws [7]. To reduce such vulnerabilities, the U.S. Office of the National Cyber Director (ONCD) recommends adopting memory-safe languages and identifies Rust as an example [8]. Cryptographic libraries have increasingly been written in Rust [9,10], whose ownership-and-borrowing type system enforces spatial and temporal memory safety at compile time with no runtime cost [11]. For acceleration, Rust implementations also include single-instruction multiple-data (SIMD) backends such as AVX2 [12].

In this paper, we contribute to this body of work by implementing the KpqC algorithms in Rust, obtaining memory safety with SIMD acceleration and efficient memory usage.

1.1 Contribution

This paper presents a memory-safe native Rust implementation of the KpqC algorithms, with trade-offs between speed and memory (see Sections 3.1 and 3.2). Our implementation includes the following optimizations:

- **Acceleration using AVX2 and PCLMULQDQ.** We optimize the core modules used in the KpqC schemes (e.g., Toom–Cook multiplication, NTT, and carry-less GF(2^λ) multiplication) using AVX2 and PCLMULQDQ (Section 3.1), for an improvement of 32%–75% over the C reference (see Table 1). As a trade-off, this speed-optimized implementation increases memory usage.
- **Low memory usage.** We reduce peak memory by regenerating data on the fly and avoiding large temporary buffers (Section 3.2). Across the four KpqC algorithms, the memory-optimized variants reduce peak memory usage by 1–94%. As a trade-off, this memory-optimized implementation increases execution time.

Beyond these optimizations, we follow standard cryptographic implementation practices, including branch-free handling of secret-dependent operations and zeroization of secret buffers.

2 Background

2.1 KpqC Algorithms

The KpqC algorithms comprise key-encapsulation mechanisms (NTRU+ [3] and SMAUG-T [4]) and digital signature schemes (HAETAE [5] and AIMer [6]). 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 GF(2^λ).

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 [13]. 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 [11].

2.3 Cryptographic Algorithm Implementations in Rust

Cryptographic implementations in Rust. Rust has increasingly been used to implement cryptographic algorithms. RustCrypto [9] provides pure-Rust implementations of common cryptographic primitives, rustls [14] is a Rust-based TLS stack, and libcrux [12] 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 [15] 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.

3 Optimized Rust Implementation of KpqC Algorithms

In this section, we present our Rust implementation of the selected KpqC algorithms with two optimization directions: speed-optimization (Section 3.1) and memory-optimization (Section 3.2). Throughout the implementations, SHAKE, a deterministic random-bit generator (DRBG), and encoding routines are shared across schemes.

3.1 Speed Optimization

The speed-optimized version uses AVX2 and PCLMULQDQ to accelerate core modules of each algorithm, including NTT-based polynomial arithmetic, Toom–Cook multiplication, and carry-less multiplication over GF(2^λ). Across the whole implementation, we adopt a 4-way SHAKE [12] that produces four streams in parallel with AVX2, used wherever the schemes expand seeds or sample.

NTRU+ It performs polynomial multiplication over $\mathbb{Z}_q[X]/(X^n - X^{n/2} + 1)$ with $q = 3457$. This multiplication is carried out by applying the NTT, multiplying in the NTT domain, and applying the inverse NTT.

NTT-based multiplication with AVX2. Following [16], we implement the NTT-based multiplication with AVX2 intrinsics. The NTT is a sequence of butterfly stages. Each radix-2 butterfly multiplies one of two coefficients by a twiddle factor (i.e., `zeta`) and outputs their sum and difference, costing one modular multiplication, one addition, and one subtraction.

We store sixteen 16-bit coefficients in one 256-bit register and perform sixteen butterflies at once. Listing 1.1 loads two blocks of sixteen coefficients (lines 5–6), multiplies one by the twiddle factor (line 7), and forms and stores their sixteen sums and differences (lines 8–11). The NTT also includes radix-3 stages that combine three coefficient groups instead of two, vectorized the same way.

