Efficient Implementations of Small-Modulus Lattice KEM Components

Ying Liu[1,2], Yu Zhang[1,2] and Xianhui Lu[1,2]

[1] Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
liuying,luxianhui,zhangyu1999@iie.ac.cn

Abstract. Small moduli support compact polynomial representations in lattice-based key encapsulation mechanisms (KEMs), including BAT (TCHES 2022) and END (ASIACRYPT 2026, to appear). However, compact encodings can be costly to convert, and uniform generation must account for both coefficient-extraction cost and random-bit demand. Incomplete number-theoretic transforms also leave residual polynomial products. To address these costs, we exploit the structure of moduli 257 and 769 to improve encoding, uniform public polynomial generation, and polynomial multiplication.

At $q = 257$, we introduce zero-selector encoding: zero payload bytes locate ambiguous coefficients, and selector bits resolve their values. We analyze selector capacity, overflow, and key-generation retries. For $n = 1024$, a 1033-byte configuration has overflow probability at most 2^{-128} under independent uniform coefficients. Checked AVX2 decoding takes 213 cycles, compared with 2203 cycles for BAT's 1040-byte format. At $q = 769$, we lower conversion costs using an existing compact radix format at the same encoded length as BAT. We specialize rejection sampling to simplify coefficient extraction at $q = 257$ and reduce random-bit demand at $q = 769$. At $(q, n) = (769, 1024)$, expected candidate-bit demand decreases by 24.39% relative to direct 10-bit rejection. Four-polynomial generation uses 7.14-13.21% fewer AVX2 cycles. For residual multiplication, proven range bounds justify omitting strict input-centering corrections, while register scheduling reduces stack accesses.

We integrate the public-key codecs and multiplication components into BAT and END. Across the tested C and AVX2 implementations, encapsulation costs decrease by $39.93 - 55.25\%$ in BAT and $41.36 - 48.84\%$ in END. The corresponding decapsulation reductions are $5.42 - 8.55\%$ in BAT and, for END, $23.99 - 45.89\%$ in C and $6.98 - 8.04\%$ in AVX2. END AVX2 gains mainly come from the public key codec. These results quantify the effect of modulus-specific optimizations on complete component and KEM costs.

Keywords: Lattice-based cryptography · KEM · Encoding · Arithmetic · Sampling · AVX2

1 Introduction

Lattice-based key encapsulation mechanisms (KEMs) require efficient polynomial representation, public generation, and multiplication. Representation determines the transmitted length and the work needed to convert between bytes and coefficients. Seed-based uniform polynomial generation combines pseudorandom expansion with coefficient parsing. Polynomial multiplication requires modular products, reductions, and data movement.

Existing designs provide several approaches to these tasks. ML-KEM uses fixed-width fields and rejection sampling [Nat24, BDK+18]. NTRU Prime and BAT use compact exact

encodings [BBC+20, FKPY22]; DAWN uses a radix-769 public-key encoding with five coefficients per 48-bit block [LZL+25]. Saber uses power-of-two moduli, which simplify reduction and avoid rejection in uniform sampling [DKSRV18]. These are established design choices. Their encoded lengths and acceptance probabilities describe only part of their implementation cost. We study how the structure of small moduli can reduce the implementation cost of these three components.

Small moduli offer implementation opportunities beyond shorter coefficients. At $q = 257$, a byte represents every coefficient except for the ambiguity between 0 and 256. At $q = 769$, five coefficients fit into a 48-bit radix integer. These structures can replace general bit extraction or division with byte operations and bounded quotient calculations. Arithmetic offers a related opportunity: when intermediate ranges are known, a multiplication pipeline can retain wider representatives and omit corrections that its next stage does not require.

Three questions guide the study. How can exact compact representations reduce conversion work while making their encoded length and accepted domain explicit? How can coefficient parsing reduce extraction cost or random input demand, and how do these savings depend on SHAKE requests and parallelism? How can residual formula selection and, within a fixed formula, reduction and register scheduling lower the cost of complete polynomial multiplication? We measure targeted operations and complete calls to relate these choices to component costs. We then integrate the encoding and arithmetic components into BAT [FKPY22] and END [LZL+26]—an NTRU-with-Trapdoor instantiation with free candidate localization (FCL)—and measure their effect on complete KEM operations.

We address these questions for

$$(q, n) \in \{(257, 256), (257, 512), (257, 1024), (769, 1024)\}$$

in $\mathbb{Z}_q[x]/(x^n + 1)$. Both moduli support seven native negacyclic NTT layers, leaving residual degrees $B = 2, 4, 8$ at the studied dimensions. The rings $(257, 512)$ and $(769, 1024)$ occur in BAT, and the latter also occurs in DAWN [FKPY22, LZL+25]; the other dimensions expose how costs change with the residual degree. These are component profiles for exact serialization, public expansion, and polynomial multiplication.

1.1 Contributions

Exact compact representations and checked conversion. At $q = 257$, ZS encoding uses the low-byte payload to locate ambiguous coefficients and stores their selectors in a bounded tail. We specify the accepted domain, canonical decoding, capacity selection, and retry behavior, and combine validation with conversion in AVX2/BMI2. At $q = 769$, bounded quotient and remainder extraction moves most radix decoding to eight 32-bit lanes while preserving DAWN's format.

Structured parsing and complete generation cost. We specialize rejection sampling to the two moduli. Word16 uses byte subtraction to reduce accepted 16-bit candidates modulo 257. Grouped48 extracts five coefficients modulo 769 from each accepted 48-bit candidate. We evaluate parsing together with SHAKE expansion under different buffering and parallelism choices. For four-polynomial generation at $(q, n) = (769, 1024)$, the reduction relative to direct 10-bit rejection decreases from 13.21% with sequential SHAKE128 to 7.14% with SHAKE128x4 when the parsers and buffering policy are fixed.

Range analysis and residual scheduling. We establish bounds for forward transforms, residual products, and inverse transforms. These bounds justify omitting strict centering corrections in the recursive $q = 769$, $B = 8$ kernel, reducing complete AVX2 multiplication

cost by 4.29%. For the fixed $B = 8$ pair formula, retaining diagonal vectors in registers and reusing the batch code reduces complete cost by 5.23–5.58% relative to fully unrolled assembly. Comparisons with BAT and END include the conversions required for canonical inputs and centered outputs.

Integration in complete KEMs. We integrate the public key codecs and arithmetic into BAT and END and measure complete KEM calls. BAT uses our complete product in encryption and reencryption. END retains its NTT/INTT and transform reuse and replaces the supported residual products. The results show substantial encapsulation gains in both schemes, smaller decapsulation gains, and little change in key generation. The selector capacity trades key size against overflow margin: at the 2^{-128} uniform-reference capacities of our main configurations, END's public keys grow by five or six bytes while BAT's do not increase; engineering capacities instead make the ZS-encoded keys no longer than the originals—up to six bytes shorter—at the price of measurable key-generation retries at $n = 1024$ (the capacity sweep in Section 6.4). The experiments also identify an integration limit: END AVX2 gains mainly come from the codec, while residual replacement alone increases decapsulation cost.

1.2 Relation to Prior Work

BAT and NTRU Prime use compact encodings [FKPY22, BBC+20], and CTRU/CNTR also studies dense representations [LFZZ22]. END provides a total CRT-based format at $q = 257$ [LZL+26]. ZS is related to exception-based compression. PFOR [ZHNB06] links exception positions through offsets stored in the code stream and stores exception values separately. In ZS, zero payload bytes directly identify all positions whose values may be either 0 or 256. Each such position requires one selector bit. A fixed selector capacity determines both the encoded length and the accepted domain. We specify canonical decoding and analyze overflow and rejection sampling for this format. DAWN specifies five-coefficient radix-769 encoding [LZL+25]. Our implementation preserves this format and uses bounded intermediate values to reduce conversion work.

Rejection sampling and unbiased bounded-integer generation are established techniques [Nat24, Lem19]. Our sampling rules specialize coefficient extraction to the two moduli. We compare their complete costs under specified buffering policies and isolate the effect of SHAKE parallelism by fixing the parsers and buffering. Each rule produces uniform coefficients from independent uniform input bits and defines its own deterministic seed mapping.

Our arithmetic builds on signed Montgomery reduction, vectorized NTTs, and incomplete transforms [Mon85, Sei18, CHK+21, ABCG20]. We use established Karatsuba identities [Kar63]. Prior work combines NTT correctness with verification of machine range [HLS+22] and exploits range bounds in Plantard arithmetic [Pla21, HZZ+22]. Our range proofs justify omitting centering corrections in the signed Montgomery kernels. END supplies related kernels at $n = 512, 1024$, with $q = 769$ as an auxiliary modulus [LZL+26], enabling comparisons of complete multiplication with closely related parameters.

2 Parameters and Component Interfaces

Notations. Let q be an odd prime and n a power of two. We identify a vector $\mathbf{a} = (a_0, \ldots, a_{n-1}) \in \mathbb{Z}_q^n$ with $a(x) = \sum_i a_i x^i$ in $\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n + 1)$. Canonical representatives lie in $[0, q-1]$, and ctr_q returns the representative in $[-(q-1)/2, (q-1)/2]$. For a positive integer m, let $v_2(m)$ denote the largest nonnegative integer t such that 2^t divides m.

Incomplete NTT. Both moduli have $v_2(q-1) = 8$, so the studied negacyclic rings admit seven native radix-two layers. With $M = 128$, $B = n/M$, and a primitive 256th root ξ, put $\omega_j = \xi^{2j+1}$. The transform implements $\mathcal{R}_q \cong \prod_{j=0}^{127} \mathbb{Z}_q[x]/(x^B - \omega_j)$. Multiplication consists of two forward transforms, 128 residual products, and one inverse transform. We use $(q, n, B) = (257, 256, 2), (257, 512, 4), (257, 1024, 8), (769, 1024, 8)$ throughout.