Listing 1.1. AVX2 implementation of the radix-2 NTT butterfly in NTRU+.

```
1  unsafe fn fwd_butterfly_simd(r: &mut [i16; NTRUPLUS_N], start: usize, step: usize, zeta: i16
       ) {
2      let zeta_v = _mm256_set1_epi16(zeta);
3      let mut i = start;
4      while i < start + step {
5          let r_lo = _mm256_loadu_si256(r.as_ptr().add(i) as *const __m256i);
6          let r_hi = _mm256_loadu_si256(r.as_ptr().add(i + step) as *const __m256i);
7          let t1 = fqmul_avx2(zeta_v, r_hi);        // 16 twiddle multiplies
8          let new_lo = _mm256_add_epi16(r_lo, t1);   // 16 sums
9          let new_hi = _mm256_sub_epi16(r_lo, t1);   // 16 differences
10         _mm256_storeu_si256(r.as_mut_ptr().add(i) as *mut __m256i, new_lo);
11         _mm256_storeu_si256(r.as_mut_ptr().add(i + step) as *mut __m256i, new_hi);
12         i += 16;
13     }
14 }
```

The twiddle multiplication in the butterfly and the pointwise multiplication in the NTT domain both require multiplication modulo q. Although each coefficient fits in a 16-bit lane since $q = 3457$, the product of two coefficients does not fit in 16 bits. We therefore compute the low and high 16-bit halves of each product and apply Montgomery reduction to sixteen lanes in parallel. Listing 1.2 shows the vectorized Montgomery multiplication. It computes the low and high halves of sixteen products (lines 4–5), derives the Montgomery quotient from the low half (line 6), and subtracts the reduction term from the high half (lines 7–8).

SMAUG-T Toom–Cook [17] is used for polynomial multiplication, because SMAUG-T uses a power-of-two modulus for which an NTT is not directly applicable. Toom–Cook splits each operand into smaller parts and forms the product from several smaller multiplications, together with linear combinations that build the smaller operands beforehand and recombine the products afterwards. We accelerate this using AVX2.

Listing 1.2. AVX2 implementation of Montgomery multiplication in NTT of NTRU+.

```
pub unsafe fn fqmul_avx2(a: __m256i, b: __m256i) -> __m256i {
    let qinv = _mm256_set1_epi16(NTRUPLUS_QINV);
    let q    = _mm256_set1_epi16(NTRUPLUS_Q);
    let lo   = _mm256_mullo_epi16(a, b);
    let hi   = _mm256_mulhi_epi16(a, b);
    let t    = _mm256_mullo_epi16(lo, qinv);
    let qhi  = _mm256_mulhi_epi16(t, q);
    _mm256_sub_epi16(hi, qhi)
}
```

Toom–Cook multiplication with AVX2. We pack sixteen 16-bit coefficients into one 256-bit register, so every operation processes sixteen coefficients at once. The evaluation and interpolation are linear combinations, which use no multiplications. Listing 1.3 shows the evaluation: it loads the four parts of the polynomial (lines 3–6) and forms each smaller operand as a weighted sum of those parts (lines 7–21). Because the weights are powers of two, the parts are scaled by left shifts and combined by additions and subtractions, sixteen coefficients at once.

Listing 1.3. AVX2 implementation of Toom–Cook in SMAUG-T.

```
pub unsafe fn tc_eval_avx2(input: &[__m256i; AVX_N], output: &mut [__m256i; TC_POLYS *
     SMALL_LEN_AVX]) {
    for i in 0..SMALL_LEN_AVX {
        let r0 = input[i];
        let r1 = input[i + SMALL_LEN_AVX];
        let r2 = input[i + 2 * SMALL_LEN_AVX];
        let r3 = input[i + 3 * SMALL_LEN_AVX];
        let r4 = _mm256_add_epi16(r0, r2);
        let r5 = _mm256_add_epi16(r1, r3);
        output[2 * SMALL_LEN_AVX + i] = _mm256_add_epi16(r4, r5);
        output[3 * SMALL_LEN_AVX + i] = _mm256_sub_epi16(r4, r5);
        let r4b = _mm256_slli_epi16(_mm256_add_epi16(_mm256_slli_epi16(r0, 2), r2), 1);
        let r5b = _mm256_add_epi16(_mm256_slli_epi16(r1, 2), r3);
        output[4 * SMALL_LEN_AVX + i] = _mm256_add_epi16(r4b, r5b);
        output[5 * SMALL_LEN_AVX + i] = _mm256_sub_epi16(r4b, r5b);
        let aw2 = _mm256_add_epi16(
            _mm256_add_epi16(_mm256_slli_epi16(r3, 3), _mm256_slli_epi16(r2, 2)),
            _mm256_add_epi16(_mm256_slli_epi16(r1, 1), r0),
        );
        output[SMALL_LEN_AVX + i] = aw2;
        output[6 * SMALL_LEN_AVX + i] = r0;
        output[i] = r3;
    }
}
```

These smaller multiplications are split further by Karatsuba [18], another divide-and-conquer step that lowers the multiplication count. This split uses the same sixteen-coefficient registers, down to a schoolbook base (Listing 1.4) where the coefficient products are formed. Because q is a power of two, this base multiply needs no reduction step: its products wrap modulo 2^{16} for free, and reduction modulo q is only a bit mask.

Listing 1.4. AVX2 implementation of the schoolbook base multiplication in SMAUG-T.

```
1  pub unsafe fn schoolbook_avx2(a: &[__m256i; 16], b: &[__m256i; 16], c: &mut [__m256i; 32],
       acc: bool) {
2      if !acc {
3          for slot in c.iter_mut() {
4              *slot = _mm256_setzero_si256();
5          }
6      }
7      for i in 0..16 {
8          let ai = a[i];
9          for j in 0..16 {
10             let prod = _mm256_mullo_epi16(ai, b[j]);
11             c[i + j] = _mm256_add_epi16(c[i + j], prod);
12         }
13     }
14 }
```

HAETAE HAETAE [5] uses NTT-based polynomial arithmetic and SHAKE256-based sampling, and its signing time is dominated by hyperball sampling and the associated SHAKE256 rather than by the NTT.

4-way SHAKE and AVX2 NTT. We apply a 4-way SHAKE to the SHAKE256 calls used in sampling, producing four independent streams in parallel, and we provide a hand-written AVX2 NTT. Because signing is sampling-bound, most of its speedup comes from the 4-way SHAKE, while the AVX2 NTT contributes more to key generation and verification.

AIMer AIMer uses arithmetic over $\mathrm{GF}(2^\lambda)$ in the AIM2 computation and in the MPC-in-the-Head multiplication-checking protocol [6]. We therefore accelerate the field multiplication.

Carry-less $\mathrm{GF}(2^\lambda)$ *multiplication with PCLMULQDQ.* Multiplication over $\mathrm{GF}(2^\lambda)$ in AIMer can be implemented as carry-less polynomial multiplication followed by reduction modulo the field polynomial [6]. A single PCLMULQDQ instruction[3] (i.e., a CPU instruction that performs carry-less multiplication) multiplies two 64-bit values, and the 128-bit product is held in one 128-bit register. Unlike the AVX2 implementation above, it does not require lane packing.

Listing 1.5 shows the multiplication for $\lambda = 128$. The 128-bit multiplication is computed from four carry-less products of the 64-bit halves (lines 4–7). These products are independent, so the processor can compute them in parallel. Here the parallelism comes from independent instructions, not from the data-parallel lanes of the AVX2 kernels. Their cross-terms are then combined by shifts and exclusive-ors (lines 8–10).

[3] https://www.felixcloutier.com/x86/pclmulqdq

Listing 1.5. Carry-less GF(2^{128}) multiplication using PCLMULQDQ.

```
unsafe fn mul_reduce(ta: __m128i, tb: __m128i) -> __m128i {
    let irr = _mm_set_epi64x(0x0, 0x87);
    // carry-less polynomial multiplication
    let mut tc0 = _mm_clmulepi64_si128(ta, tb, 0x00);
    let mut tc1 = _mm_clmulepi64_si128(ta, tb, 0x11);
    let t01 = _mm_clmulepi64_si128(ta, tb, 0x01);
    let t10 = _mm_clmulepi64_si128(ta, tb, 0x10);
    let t0 = _mm_xor_si128(t01, t10);
    tc0 = _mm_xor_si128(tc0, _mm_slli_si128(t0, 8));
    tc1 = _mm_xor_si128(tc1, _mm_srli_si128(t0, 8));
    // modular reduction
    let r0 = _mm_clmulepi64_si128(tc1, irr, 0x01);
    let r1 = _mm_slli_si128(r0, 8);
    let mut r2 = _mm_srli_si128(r0, 8);
    r2 = _mm_xor_si128(r2, tc1);
    tc0 = _mm_xor_si128(tc0, _mm_clmulepi64_si128(r2, irr, 0x00));
    _mm_xor_si128(tc0, r1)
}
```

3.2 Memory Optimization

The memory-optimized implementation keeps peak memory low by regenerating large temporary objects from a seed when they are needed, or by producing and consuming them incrementally, at the cost of additional execution time.