Montgomery convention. For $R = 2^{16}$ and signed $u \equiv tq^{-1} \pmod{R}$, $-R/2 \le u < R/2$, define $\mathsf{M}_q(t) = (t - uq)/R \equiv tR^{-1} \pmod{q}$. Stored twiddles $\omega R \bmod q$ preserve the operand scale. Residual products introduce R^{-1}, compensated by the final inverse scaling. Section 5.2 and Appendix A establish the required machine bounds.

Representation. A format maps canonical vectors to fixed length strings or failure $\perp$, with $\mathsf{Unpack}(\mathsf{Pack}(\mathbf{a})) = \mathbf{a}$ whenever packing succeeds. Total encoders accept every canonical vector; partial encoders specify a subset and overflow condition. Canonical decoders accept exactly the encoder's image. Checked packing validates coefficients; checked unpacking validates length and canonicality. Conversion timings exclude candidate generation and retries.

Uniform polynomial generation. A generator first expands a seed into pseudorandom bytes and then applies rejection sampling to obtain a coefficient vector in $\mathbb{Z}_q^n$. The sampling rule yields an exactly uniform vector under independent uniform input bits. Parser timing measures the conversion of prepared bytes into coefficients, including rejection sampling. Complete-generation timing covers the full process from the seed to the resulting polynomial.

Arithmetic and integration. Complete multiplication maps canonical inputs to centered representatives of $ab \bmod (x^n + 1, q)$. Direct timings include two forward transforms, residual multiplication, one inverse transform, and all required copies, reductions, layout conversions, and output centering. Standalone comparisons use each backend's own transforms and residual kernel. Residual timings cover all 128 products on prepared transform outputs. Section 6.4 evaluates public-key codec replacement in BAT and END, together with complete-product replacement in BAT and residual replacement in END.

3 Compact Exact Representations

We use the modulus structure to reduce conversion work while retaining exact encoding and decoding. At $q = 257$, a bounded zero-selector (ZS) repairs byte ambiguities. At $q = 769$, bounded quotient extraction accelerates DAWN's radix format. Fixed-width, BAT, and END formats provide comparisons in encoded length, accepted domain, and checked conversion cost.

3.1 Total Encoding Baselines

Fixed-width encoding uses $w_q = \lceil \log_2 q \rceil$ bits per coefficient, against the total-format lower bound $\lceil n \log_2 q \rceil$ bits. C processes eight fields with shifts and masks; AVX2 processes sixteen with multiply-adds for packing and byte shuffles for extraction.

BAT stores low bits directly and jointly encodes high digits [FKPY22, Por21]. At $q = 257$, eight four-bit low parts and base-17 high digits occupy $32 + 33 = 65$ bits. At $q = 769$, five three-bit low parts and base-97 high digits occupy $15+33 = 48$ bits; remaining coefficients use ten-bit fields. Checked decoding verifies coefficient ranges and canonical padding. END uses CRT residues modulo 86 and 3 [LZL+26] to encode every canonical

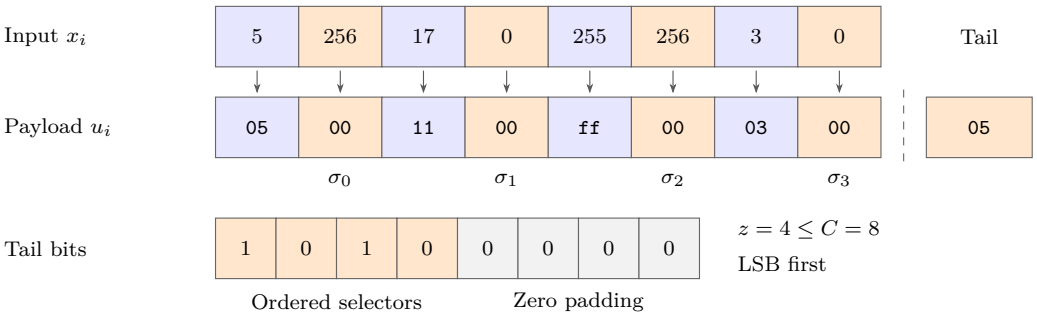
$q = 257$ vector in 514/1027 bytes at $n = 512/1024$. ZS instead has a capacity-dependent domain.

3.2 The $q = 257$ Zero-Selector

For canonical $\mathbf{x} \in \mathbb{Z}_{257}^n$, store the low bytes $u_i = x_i \bmod 256$ in coefficient order. A nonzero byte determines $x_i = u_i$; a zero byte represents either 0 or 256. If $i_0 < \cdots < i_{z-1}$ are the zero-byte positions, the selectors $\sigma_j = x_{i_j}/256 \in \{0, 1\}$ resolve these ambiguities without storing a position list.

For capacity C bits, packing returns $\perp$ when $z > C$. Otherwise it appends $\lceil C/8 \rceil$ selector bytes to the n-byte payload. Selectors follow zero-position order and fill each byte from least to most significant bit; unused and byte-alignment bits are zero. The length is $L_{\mathsf{ZS}}(n, C) = n + \lceil C/8 \rceil$ bytes. Decoding accepts exactly this length, at most C zero payload bytes, and zero unused selector bits. It restores $x_{i_j} = 256\sigma_j$ and uses $x_i = u_i$ elsewhere. Figure 1(a) gives an example.

(a) Exact encoding: low-byte payload and a bounded selector tail



(b) AVX2 + BMI2: compact selectors and scatter the repair mask

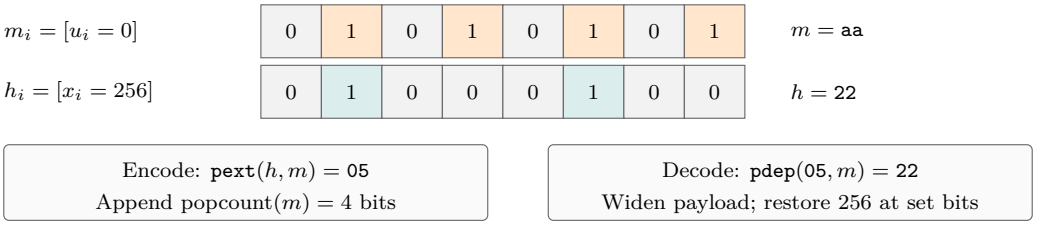


Figure 1: ZS layout and selector processing. (a) At $n = 8, C = 8$, four ambiguous positions yield tail byte 05. (b) The same eight positions illustrate a 32-coefficient AVX2/BMI2 block; the remaining payload bytes are nonzero. Payload bytes, the tail byte, and packed masks are hexadecimal. Bit rows run from low to high index.

Lemma 1 (Exactness and canonicality)**.** *Packing followed by unpacking recovers every encodable vector. Unpacking followed by packing reproduces every accepted string.*

Proof. Nonzero payload bytes recover their coefficients directly; zero bytes recover 0 or 256 from the same ordered selectors. Re-encoding reproduces the payload, used selectors, and zero padding, so accepted encodings are unique. □

Table 1: ZS capacities for uniform coefficients. Entries give selector bits C/total bytes. C_λ meets a uniform-reference $2^{-\lambda}$ overflow target.

n	Engineering	$p_\perp$ (engineering)	C_{64}/bytes	C_{128}/bytes
256	8/257	$2.12 \cdot 10^{-4}$	25/260	40/261
512	16/514	$9.18 \cdot 10^{-7}$	33/517	51/519
1024	16/1026	$3.44 \cdot 10^{-3}$	45/1030	67/1033

3.2.1 Capacity and public-key pseudorandomness

Let $Z(\mathbf{x}) = |\{i : x_i \in \{0, 256\}\}|$ and $E_C = \{\mathbf{x} : Z(\mathbf{x}) \le C\}$. For uniform $\mathbf{v} \in \mathbb{Z}_{257}^n$, $Z(\mathbf{v}) \sim \mathrm{Bin}(n, 2/257)$, hence

$$p_\perp(n, C) = \Pr[Z(\mathbf{v}) > C] = \sum_{j=C+1}^{n} \binom{n}{j} \left(\frac{2}{257}\right)^j \left(\frac{255}{257}\right)^{n-j}. \tag{1}$$

Define $C_\lambda(n) = \min\{C : p_\perp(n, C) \le 2^{-\lambda}\}$. Table 1 gives these uniform-reference capacities and the smallest whole byte capacities meeting a 1% overflow budget, called engineering capacities.

The probabilities $p_\perp(n, C)$ above are exact for uniform inputs. For an actual public key candidate $\mathbf{b}$, let δ_C denote the absolute difference between its overflow probability and $p_\perp(n, C)$. This difference is the distinguishing advantage of the $O(n)$-time test $D_C(\mathbf{x}) = \mathbf{1}[Z(\mathbf{x}) > C]$ between the candidate distribution and the uniform distribution. Therefore,

$$\Pr[Z(\mathbf{b}) > C] \le p_\perp(n, C) + \delta_C. \tag{2}$$

At capacity C_λ, this gives an overflow bound of $2^{-\lambda} + \delta_{C_\lambda}$. The probability is averaged over the randomness used to generate a candidate before the ZS acceptance test.

For uniform a and $b = as + e$, the relevant pseudorandomness assumption is decisional RLWE for the specified ring, modulus, and exact secret and error distributions [LPR10, LPR13]. For $a = \mathrm{Expand}(\rho)$ with a public seed ρ, the corresponding comparison is between (ρ, a, b) and (ρ, a, v) under the model used for Expand, where v is uniform and independent of (ρ, a). Any distinguishing-advantage bound for the applicable joint distributions also bounds δ_C, since D_C uses only the candidate vector.