NTRU+ Following the memory-optimization methodology of [19], we reduce NTRU+'s peak stack. Since NTRU+ keeps no large public-key matrix or multiplication scratch, its reduction instead comes from reusing buffers to keep fewer polynomials live. In key generation, the inverse of a secret polynomial is used only as an invertibility test [3, Section 5.1], so it is written into the buffer that later holds the public key. In decapsulation, the modular reduction and the subtraction run in place, and the re-encryption is compared against the ciphertext block by block, so the re-encrypted polynomial is not stored in full.

SMAUG-T Following [19], we reduce the largest stack objects in turn. The public-key matrix A [4, Section 3.1] is regenerated on the fly, one entry at a time as it is used, instead of being stored. The polynomial multiplication uses a schoolbook method, which avoids the large scratch buffers of the Toom–Cook multiplication [20] used in the speed-optimized version. Each secret-key coefficient (-1, 0, or 1) is stored in a single byte, where the C reference uses two.

HAETAE Following the memory-optimization techniques for HAETAE of [19,21], we reduce peak heap. HAETAE keeps its large objects, the public-key matrix and the buffer of random values for the signature, on the heap. The matrix is regenerated on the fly, one row at a time as the multiplication consumes it. The random values are used twice during signing [5, Section 4.2]; the C reference

implementation [15] stores them in a buffer for the second use, whereas we regenerate them from the same seed, removing the buffer at the cost of generating them twice.

AIMer AIMer's signing runs an MPC-in-the-Head computation among N parties across parallel repetitions [6, Section 4.2]. The C reference implementation [15] keeps the data for every repetition on the heap at once, so peak heap grows with N times the number of repetitions. We instead keep only a seed per repetition and regenerate its data when it is needed for hashing or opening, discarding it after use, so only one repetition's data is resident at a time.

4 Results

This section evaluates the performance of our Rust implementation of the KpqC algorithms. The implementation is validated against the test vectors, and all experiments are conducted on an Intel Core i5-1135G7 (Linux).

4.1 Speed Optimization

Table 1 shows that our speed-optimized Rust implementation improves over the C reference implementation. In NTRU+, the AVX2 NTT handles the polynomial multiplication in all three operations and the extra inversions in key generation, so key generation obtains the largest improvement, 75–81%. SMAUG-T applies its Toom–Cook multiplication across all operations and improves each by 49–64%. HAETAE improves less, by 16–46%, because its cost is dominated by hyperball sampling. In AIMer, the $\mathrm{GF}(2^{\lambda})$ multiplication accelerated by PCLMULQDQ makes up a large part of key generation and signing, which improve by 67–72% and 56–75%.

4.2 Memory Optimization

Table 2 shows the memory usage of our memory-optimized implementation against the C reference. Note that total memory usage is the sum of peak stack and peak heap memory (i.e., Total = Stack + Heap). Relative to the C reference implementation, our implementation matches or reduces the total peak memory usage across all KpqC algorithms.

For NTRU+, the peak working set is stack-only for both implementations, for a total reduction of 22.8–27.6%. For SMAUG-T, the peak is likewise stack-only, for a total reduction of 58.8–78.7%. It regenerates the public matrix on the fly and uses school-book multiplication without scratch buffers, which lowers its large stack objects. For HAETAE, both the stack and the heap decrease, except at HAETAE2 where the heap increases slightly. The total reduction rate by removing the sample buffer is 0.7–19.4%. For AIMer, the stack increases while the heap decreases. Our implementation regenerates each repetition from a seed and holds only one at a time, so the data that was on the heap is computed on the stack for a single repetition. The stack therefore rises, but removing the per-repetition heap storage gives a total reduction of 77–94%.

Table 1. Execution time comparison against the C reference (unit: milliseconds).

Scheme (KEM)	KeyGen	Encaps	Decaps	Total
NTRU+768	{0.012, 0.059} (79.3%)	{0.010, 0.038} (75.1%)	{0.014, 0.044} (68.8%)	{0.035, 0.141} (74.9%)
NTRU+864	{0.018, 0.071} (75.2%)	{0.014, 0.043} (67.1%)	{0.020, 0.051} (61.2%)	{0.051, 0.164} (68.8%)
NTRU+1152	{0.021, 0.115} (81.3%)	{0.015, 0.059} (74.8%)	{0.021, 0.069} (69.1%)	{0.057, 0.243} (76.3%)
SMAUG-T1	{0.012, 0.024} (51.5%)	{0.010, 0.021} (51.9%)	{0.014, 0.028} (48.5%)	{0.036, 0.073} (50.4%)
SMAUG-T3	{0.020, 0.047} (58.0%)	{0.019, 0.043} (56.3%)	{0.025, 0.053} (52.4%)	{0.064, 0.143} (55.4%)
SMAUG-T5	{0.028, 0.075} (63.6%)	{0.027, 0.070} (61.8%)	{0.035, 0.084} (57.5%)	{0.090, 0.229} (60.9%)
SMAUG-TiMER	{0.011, 0.024} (53.0%)	{0.009, 0.021} (55.2%)	{0.013, 0.028} (53.1%)	{0.033, 0.072} (53.7%)

Scheme (DSA)	KeyGen	Sign	Verify	Total
HAETAE2	{0.121, 0.149} (18.3%)	{0.779, 1.230} (36.9%)	{0.032, 0.050} (35.5%)	{0.932, 1.429} (34.8%)
HAETAE3	{0.327, 0.400} (18.3%)	{0.857, 1.350} (36.5%)	{0.054, 0.088} (38.4%)	{1.238, 1.838} (32.6%)
HAETAE5	{0.299, 0.357} (16.2%)	{0.989, 1.660} (40.4%)	{0.059, 0.109} (45.6%)	{1.347, 2.126} (36.6%)
AIMer128f	{0.013, 0.038} (67.1%)	{0.922, 2.140} (56.9%)	{1.220, 1.990} (39.0%)	{2.155, 4.168} (48.3%)
AIMer128s	{0.011, 0.038} (69.5%)	{7.590, 17.300} (56.2%)	{10.500, 17.100} (38.5%)	{18.102, 34.438} (47.4%)
AIMer192f	{0.028, 0.102} (72.4%)	{1.040, 4.210} (75.2%)	{1.980, 3.890} (49.0%)	{3.048, 8.202} (62.8%)
AIMer192s	{0.028, 0.101} (72.2%)	{8.080, 32.600} (75.2%)	{16.400, 32.500} (49.6%)	{24.508, 65.201} (62.4%)
AIMer256f	{0.073, 0.249} (70.8%)	{4.120, 10.800} (61.7%)	{5.330, 9.910} (46.2%)	{9.523, 20.959} (54.6%)
AIMer256s	{0.072, 0.248} (70.9%)	{31.000, 84.100} (63.1%)	{42.800, 83.600} (48.8%)	{73.872, 167.948} (56.0%)

Each cell is {Rust implementation (ours), C reference (KpqClean)}, and all values are rounded.
Improvement percentages are relative to the C reference.

Table 2. Peak memory usage comparison against the C reference (unit: bytes).

Scheme (KEM)	Stack	Heap	Total
NTRU+768	{12,791, 16,575}	{0, 0}	{12,791, 16,575} (22.8%)
NTRU+864	{13,991, 18,511}	{0, 0}	{13,991, 18,511} (24.4%)
NTRU+1152	{17,591, 24,303}	{0, 0}	{17,591, 24,303} (27.6%)
SMAUG-T1	{5,335, 14,119}	{0, 0}	{5,335, 14,119} (62.2%)
SMAUG-T3	{5,031, 19,031}	{0, 0}	{5,031, 19,031} (73.6%)
SMAUG-T5	{5,351, 25,111}	{0, 0}	{5,351, 25,111} (78.7%)
SMAUG-TiMER	{5,719, 13,895}	{0, 0}	{5,719, 13,895} (58.8%)
HAETAE2	{82,327, 83,599}	{10,786, 10,192}	{93,113, 93,791} (0.7%)
HAETAE3	{120,263, 126,991}	{19,885, 25,376}	{140,148, 152,367} (8.0%)
HAETAE5	{136,871, 160,111}	{21,284, 36,192}	{158,155, 196,303} (19.4%)
AIMer128f	{17,919, 10,703}	{10,576, 113,808}	{28,495, 124,511} (77.1%)
AIMer128s	{37,119, 10,431}	{66,896, 909,456}	{104,015, 919,887} (88.7%)
AIMer192f	{27,935, 21,903}	{20,856, 253,064}	{48,791, 274,967} (82.3%)
AIMer192s	{56,671, 21,311}	{104,376, 2,005,832}	{161,047, 2,027,143} (92.1%)
AIMer256f	{63,782, 53,775}	{49,216, 522,112}	{112,998, 575,887} (80.4%)
AIMer256s	{63,750, 52,735}	{170,176, 4,078,976}	{233,926, 4,131,711} (94.3%)

Each cell is {Rust implementation (ours), C reference (KpqClean)}.
Reduction percentages (Total only) are relative to the C reference.