Proposition 1 (ZS overflow for NTRU public keys)**.** *Fix $q = 257$ and a scheme, parameter set, and implementation θ. Let $\mathcal{D}_\theta$ be the joint distribution of (f, g) returned by the key generator before adding ZS rejection, including all its original sampling and rejection conditions. Let $h = gf^{-1} \in R_q$, where f is invertible, and let $\Phi_\theta : R_q \to \mathbb{Z}_q^n$ be the bijective representation supplied to ZS, with canonical entries. Define*

$$A_{\theta,C}(u) = D_C(\Phi_\theta(u)), \qquad \varepsilon_\theta(C) = \Pr_{(f,g)\leftarrow\mathcal{D}_\theta}[A_{\theta,C}(gf^{-1}) = 1].$$

For a reference distribution Q on R_q, define

$$\mathrm{Adv}_{\theta,Q}^{\text{D-NTRU}}(A) = \Big| \Pr_{(f,g)\leftarrow\mathcal{D}_\theta}[A(gf^{-1}) = 1] - \Pr_{u\leftarrow Q}[A(u) = 1] \Big|.$$

Writing $U(S)$ for the uniform distribution on S and $\alpha_{n,q} = |R_q^\times|/|R_q|$, we have

$$\varepsilon_\theta(C) \le p_\perp(n, C) + \mathrm{Adv}_{\theta,U(R_q)}^{\text{D-NTRU}}(A_{\theta,C}), \tag{3}$$

$$\varepsilon_\theta(C) \le \frac{p_\perp(n, C)}{\alpha_{n,q}} + \mathrm{Adv}_{\theta,U(R_q^\times)}^{\text{D-NTRU}}(A_{\theta,C}). \tag{4}$$

Proof. Apply the definition of distinguishing advantage to $A_{\theta,C}$. Since Φ_θ is bijective, a uniform element of R_q maps to a uniform vector in $\mathbb{Z}_{257}^n$, whose overflow probability is $p_\perp(n, C)$. The uniform distribution on $R_q^\times$ is the uniform distribution on R_q conditioned on invertibility. This event has probability $\alpha_{n,q}$, so the conditional overflow probability is at most $p_\perp(n, C)/\alpha_{n,q}$. □

For the full-ring reference, the advantage term in (3) equals δ_C for the candidate vector $\Phi_\theta(h)$. BAT uses the uniform-invertible reference [FKPY22], corresponding to (4). In either case, an applicable advantage bound must concern $\mathcal{D}_\theta$, including its original rejection conditions. The test $A_{\theta,C}$ takes $T_{\Phi_\theta} + O(n)$ time. For $n \in \{512, 1024\}$, the polynomial $x^n + 1$ has 128 distinct irreducible factors of degree $n/128$ over $\mathbb{Z}_{257}$. Hence $\alpha_{n,257} = \left(1 - 257^{-n/128}\right)^{128}$.

3.2.2 Retries and integration

Let $a, s, e \in R_q$ and $b = as+e$. Write $\mathbf{b} = (b_0, \ldots, b_{n-1})$ for the canonical coefficient vector of b, with $b_i \in \{0, \ldots, q-1\}$. With a fixed, successive attempts draw independent pairs (s, e) from the same distribution, compute $b = as + e$, and encode its canonical coefficient vector $\mathbf{b}$. Overflowing candidates and partial encodings are discarded. On success, the matching secret material is retained.

Proposition 2 (Effect of retries)**.** *Let P be the distribution of a candidate vector, with $\varepsilon = P(E_C^{\mathrm{c}}) < 1$. For independent draws from P, let A count the attempts up to and including the first success. Then*

$$\mathbb{E}[A] = (1-\varepsilon)^{-1}, \qquad P_{\mathrm{acc}} = P(\cdot \mid E_C), \qquad \Delta(P_{\mathrm{acc}}, P) = \varepsilon,$$

where Δ denotes total variation distance.

Proof. The attempt count is geometric, and acceptance conditions P on E_C. The absolute probability differences on E_C and E_C^{c} each sum to ε. Thus, $\Delta(P_{\mathrm{acc}}, P) = \frac{1}{2}(\varepsilon+\varepsilon) = \varepsilon$. □

When a is retained, let P_a be the conditional distribution of $\mathbf{b}$ given a, and write $\varepsilon_a = P_a(E_C^{\mathrm{c}})$. Assume $\varepsilon_a < 1$ for every a with positive generation probability. Unlimited retries preserve the marginal distribution of a and replace each P_a by $P_a(\cdot \mid E_C)$. This generally differs from conditioning the original joint distribution on acceptance. The expected number of attempts is $\mathbb{E}[A] = \mathbb{E}_a\left[(1-\varepsilon_a)^{-1}\right]$. Resampling a, s, e on every attempt instead gives $\mathbb{E}[A] = (1-\bar{\varepsilon})^{-1}$, where $\bar{\varepsilon} = \mathbb{E}_a[\varepsilon_a] \le p_\perp(n, C) + \delta_C$. A bound on $\bar{\varepsilon}$ alone does not, in general, bound $\mathbb{E}_a[(1-\varepsilon_a)^{-1}]$ from above.

With a retained and at most r total attempts, the conditional failure probability is ε_a^r, and the average is $\mathbb{E}_a[\varepsilon_a^r]$. Among successful outputs, the marginal probability of a is proportional to its original probability times $1 - \varepsilon_a^r$. An integration specifies which randomness is refreshed, the attempt limit, and the action on failure. These failure probabilities concern encoding overflow during key generation.

Encoding and overflow handling. Engineering capacities are intended for public data. Since ZS is invertible on E_C, an accepted vector's selector count and positions are determined by its published encoding. For i.i.d. retries at fixed a with positive acceptance probability, let $X_i = (s_i, e_i, \mathbf{b}_i)$, let A be the first successful attempt, and let $H = (X_1, \ldots, X_{A-1})$. For any set S and history h with $\Pr[H = h \mid a] > 0$,

$$\Pr[X_A \in S \mid a, H = h] = \Pr[X_1 \in S \mid a, \mathbf{b}_1 \in E_C].$$

Hence H and X_A are independent given a. This independence persists when conditioning on $A \le r$ for a fixed attempt limit r, since $A = |H| + 1$. Observations determined solely by

$(a, H, \mathbf{b}_A)$, including retry counts and selector positions, therefore reveal no information about (s_A, e_A) beyond the public $(a, \mathbf{b}_A)$. For secret inputs, encoding, decoding, and overflow handling must be constant time. The capacity must cover all possible secret inputs or meet a specified negligible bound on the probability of any overflow across the intended invocations, under the actual secret distribution.

3.2.3 C and AVX2 conversion

Both implementations separate regular payload processing from selector extraction. The C payload loop can therefore vectorize without a selector-index dependency. For a little-endian 64-bit payload word v and $\kappa = \texttt{0x7f7f7f7f7f7f7f7f}$, the mask

$$m = \mathrm{zmask}(v) = \sim\big(((v \,\&\, \kappa) + \kappa) \mid v \mid \kappa\big) \tag{5}$$

marks exactly the zero bytes with `0x80`. No carry crosses a byte boundary. Repeated trailing-zero extraction and $m \leftarrow m \,\&\, (m-1)$ enumerate the marked positions; the decoder uses the same order after widening the payload. Successful conversion costs $\Theta(n) + O(\lceil C/8 \rceil)$.

AVX2 processes 32 coefficients per iteration. Byte comparisons and mask extraction locate the ambiguities directly from the payload; BMI2 `pext` collects their selectors and `pdep` scatters the repair bits during decoding (Figure 1(b)). This removes a separate position stream while keeping the payload loop regular. Fused checked variants validate coefficient ranges or selector capacity during conversion and check unused tail bits; 64-bit selector refills stay within the allocated tail. The measured AVX2 checked paths use these variants. C also retains paths with a separate validation pass, selected by operation and capacity. Thus the capacity comparison in Section 6.1 evaluates the corresponding implementations.

3.3 Radix-769 Encoding

DAWN's format maps five coefficients to $w = \sum_{i=0}^{4} a_i 769^i < 769^5 = 268925323054849 < 2^{48}$ and allocates six bytes [LZL+25, Section 6.2]. At $n = 1024$, 204 full groups and a five-byte, four-coefficient tail use 1229 bytes, matching BAT and saving 51 bytes over fixed width. Four full radix integers occupy three little-endian 64-bit words (Figure 2(a)): the first three fill the low 48 bits, and successive 16-bit pieces of the fourth fill the high bits.

For this encoding method, we have developed our own implementations for C and AVX2. The C encoder uses constant multiply-adds. AVX2 constructs four integers in parallel from

$$w_L = a_0 + 769a_1 + 769^2 a_2 < 2^{29}, \quad w_H = a_3 + 769a_4 < 2^{20}, \quad w = w_L + 769^3 w_H.$$

The final product uses unsigned 32-by-32-bit multiplication with a 64-bit result. Decoding splits $h = \lfloor w/769^2 \rfloor$ and $r = w - 769^2 h$. Since $h < 2^{29}$ and $r < 2^{20}$, remaining digit extraction uses 32-bit operands. C uses separate quotient/remainder chains for h and r; AVX2 combines two four-lane splits into eight lanes (Figure 2(b)). Quotients use reciprocal multiplication with masked correction. Checked decoding requires $w < 769^5$ and a tail below 769^4; checked encoding verifies coefficients in $[0, 768]$. The scalar tail reads or writes exactly five bytes. The split preserves DAWN's representation while moving most extraction to narrower operands. Section 6.1 evaluates whether this conversion strategy retains the size benefit at a practical cost.