5 Conclusion

Memory-safe languages have recently been recommended, and Rust has increasingly been adopted in cryptographic implementations. However, the KpqC algorithms are currently distributed as C reference implementations, leaving a gap for a memory-safe Rust realization.

We presented a native Rust implementation of the KpqC algorithms (NTRU+, SMAUG-T, HAETAE, and AIMer), spanning schemes of very different structure in a single memory-safe framework. Using AVX2 and PCLMULQDQ, our speed-optimized build accelerates the core modules including the NTT, Toom–Cook, and carry-less GF(2^λ) multiplication, and a 4-way SHAKE, reducing the execution time on every operation by up to 76% over the C reference. Our memory-optimized variants regenerate large temporaries instead of storing them, reducing total memory usage up to 94% over the C reference.

An ARM backend using the NEON or SVE vector extensions would extend our optimized Rust implementation. Benchmarking across a wider range of environments and a balanced design between the memory- and speed-optimized implementations would also be interesting directions for future work.

References

1. 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.
2. 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.
3. Jonghyun Kim and Jong Hwan Park. NTRU+: Compact construction of NTRU using simple encoding method, January 2026. Korean Post-Quantum Cryptography standardization document.
4. Jung Hee Cheon, Hyeongmin Choe, et al. SMAUG-T: The key exchange algorithm based on Module-LWE and Module-LWR, February 2026. Korean Post-Quantum Cryptography standardization document.
5. Jung Hee Cheon, Hyeongmin Choe, et al. HAETAE: Shorter lattice-based fiat-shamir signatures, February 2026. Korean Post-Quantum Cryptography standardization document.
6. Jihoon Cho, Jincheol Ha, et al. The AIMer Signature Scheme, January 2026. Korean Post-Quantum Cryptography standardization document.
7. Jenny Blessing, Michael A. Specter, and Daniel J. Weitzner. You really shouldn't roll your own crypto: An empirical study of vulnerabilities in cryptographic libraries. arXiv:2107.04940, 2021.
8. Office of the National Cyber Director. Back to the building blocks: A path toward secure and measurable software. The White House, February 2024. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.
9. RustCrypto Developers. RustCrypto: Cryptographic algorithms written in pure Rust. https://github.com/RustCrypto, 2024.

10. rusty-saber contributors. rusty-saber: A safe pure-Rust implementation of the Saber KEM. https://crates.io/crates/rusty-saber, 2022.
11. Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2nd edition, 2023.
12. Cryspen. libcrux: A formally verified cryptographic library. https://github.com/cryspen/libcrux, 2024.
13. National Security Agency. Software memory safety. Cybersecurity Information Sheet, November 2022. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.
14. The Rustls Project. Rustls: A modern TLS library in Rust, 2024. https://github.com/rustls/rustls.
15. 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.
16. Gregor Seiler. Faster AVX2 optimized NTT multiplication for Ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018. https://eprint.iacr.org/2018/039.
17. Marco Bodrato. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In *Arithmetic of Finite Fields (WAIFI)*, volume 4547 of *LNCS*, pages 116–133, 2007.
18. Anatolii Karatsuba and Yuri Ofman. Multiplication of many-digital numbers by automatic computers. *Doklady Akademii Nauk SSSR*, 145(2):293–294, 1962.
19. Yulim Hyoung, Subeen Cho, et al. Memory-efficient implementation of SMAUG-T and HAETAE. Cryptology ePrint Archive, Paper 2026/442, 2026. https://eprint.iacr.org/2026/442.
20. Jose Maria Bermudo Mera, Angshuman Karmakar, and Ingrid Verbauwhede. Time-memory trade-off in Toom–Cook multiplication: An application to module-lattice based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, 2020(2):222–244, 2020. Cryptology ePrint Archive, Paper 2020/268.
21. Gustavo Banegas, YoungBeom Kim, Seog Chung Seo, and Christine van Vredendaal. Low-stack HAETAE for memory-constrained microcontrollers. Cryptology ePrint Archive, Paper 2026/635, 2026. https://eprint.iacr.org/2026/635.