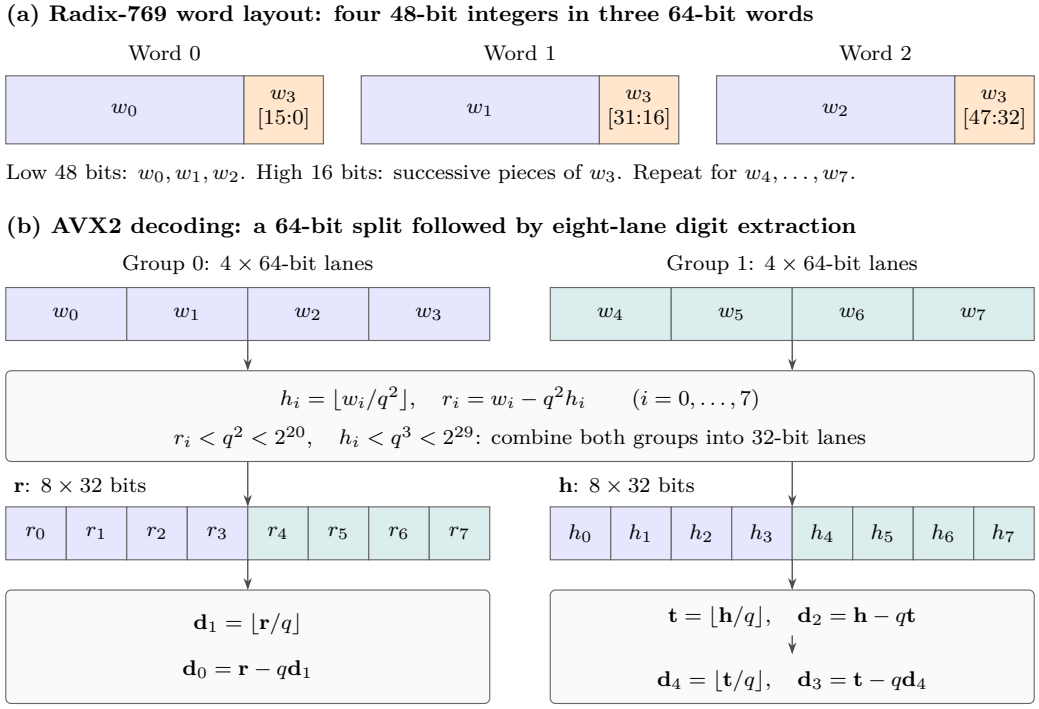


Figure 2: Radix-769 storage and AVX2 decoding. (a) Little-endian words with bit positions increasing left to right. (b) 256-bit register rows colored by input group. Split bounds enable eight-lane 32-bit extraction, with one extra quotient dependency in the high branch. Quotients use reciprocal multiplication and masked correction.

4 Exact Uniform Polynomial Sampling

Sampling uniformly from R_q is equivalent to sampling a coefficient vector uniformly from $\mathbb{Z}_q^n$. We implement this process in two stages. A SHAKE component expands a seed into pseudorandom bytes [Nat15]. A rejection sampler then converts these bytes into n polynomial coefficients. We compare conventional direct rejection sampling from fixed-width bit strings, as used in ML-KEM [Nat24], with word-based rejection sampling for $q = 257$ and grouped rejection sampling for $q = 769$. For $q = 257$, the word-based approach reduces both the expected input demand and parser cost. For $q = 769$, the grouped approach requires radix conversion but consumes fewer input bits on average than direct rejection. Although parser cost may be higher, the reduced demand for pseudorandom bytes can lower the overall generation cost.

4.1 $q = 257$: Word-Based Rejection Sampling

Direct rejection sampling draws a uniform 9-bit integer x and accepts it when $x < 257$. The accepted value is uniform in $\mathbb{Z}_{257}$, with acceptance probability $257/512$ and expected input demand $9 \cdot 512/257 \approx 17.930$ bits per coefficient. The word-based method instead draws a uniform 16-bit integer x and rejects only $x = 65535$. The accepted interval contains $65535 = 255 \cdot 257$ consecutive integers, giving each residue modulo 257 exactly 255 preimages. Reduction modulo 257 therefore yields an exactly uniform coefficient.

Writing $x = x_0 + 256x_1$, where $0 \le x_0, x_1 < 256$, and using $256 \equiv -1 \pmod{257}$ gives

$$x \bmod 257 = (x_0 - x_1) \bmod 257.$$

The byte difference lies in $[-255, 255]$, so adding 257 when it is negative yields the canonical representative. The acceptance probability is $1 - 2^{-16}$, reducing the expected input demand to $16/(1 - 2^{-16}) \approx 16.0002$ bits per coefficient, 10.76% below direct rejection.

The word-based method also simplifies the parser by replacing packed 9-bit field extraction with byte loads and subtraction. The AVX2 implementation performs subtraction and conditional correction in 16-bit lanes. When all candidates in a batch are accepted, the resulting coefficients are stored directly. Otherwise, only values that satisfy the acceptance condition are retained.

4.2 $q = 769$: Grouped Rejection Sampling

Direct rejection sampling draws a uniform 10-bit integer x and accepts it when $x < q = 769$. Its acceptance probability is $q/2^{10}$, giving an expected input demand of $10 \cdot 2^{10}/q \approx 13.316$ bits per coefficient. Grouped rejection instead draws a uniform 48-bit integer z and accepts it when $z < q^5$. Each accepted integer has a unique radix-q expansion $z = \sum_{j=0}^{4} a_j q^j$, for $0 \le a_j < q$. This expansion defines a bijection between $\{0, \ldots, q^5 - 1\}$ and $\mathbb{Z}_q^5$, so the coefficient vector $(a_0, \ldots, a_4)$ is jointly uniform. The acceptance probability is $p_5 = q^5/2^{48} \approx 0.9554$, giving an expected input demand of $48/(5p_5) \approx 10.048$ bits per coefficient for complete groups. For $n = 1024$, 204 groups provide 1020 coefficients. The remaining four are generated by accepting a uniform 39-bit integer below q^4 and extracting its four radix-q digits. With $p_4 = q^4/2^{39}$, the expected input demand for the full polynomial is $(204 \cdot 48/p_5 + 39/p_4)/1024 \approx 10.069$ bits per coefficient. Each tail candidate is read from five bytes with the highest bit cleared. This estimate counts candidate bits and therefore excludes byte-alignment and buffering overhead.

The C implementation of direct rejection extracts each 10-bit field from two adjacent bytes, while its AVX2 counterpart performs candidate extraction and rejection tests in parallel. Grouped rejection reduces input demand but requires radix conversion. We use the bounded decomposition from Section 3.3, where $h = \lfloor z/q^2 \rfloor$ and $r = z - q^2 h$ satisfy $h < q^3 < 2^{29}$ and $r < q^2 < 2^{20}$. These bounds allow subsequent digit extraction to use 32-bit operands, with division by constants implemented through reciprocal multiplication and correction. When all eight candidates in a batch are accepted, the AVX2 implementation extracts their coefficients in parallel. Otherwise, only candidates that satisfy the acceptance condition are retained.

4.3 Complete Generation Cost

The input demands derived above quantify the expected number of candidate bits examined by the sampling procedures. Complete generation also includes the cost of producing these bits with SHAKE. Since SHAKE generates output in blocks, fewer candidate bits do not always imply fewer permutations. The benefit of either sampling method therefore depends on both parser cost and the cost of SHAKE expansion.

Single-polynomial generation combines either the C or AVX2 parser with the same sequential SHAKE implementation [BDP+20, BSVA15]. For $q = 769$, four-polynomial generation also uses a four-stream SHAKE implementation to expand domain-separated streams in parallel [Wes20]. We compare sequential and parallel expansion using the same AVX2 parsers and buffering policy. For a fixed sampling rule and SHAKE variant, these implementations preserve the seed-to-polynomial mapping.

At $q = 769$, single-polynomial generation uses request-sized squeezing. Four-polynomial generation uses 168-byte rate buffering and retains unused output. To assess the effect

of SHAKE parallelism, we compare the sequential and four-stream implementations of the four-polynomial task with the AVX2 parsers and buffering fixed. The reported input-byte estimates describe theoretical candidate demand, while complete costs are measured directly from a seed to the polynomial output.

5 Arithmetic with Incomplete NTTs

Complete multiplication uses two seven-layer Cooley–Tukey forward transforms, 128 residual products, and one seven-layer Gentleman–Sande inverse transform. The C and AVX2 paths use their corresponding transforms throughout, 16-bit coefficient storage, and Montgomery radix $R = 2^{16}$. For each parameter set, the residual methods use common forward transforms and satisfy the corresponding layout, range, and scale requirements. We compare residual formulas, then optimize reductions and register use within a fixed formula.

5.1 Residual Formulas

In a residual ring $\mathbb{Z}_q[x]/(x^B-\omega_j)$, write the ordinary convolution of f and g as $\sum_{k=0}^{2B-2} c_k x^k$. The common folding step returns $h_k = c_k + \omega_j c_{k+B}$ with $0 \le k < B-1$ and $h_{B-1} = c_{B-1}$. It uses $B-1$ twiddle multiplications and can be interleaved with convolution accumulation. Schoolbook computes all B^2 coefficient products.

Nonrecursive symmetric-pair Karatsuba computes the diagonal products $d_i = f_i g_i$ once, then uses

$$p_{ir} = (f_i + f_r)(g_i + g_r) - d_i - d_r = f_i g_r + f_r g_i \qquad (i < r).$$

Each p_{ir} contributes to c_{i+r}, and each d_i to c_{2i}. The B diagonals are reused across all pairs, giving $B + \binom{B}{2} = B(B+1)/2$ convolution multiplications.

Recursive Karatsuba instead splits both polynomials in half and computes three half-size products at each level. Its count satisfies $P(B) = 3P(B/2)$, $P(1) = 1$. We expand the recursion at source level for $B = 2, 4, 8$, avoiding runtime recursive calls.

Including folding, the multiplications in $\mathbb{Z}_q$ at $B = 2, 4, 8$ are respectively $5, 19, 71$ for schoolbook, $4, 13, 43$ for pair, and $4, 12, 34$ for recursion. At $B = 2$ the Karatsuba formulas coincide. Larger recursive kernels save products but increase operand sums and recombination dependencies; pair kernels instead reuse diagonal intermediates. Both effects influence reduction and register costs.

5.2 Ranges and Montgomery Scale

Our signed Montgomery reduction satisfies

$$\mathsf{M}_q(t) = (t - uq)/R \equiv tR^{-1} \pmod q, \qquad |\mathsf{M}_q(t)| \le \mu_q(|t|) := \lfloor |t|/R + q/2 \rfloor .$$

The numerator $t - uq$ fits signed 32-bit arithmetic when $|t| + Rq/2 < 2^{31}$. Appendix A proves the bounds below, including intermediate sums, integer width bounds for the Montgomery correction, and the final scale. It permits wider signed representatives when their subsequent sums and products remain within the machine ranges.

Forward and residual stages. For canonical inputs and stored twiddles $z \in [0, q-1]$ representing ωR, a forward butterfly maps (x, y) to $(x + \mathsf{M}_q(zy), x - \mathsf{M}_q(zy))$. The recurrence for coefficient magnitude bounds $F_0 = q - 1$ and $F_{s+1} = F_s + \mu_q((q-1)F_s)$ gives $F_7 = 1171$ and 3618 at $q = 257$ and 769, respectively. No additional coefficient reduction is required between forward layers. Schoolbook and pair kernels accept these outputs directly. Recursive Karatsuba also does so at $q = 257$. But at $q = 769$, the recursive kernel

Table 2: Magnitude bounds for forward outputs and folded residual outputs. Recursive inputs are unreduced at $q = 257$; the two $q = 769$ entries use strict and lazy input reduction, respectively.

q	B	F_7	Schoolbook	Pair	Recursive
257	2	1171	298	510	510
257	4	1171	596	1020	1907
257	8	1171	1192	2040	7981
769	8	3618	4672	9404	10852/10946

reduces its inputs so that the subsequent recombinations fit signed 16-bit lanes and the folded outputs satisfy the inverse input bound.

Strict reduction centers each $q = 769$ input in $[-384, 384]$. Our lazy recursive variant instead retains

$$x' = x - 769 \left\lfloor \frac{43x + 2^{14}}{2^{15}} \right\rfloor, \qquad |x| \le 3618.$$

This preserves the residue and guarantees $|x'| \le 417$. The resulting output bound 10946 satisfies the inverse input limit 16383. The implementation therefore retains the approximate quotient and subtraction, and omits the two masked corrections that would enforce strict centering. Table 2 summarizes the bounds used by the pipeline.

Inverse schedule and scaling. The inverse butterfly maps (x, y) to $(x+y, \mathsf{M}_q(z(x-y)))$. The input limit keeps the first sums and differences within $[-32766, 32766]$. Coefficient reductions after this first logical layer allow the remaining six layers to run without further coefficient centering. Their magnitude bounds reach 8192 at $q = 257$ and 24576 at $q = 769$, within signed 16-bit range.

Montgomery-scaled twiddles preserve their operand scale, whereas residual coefficient products introduce R^{-1}. The final inverse multiplication uses $\gamma_q = R^2\, 128^{-1} \bmod q$ to cancel that factor and normalize the seven layers. Applying the Montgomery bound to the final scaling gives $|x| \le \mu_{257}(8192 \cdot 255) = 160$ and $|x| \le \mu_{769}(24576 \cdot 655) = 630$, respectively. Since $|x| < q$ in both cases, at most one conditional addition or subtraction of q maps each output to $\mathrm{ctr}_q(x) \in [-(q-1)/2, (q-1)/2]$. The complete-call wrapper performs this centering.

5.3 Implementation and Register Scheduling

The residual layout places corresponding coefficients from 16 rings contiguously. In AVX2, each vector holds one coefficient position in 16 signed 16-bit lanes, so lane-wise operations process 16 rings in parallel. C loops over the rings, allowing compiler vectorization, and uses fixed layout permutations between transforms and residual kernels. AVX2 uses the transform layouts directly.

All kernels use signed Montgomery reduction with $R = 2^{16}$. For a coefficient product $t = ab$, let u be the signed 16-bit representative of $tq^{-1} \bmod R$. The generic AVX2 implementation computes u by multiplying the low 16 bits of t by q^{-1} and retaining the low 16 bits [Sei18]. Since t and uq have identical low 16 bits, $\mathsf{M}_q(t) = \lfloor \frac{t}{R} \rfloor - \lfloor \frac{uq}{R} \rfloor$. Signed high-half multiplication instructions obtain the two terms, and a 16-bit subtraction produces the result. The established range bounds ensure that this difference fits a signed 16-bit lane. Each 256-bit vector therefore processes 16 independent Montgomery multiplications without widening the lanes. The C implementation evaluates the same reduction using 32-bit intermediate products and arithmetic.

For the $B = 8$ nonrecursive symmetric-pair method, we compare an intrinsic implementation with fully unrolled and register-scheduled assembly implementations, as illustrated

in Figure 3.

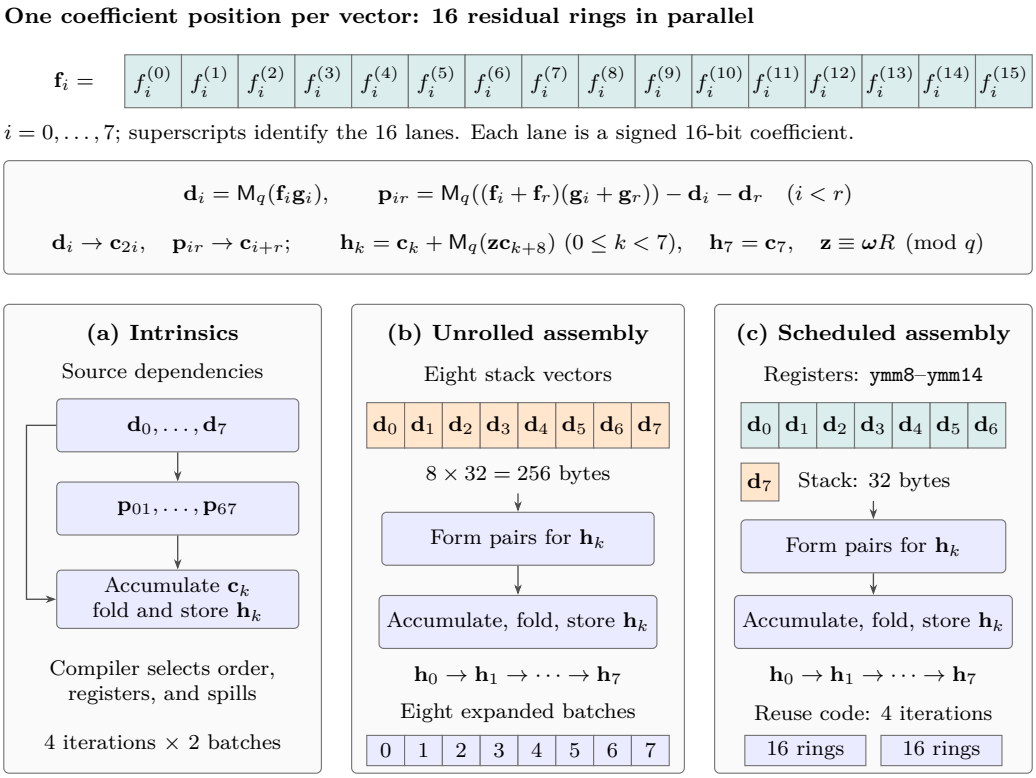


Figure 3: The scheduling of $B = 8$ pair method with generic Montgomery reduction. Panel (a) shows source dependencies; the compiler selects the instruction schedule and storage. Both assembly kernels in panels (b) and (c) store one output vector at a time. Orange and teal mark stack and register storage; byte counts refer only to diagonal storage. In panel (c), each scheduled iteration processes two batches of 16 rings.

The intrinsic implementation leaves instruction scheduling and register allocation to the compiler. Both assembly implementations accumulate one output coefficient vector at a time and immediately fold and store it. The unrolled version stores all eight diagonal vectors on the stack and expands all residual batches. The scheduled version retains seven diagonal vectors in registers and one on the stack. It reuses the batch code across four iterations, each processing two batches of 16 rings. The coefficient products are unchanged; the schedule depends on the fixed $B = 8$ layout and available registers. Additional C and intrinsic candidates compute and fold one output coefficient at a time, leaving register allocation to the compiler. They provide the corresponding output-oriented schedule without manual register assignment.

Section 6.3 first compares formulas under common transforms, then holds a formula fixed to evaluate reduction and scheduling changes. The range proof justifies deleting corrections; the assembly comparison measures the combined effect of register retention and batch-code reuse.

6 Evaluation

We evaluate the methods of Sections 3–5: encoding and decoding, uniform polynomial generation, and polynomial multiplication. We also measure the effect of encoding and

Table 3: Encoded size and median cycles of encoding and decoding. ZS uses selector capacities $C = 8, 16, 16$ bits in increasing dimension order.

			C		AVX2	
(q, n)	Format	Bytes	Enc.	Dec.	Enc.	Dec.
$(257, 256)$	Fixed width	288	345	379	235	171
	BAT	260	760	2300	296	590
	ZS (eng.)	257	238	256	98	93
$(257, 512)$	Fixed width	576	628	700	443	324
	BAT	521	1436	4578	535	1110
	END	514	5729	8237	2727	8105
	ZS (eng.)	514	420	475	140	131
$(257, 1024)$	Fixed width	1152	1124	1346	923	557
	BAT	1040	2816	9103	1042	2203
	END	1027	11351	15977	11351	15977
	ZS (eng.)	1026	907	906	231	199
$(769, 1024)$	Fixed width	1280	1197	1244	924	563
	BAT	1229	3379	10291	1803	2754
	Direct radix	1229	1118	2356	779	2055

Note. When $n = 1024$, END uses C code for encoding and decoding in both its C and AVX2 implementations. The same timings are reported in both columns.

Table 4: ZS capacity, encoded size, and median cycles of checked conversion at $q = 257, n = 1024$. Overflow targets refer to independent uniform coefficients.

			C		AVX2	
C (bits)	Encoded Bytes	Overflow bound	Enc.	Dec.	Enc.	Dec.
16	1026	10^{-2}	907	906	231	199
45	1030	2^{-64}	834	845	236	212
67	1033	2^{-128}	838	852	234	213

arithmetic replacements on complete BAT and END KEM operations. Experiments ran on an Intel Core i5-11500, Ubuntu 22.04, and Clang 14.0.0 with SMT and Turbo disabled. C uses `-O3 -march=x86-64-v3 -mtune=native` with automatic vectorization; explicit SIMD uses AVX2, with BMI2 for selectors. We measure time using fenced `RDTSCP` and report TSC ticks as cycles throughout this section. Component timings are rounded medians. Their percentage reductions are computed from unrounded medians as $100(1 - T_{\text{candidate}}/T_{\text{baseline}})$. We retain pinned BAT [Por21] and END [LZL+26] sources, licenses, and adapters. Parser backends share sequential SHAKE; SHAKE128x4 is described in Section 4.3.

6.1 Exact Encoding and Conversion Cost

Table 3 compares size and cycles of conversions. At $q = 257$, every format uses the same uniform inputs conditioned on ZS success at the engineering capacity. ZS combines the smallest tested engineering lengths with the lowest checked conversion costs in all compared methods. Its main benefit is conversion speed: at $n = 1024$, checked AVX2 decoding costs 199 cycles versus 2203 for BAT. The payload supplies both coefficients and repair positions, avoiding the multi-digit reconstruction of dense radix or CRT formats.

Table 4 shows that fast conversion persists at larger capacities. For $n = 1024$, the 16-bit selector has overflow probability 0.003444; targets 2^{-64} and 2^{-128} require 45 and 67 bits selector, corresponding to 1030 and 1033 encoded bytes. The 67-bit configuration is six bytes longer than END's total format, yet checked AVX2 decoding remains at 213 cycles. Thus ZS offers a capacity–length–speed choice, including configurations with much smaller reference overflow than the engineering setting.

Table 5: Uniform polynomial generation at $q = 257$: cycles for parsing and complete generation, and theoretical expected input bytes.

n	Parser Impl.	Parsing			Complete			Input (bytes)		
		Direct	Word	Reduction	Direct	Word	Reduction	Direct	Word	Reduction
256	C	1895	1354	28.55%	8122	7593	6.51%	574	512	10.80%
256	AVX2	939	396	57.83%	7159	6663	6.93%	574	512	10.80%
512	C	3590	2490	30.64%	14228	13095	7.96%	1148	1024	10.80%
512	AVX2	1530	607	60.33%	12214	11250	7.89%	1148	1024	10.80%
1024	C	7012	4764	32.06%	27830	24127	13.31%	2295	2048	10.76%
1024	AVX2	2732	954	65.08%	23658	20314	14.13%	2295	2048	10.76%

At $q = 769$, direct radix retains BAT's 1229-byte length and lowers encoding and decoding costs in C and AVX2; AVX2 encoding falls from 1803 to 779 cycles. Bounded extraction thus makes compact conversion less costly. Fixed width still decodes faster by avoiding quotient dependencies, trading 51 additional bytes for lower decoding cost.

6.2 Parsing and Complete Uniform Polynomial Generation

We evaluate the parsing and complete generation costs of the sampling methods described in Section 4. All experiments use SHAKE128, as in the public-matrix generation of ML-KEM [Nat24]. Parser timings measure coefficient extraction and rejection sampling from prepared bytes, while complete generation timings cover the entire process from a seed to the resulting polynomial, including SHAKE128 computation.

Generation at $q = 257$. Table 5 compares conventional 9-bit direct rejection sampling with 16-bit word-based rejection sampling in terms of parsing cost, complete generation cost, and theoretical expected input bytes. The word-based method reduces both parsing and complete generation costs across all tested dimensions and backends. At $n = 256$, parsing costs decrease by 28.55% in C and 57.83% in AVX2, while at $n = 1024$, the reductions reach 32.06% and 65.08%, respectively. These improvements arise from simpler coefficient extraction using regular word loads and byte subtraction, together with the near-certain acceptance of candidates that reduces rejection-handling overhead.

The word-based method also reduces the expected input demand by approximately 10.8% across all tested dimensions. At $n = 256$, the expected demand decreases from approximately 574 to 512 bytes, while at $n = 1024$, it decreases from approximately 2295 to 2048 bytes. At smaller dimensions, the improvement in complete generation primarily comes from cheaper parsing. At larger dimensions, the reduced input demand can additionally lower SHAKE128 computation costs. These results demonstrate that word-based rejection sampling improves complete polynomial generation through both simpler coefficient extraction and lower pseudorandom input demand.

Generation at $q = 769$. Table 6 compares 10-bit direct rejection sampling with 48-bit grouped rejection sampling. For single-polynomial generation, grouped rejection increases parsing cost by 14.08% in C and 15.65% in AVX2 relative to direct 10-bit rejection. Nevertheless, complete generation cost decreases by 6.96% and 14.55%, respectively. The expected input demand falls by 24.39%. Thus, parser timings and complete-generation timings favor different sampling rules in this configuration. For four-polynomial generation, the AVX2 parsers and rate-buffering policy are fixed when comparing the SHAKE implementations. Grouped rejection reduces complete cost by 13.21% with sequential SHAKE128 and 7.14% with SHAKE128x4. The expected input reduction remains 24.39%. These results show that the benefit of lower input demand depends on the SHAKE implementation, even when parser code and buffering remain unchanged.

Table 6: Uniform polynomial generation at $q = 769, n = 1024$: cycles for parsing and complete generation, and theoretical expected input bytes. Seq. and x4 denote sequential SHAKE128 and SHAKE128x4, respectively.

Polys.	Parser Impl.	SHAKE128	Parsing			Complete			Input (bytes)		
			Direct	Grouped	Reduction	Direct	Grouped	Reduction	Direct	Grouped	Reduction
1	C	Seq.	9250	10552	-14.08%	26059	24245	6.96%	1705	1289	24.39%
1	AVX2	Seq.	2332	2697	-15.65%	19158	16371	14.55%	1705	1289	24.39%
4	C	Seq.		–		111770	96968	13.24%	6819	5156	24.39%
4	AVX2	Seq.		–		75597	65610	13.21%	6819	5156	24.39%
4	AVX2	x4		–		29346	27251	7.14%	6819	5156	24.39%

6.3 Polynomial Multiplications

Residual product method comparison. Table 7 compares the cycles of schoolbook, non-recursive symmetric pair, and recursive Karatsuba based residual multiplications. At $B = 2$ the Karatsuba formulas coincide and their costs are close. At $B = 8$, both improve on schoolbook. Recursion gives the lowest cost at $q = 257$, whereas at $q = 769$ C favors pair and AVX2 favors recursive Karatsuba with strict input. Recursion's product savings compete with larger operand sums and recombination; pair trades more products for diagonal reuse (Section 5.1). The next controls optimize these costs with the formula fixed.

Table 7: Cycles of complete polynomial multiplication with three residual formulas.

	C			AVX2		
(q, n)	Schoolbook	Pair	Recursive	Schoolbook	Pair	Recursive
$(257, 256)$	5317	5268	5274	919	915	917
$(257, 512)$	7589	7420	7232	1843	1826	1813
$(257, 1024)$	18638	17616	16881	5728	5235	4566
$(769, 1024)$	18421	17455	18063	5581	5189	4943

Implementation optimizations with fixed residual method. Table 8 compares cycle counts for the pair method using the scheduling strategies shown in Figure 3 of Section 5.3 and for the recursive method with inputs reduced to different ranges. Relative to unrolled pair, scheduled pair reduces diagonal stack writes per vector block from 8 to 1 and reads from 64 to 8. Diagonal scratch space falls from 256 to 32 bytes, and the assembly function size from 23672 to 4973 bytes. Residual costs decrease by 18.96% at $q = 257$ and 20.98% at $q = 769$; complete costs decrease by 5.23% and 5.58%, respectively. This is the gain of the combined register retention and batch reuse schedule: the same products are computed with fewer stack accesses and a smaller repeated code body.

At $q = 769$, lazy recursive inputs omit the two masked corrections used for strict centering. Appendix A proves that the subsequent products and recombinations meet the inverse-entry contract. Complete cost falls by 4.29%; the separately measured residual reduction is 21.18%. The compiled function has 64 fewer `vpcmpgtw` instructions and shrinks from 5567 to 4767 bytes. Thus a range bound removes concrete instructions while preserving the formula and output interface. At $q = 257$, recursion already accepts unreduced forward outputs, so this particular optimization applies only to $q = 769$. Residual and complete comparisons each use their own batch and scope.

Comparison with BAT and END. According to the experiments above, we select symmetric-pair for $(257, 256)$ and recursion for $(257, 512)$ and $(257, 1024)$. At $(769, 1024)$, C selects pair and AVX2 lazy recursion. All choices have the lowest median in every run except AVX2 at $(257, 1024)$, where scheduled pair wins the third run.

Table 8: AVX2 cycles for residual products and complete multiplication at $B = 8$.

	$q = 257, n = 1024$		$q = 769, n = 1024$	
Implementation	Residual	Complete	Residual	Complete
Intrinsics pair	1498	5235	1494	5189
Unrolled pair	1492	5089	1530	5075
Scheduled pair	1209	4823	1209	4792
Recursive	861	4566	1382	4943
Lazy recursive	–	–	1089	4731

The first three rows share the pair formula; the last two use recursive Karatsuba.

Table 9: Cycles of complete multiplication with strictly centered output: BAT, END and our selected implementations.

	C			AVX2		
(q, n)	BAT	END	Selected	BAT	END	Selected
$(257, 256)$	10901	–	5268	2398	–	915
$(257, 512)$	16486	11427	7232	5098	1723	1813
$(257, 1024)$	22880	40608	16881	10965	4736	4566
$(769, 1024)$	22924	38238	17455	11183	4695	4731

BAT's arithmetic implementation supports variable degrees through the `logn` parameter. The END implementations are specialized for $n = 512$ and $n = 1024$. No END implementation was adapted or measured at $n = 256$, which is marked by "–". All candidates are jointly timed on common inputs.

Table 9 compares BAT, END, and our selected implementations using canonical inputs and centered outputs. The BAT columns report its polynomial-arithmetic implementation instantiated at each listed (q, n). Timings include BAT's `mq_set`/`mq_snorm` conversions and all required copies, layout conversions, and output centering. END's native C output is nonnegative, while its signed AVX2 output need not be centered.

Our selected complete multiplication costs are 23.86–56.13% lower than BAT in C and 57.69–64.43% lower in AVX2. Against END, the C reductions are 36.71%–58.43%. Against END AVX2, our costs are 5.24% higher at $(257, 512)$, 3.58% lower at $(257, 1024)$, and 0.76% higher at $(769, 1024)$. The $(769, 1024)$ costs are similar on this platform.

The gains over BAT extend beyond the residual formula. BAT uses Montgomery radix 2^{32}, representatives in $1, \ldots, q$, 16-bit AVX2 butterflies, and widened residual products with reductions amortized over short sums. At $(257, 1024)$, our AVX2 implementation with schoolbook residual multiplication already reduces complete cost from BAT's 10965 cycles to 5728 cycles. Recursive Karatsuba further reduces it to 4566 cycles. At $(769, 1024)$, lazy recursion reduces complete cost by 15.24% relative to our schoolbook implementation. These internal comparisons distinguish the effects of residual formulas and reductions from other implementation differences. Complete multiplication includes transforms and interface conversions in addition to residual products. Section 6.4 therefore evaluates residual replacement within END's existing transform pipeline, including the required conversions, to determine its effect on complete KEM calls.

6.4 KEM integration

To evaluate the impact of our encoding, decoding, and arithmetic components on complete KEM performance, we replace the corresponding routines in BAT and END and benchmark the resulting implementations. We perform three runs. Each run uses 31 paired key generation inputs and 101 paired batches of ten calls each for encapsulation and decapsulation. For each paired key-generation input or encapsulation/decapsulation batch j, we compute $\Delta_j = 100(1 - T_{\text{modified},j}/T_{\text{baseline},j})$, where T denotes the measured cost per call. Costs and reductions are summarized separately as the medians of their three within-run medians. The smaller-capacity ZS experiment instead reports mean KeyGen

costs, including all retries.

We test BAT with $(q, n) = (257, 512)$ and $(769, 1024)$, and END with $(q, n) = (257, 512)$ and $(257, 1024)$, in both C and AVX2. The corrections are applied before evaluating component replacements. The component results guide the replacement scope. For BAT, we replace the complete polynomial multiplication in encryption and reencryption. Its key generation and decryption core retain their original operations and transform reuse. For END, we limit our arithmetic modifications to replacing the residual multiplication, as its forward and inverse NTT routines are already highly optimized. Both integrations replace public key encoding and decoding, leaving ciphertext formats unchanged. Arithmetic replacements are checked for bytewise agreement and cross-decapsulation. Codec replacements are checked for equality of decoded public keys and KEM outputs under their respective formats. Low-capacity ZS tests verify retries of the entire key-generation call.

Table 10: Cycles of KEM for the baseline and combined component replacements.

Instance	Impl.	Configuration	PK	KeyGen		Encaps		Decaps	
			bytes	cycles	Δ (%)	cycles	Δ (%)	cycles	Δ (%)
BAT-512	C	Baseline	521	25,440,652	–	30,825	–	171,541	–
		Modified	520	25,505,496	−0.09†	13,790	55.25	158,746	7.41
BAT-512	AVX2	Baseline	521	24,383,053	–	11,067	–	50,603	–
		Modified	520	24,312,652	0.17	5,914	46.54	46,266	8.55
BAT-1024	C	Baseline	1230	104,367,972	–	51,327	–	288,421	–
		Modified	1230	104,331,625	0.01†	28,460	44.52	272,856	5.42
BAT-1024	AVX2	Baseline	1230	96,591,405	–	22,825	–	107,170	–
		Modified	1230	96,595,803	0.05	13,728	39.93	98,765	7.81
END-512	C	Baseline	514	9,616,670	–	26,356	–	234,909	–
		Modified	519	9,550,714	0.41	15,459	41.36	178,488	23.99
END-512	AVX2	Baseline	514	5,344,105	–	17,453	–	72,833	–
		Modified	519	5,338,578	0.12	10,146	41.89	67,918	6.98
END-1024	C	Baseline	1027	32,458,445	–	60,764	–	745,084	–
		Modified	1033	32,308,036	0.47	31,116	48.84	403,299	45.89
END-1024	AVX2	Baseline	1027	23,618,185	–	32,003	–	168,629	–
		Modified	1033	23,530,721	0.34	17,494	45.37	155,127	8.04

Baseline denotes the original BAT implementation and FCL correction for END. Modified replaces the complete encryption and reencryption product and the public-key codec in BAT, and the supported residual products and public-key codec in END. All END configurations share the same baseline corrections. † marks a difference not consistently resolved across the three runs.

Complete KEM costs. Table 10 reports the combined arithmetic and codec replacements. At $q = 257$, ZS uses $(n, C) = (512, 51)$ and $(1024, 67)$. Both meet the 2^{-128} reference overflow target, using $p_\perp(n, C)$ for uniform ring elements and $p_\perp(n, C)/\alpha_{n,257}$ for uniform invertible ring elements. For the implemented KeyGen distribution, Proposition 1 adds the corresponding decisional-NTRU distinguishing advantage to the reference bound.

In BAT, combined replacement reduces encapsulation and decapsulation costs by 39.93–55.25% and 5.42–8.55%, respectively. Complete-product replacement alone gives reductions of 30.21–41.39% and 5.38–8.39%. The accelerated product also serves reencryption, while the decryption core is unchanged. Decapsulation reads public key coefficients from the expanded secret key, so faster public key decoding provides no direct benefit.

Table 11: Public key size and KeyGen cost at smaller ZS capacities. Δ_{KG} is the percentage change in mean KeyGen cost, including retries, relative to $C = 51$ for $n = 512$ and $C = 67$ for $n = 1024$. Positive values indicate higher cost.

Instance	C	PK	Overflow probability	Overflows		Δ_{KG} (%)	
	bits	bytes	$p_\perp$	C	AVX2	C	AVX2
BAT-512	51	520	4.15×10^{-40}	0	0	–	–
	33	518	7.64×10^{-21}	0	0	+0.010	0.000
	16	515	9.18×10^{-7}	0	0	+0.004	−0.002
END-512	51	519	4.15×10^{-40}	0	0	–	–
	33	517	7.64×10^{-21}	0	0	+0.011	+0.003
	16	514	9.18×10^{-7}	0	0	+0.030	−0.001
END-1024	67	1033	5.23×10^{-40}	0	0	–	–
	45	1030	1.09×10^{-20}	0	0	−0.004	−0.001
	16	1026	3.44×10^{-3}	24	20	+0.365	+0.316

C is the selector capacity in bits. The reference probability is $p_\perp(n, C) = \Pr[Z > C]$ for $Z \sim \mathrm{Bin}(n, 2/257)$. Each overflow count is taken over 6144 initial public-key candidates. Overflow counts are measured on actual generated public keys. These probabilities concern encoding overflow, not decryption failure.

For END C, residual replacement alone reduces encapsulation and decapsulation costs by 14.09–26.14% and 21.06–44.02%, respectively. Adding codec replacement raises these reductions to 41.36–48.84% and 23.99–45.89%. For END AVX2, combined replacement gives reductions of 41.89–45.37% and 6.98–8.04%. Ablation results show that, after interface adaptation, the END AVX2 gains mainly come from the codec.

Timings include full KeyGen restarts on ZS overflow. Under independent restarts, the retry proposition applies with candidate overflow probability $\varepsilon_\theta(C)$. KeyGen changes are below 0.5%, with several differences not consistently resolved across runs (Table 10). NTRU equation solving and the remaining KeyGen arithmetic are retained, so the replacements affect only a small fraction of the total cost.

Selector capacity and key-generation retries. With arithmetic fixed, $C = 16$ gives public key sizes of 515, 514, and 1026 bytes for BAT-512, END-512, and END-1024, saving 6, 0, and 1 bytes over their original formats, respectively. For END-1024, mean KeyGen cost, including retries, increases by 0.365% in C and 0.316% in AVX2 relative to $C = 67$. Table 11 reports overflows among $3 \times 2048 = 6144$ initial public key candidates per scheme. Under independent, identically distributed sampling, the zero counts at all tested $C \geq 33$ give a one-sided exact 95% upper confidence limit of 4.875×10^{-4} per configuration. For END-1024 at $C = 16$, the 24 overflows in C and 20 in AVX2 are compatible with the uniform-model expectation of 21.16. Backend results are analyzed separately. These observations neither establish the much smaller reference probabilities nor quantify the decisional-NTRU distinguishing advantage.

7 Conclusion

The structure of the moduli 257 and 769 supports faster encoding, uniform polynomial generation, and polynomial multiplication. ZS makes the capacity and accepted domain explicit. Bounded radix quotients reduce conversion work and support grouped sampling. Proved arithmetic ranges remove unnecessary corrections, while register scheduling reduces storage within a fixed residual formula. Complete KEM experiments show which component gains survive integration. Replacing BAT's encryption product and public-key codec reduces encapsulation time by 39.93–55.25%; END C benefits from residual and codec replacement, while its AVX2 gains mainly come from the codec. Component

selection must therefore be validated in the calling code, including transform reuse, format choices, and adaptation costs.

A Proof of the Arithmetic Range Contract

The bounds cover one complete product for $|a_i|, |b_i| \le q-1$. Layout adapters only permute entries. Ordinary signed sums must fit their types; low-word operations in Montgomery correction intentionally use modular semantics. The tested x86-64 C implementation uses arithmetic right shifts and two's-complement low-16-bit narrowing to `int16_t`.

Reduction and forward layers.

Lemma 2 (Signed Montgomery bound)**.** *For $u = \mathrm{int16}(tq^{-1} \bmod R)$, $t - uq$ is divisible by $R = 2^{16}$ and $|u| \le R/2$. Consequently*

$$\mathsf{M}_q(t) = (t - uq)/R \equiv tR^{-1} \pmod q, \qquad |\mathsf{M}_q(t)| \le \mu_q(|t|) = \lfloor |t|/R + q/2 \rfloor . \tag{6}$$

The condition $|t| + Rq/2 < 2^{31}$ bounds the signed correction difference.

The claims follow from divisibility and $|u| \le R/2$. C forms u by unsigned low-word multiplication; AVX2 subtracts signed high words whose low words agree. Applying this bound to each forward butterfly gives

$$F_{s+1} = F_s + \mu_q((q-1)F_s), \qquad F_0 = q - 1. \tag{7}$$

Its seventh terms are 1171 and 3618, so all forward sums fit 16 bits.

Residual bounds. Put $F = F_7$, $D = \mu_q(F^2)$ and $S = \mu_q(4F^2) + 2D$. A schoolbook product is bounded by D and a pair cross term by S. For convolution column k, let e_k count pairs $i < j$ with $i + j = k$, and d_k indicate a diagonal term. Then $C_k = e_k S + d_k D$ bounds the pair column; schoolbook instead uses D times the number of ordered pairs. Folding gives

$$H_k = C_k + \mu_q((q-1)C_{k+B}) \quad (k < B-1), \qquad H_{B-1} = C_{B-1}. \tag{8}$$

These absolute-sum bounds cover partial accumulations as well as outputs.

For recursive Karatsuba with input bound G, let $K_{m,s}$ be the nonnegative bound vector for convolution of length-m operands, each coefficient summing s original inputs. The leaf is $K_{1,s} = (\mu_q((sG)^2))$. For $m > 1$, set $A = K_{m/2,s}$ and $D' = K_{m/2,2s}$, then form

$$K_{m,s} = \mathrm{shift}_0(A) + \mathrm{shift}_m(A) + \mathrm{shift}_{m/2}(D' + 2A),$$

where shifts place the shorter vectors into $2m - 1$ zero-initialized entries. This bounds the cross-product subtraction and overlapping recombination. Applying the same folding bound yields Table 2: use $G = 1171$ for $q = 257$ and $G = 384$ for strict $q = 769$ reduction. For the lazy variant, rounding and $43 \cdot 769 - 2^{15} = 299$ give

$$|x'| \le \lfloor 769/2 + 299 \cdot 3618/2^{15} \rfloor = 417.$$

Using $G = 417$ in the recurrence gives maximum convolution and folded bounds $10946 < 16383$. Across all cases the largest leaf operand is $8 \cdot 1171 = 9368$; its square and the Montgomery correction fit 32 bits. All recombination bounds are below 2^{15}, as are their absolute-sum partial bounds.

Inverse and scale. The entry bound 16383 limits first-layer sums and differences to 32766. We center both first-layer outputs to $h = (q-1)/2$. For subsequent layer $j = 1, \ldots, 6$, sums and differences are bounded by $2^j h$ and $\mu_q((q-1)2^j h) \le 2^j h$. Thus the last-layer bounds are 8192 and 24576. AVX2 centering uses $v_q = \lfloor 2^{15}/q + 1/2 \rfloor$ and $r = x - q\lfloor x v_q / 2^{15} + 1/2 \rfloor$. For signed 16-bit x, $|r| \le q/2 + |q v_q - 2^{15}| < q$; one masked correction on either side centers it. The C path uses a rounded $2^{24}/q$ reciprocal. Its reciprocal errors are 1 and 57; since q is odd, $32768 \cdot 57 < 2^{23}$ ensures the exact nearest-integer quotient throughout the signed 16-bit interval. The biased products lie in $[-2130739200, 2147451135]$, within signed 32-bit range.

Forward and folding twiddles carry R, while coefficient products introduce R^{-1}. Final scaling by $\gamma_q = R^2/128 \bmod q$, namely 255 and 655, gives $128R^{-1}\gamma_q R^{-1} \equiv 1 \pmod q$. AVX2 fuses the last difference twiddle with this scaling, giving constants 225 and 147, both no larger than γ_q. C applies γ_q to every last-layer output. Equation (6) therefore bounds final outputs by $\mu_{257}(8192 \cdot 255) = 160$ and $\mu_{769}(24576 \cdot 655) = 630$. In particular, the implementation inverse satisfies $\mathsf{INTT}_{\mathrm{impl}}(\mathsf{NTT}_{\mathrm{impl}}(a)) \equiv Ra \pmod q$ after the required layout conversion: it is normalized for a preceding residual product. Further product accumulation needs separate bounds.

References

[ABCG20] Erdem Alkim, Yusuf Alper Bilgin, Murat Cenk, and François Gérard. Cortex-m4 optimizations for {R, M} lwe schemes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 336–357, 2020.

[BBC+20] Daniel J Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. Ntru prime: round-3 updates. *National Institute of Standards and Technology (NIST)*, 2020.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European symposium on security and privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BDP+20] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. The Keccak Code Package. GitHub repository, 2020. Public-domain (CC0); includes Van Keer's simple 64-bit Keccak implementation.

[BSVA15] Daniel J. Bernstein, Peter Schwabe, and Gilles Van Assche. Tweetable FIPS 202. Online, 2015.

[CHK+21] Chi-Ming Marvin Chung, Vincent Hwang, Matthias J Kannwischer, Gregor Seiler, Cheng-Jhih Shih, and Bo-Yin Yang. NTT multiplication for NTT-unfriendly rings: New speed records for saber and NTRU on cortex-M4 and AVX2. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):159–188, 2021.

[DKSRV18] Jan-Pieter DAnvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren. Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM, 2018.

[FKPY22] Pierre-Alain Fouque, Paul Kirchner, Thomas Pornin, and Yang Yu. BAT: Small and fast KEM over NTRU lattices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(2):240–265, 2022.

[HLS+22] Vincent Hwang, Jiaxiang Liu, Gregor Seiler, Xiaomu Shi, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. Verified NTT multiplications for NIST-PQC KEM lattice finalists: Kyber, SABER, and NTRU. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):718–750, 2022.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray C. C. Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):614–636, 2022.

[Kar63] Anatolii Karatsuba. Multiplication of multidigit numbers on automata. In *Soviet physics doklady*, volume 7, pages 595–596, 1963.

[Lem19] Daniel Lemire. Fast random integer generation in an interval. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 29(1):1–12, 2019.

[LFZZ22] Zhichuang Liang, Boyue Fang, Jieyu Zheng, and Yunlei Zhao. Compact and efficient KEMs over NTRU lattices. arXiv:2205.05413, version 2, 2022.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 1–23. Springer, 2010.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. A toolkit for ring-lwe cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 35–54. Springer, 2013.

[LZL+25] Yijian Liu, Yu Zhang, Xianhui Lu, Yao Cheng, and Yongjian Yin. DAWN: Smaller and faster NTRU encryption via double encoding. Cryptology ePrint Archive, Paper 2025/1520, 2025.

[LZL+26] Yijian Liu, Yu Zhang, Xianhui Lu, Yao Cheng, and Yongjian Yin. How compact can NTRU encryption be? heuristic frontiers and practical schemes. Cryptology ePrint Archive, Paper 2026/1715, 2026.

[Mon85] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[Nat15] National Institute of Standards and Technology. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Federal Information Processing Standards Publication 202, National Institute of Standards and Technology, 2015.

[Nat24] National Institute of Standards and Technology. Module-Lattice-Based Key-Encapsulation Mechanism Standard. Federal Information Processing Standards Publication NIST FIPS 203, U.S. Department of Commerce, Washington, DC, August 2024.

[Pla21] Thomas Plantard. Efficient word size modular arithmetic. *IEEE Transactions on Emerging Topics in Computing*, 9(3):1506–1518, 2021.

[Por21] Thomas Pornin. BAT KEM/signature reference implementation. GitHub repository, 2021. Commit `8280af7c9d4d`.

[Sei18] Gregor Seiler. Faster AVX2 optimized NTT multiplication for ring-LWE lattice cryptography. Cryptology ePrint Archive, Paper 2018/039, 2018.

[Wes20] Bas Westerbaan. New f1600x4: four-way parallel Keccak-p[1600] for SPHINCS+. GitHub pull request #14, SPHINCS+ repository, 2020. Adapted from the author's four-way AVX2 Keccak in Cloudflare's CIRCL library.

[ZHNB06] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 59–59. IEEE, 2006.