HAETAE on a Diet: Less Stack, Less Time on Low-End Microcontrollers

MinGi Kim, YoungBeom Kim, DongHyun Shin, Seog Chung Seo*

Kookmin University, Seoul, Republic of Korea
{mingi1684,dkldkl13,darania,scseo}@kookmin.ac.kr

Abstract. HAETAE has shorter public keys and signatures than ML-DSA, but the signing stack of its reference implementation reaches 158 KiB on Cortex-M4. The Low-Stack HAETAE implementation presented at CHES'26 brings the signing stack down to 5.7–5.9 KiB; yet even HAETAE-2 still needs 8,718 B in a realistic IoT scenario, where the secret key and the signature buffer share the same SRAM. Gaussian regeneration further increases signing time by up to a factor 2.6 relative to the C reference. Although HAETAE's modulus $q = 64\,513$ fits in 16 bits, signing and verification also need values modulo $2q$, so existing implementations keep 32-bit working polynomials. We store them in 16 bits and lift each coefficient to modulo $2q$ in a register only when it is consumed, and let objects with disjoint lifetimes share buffers, which lowers peak stack usage without any additional Gaussian regeneration. Exact Gaussian replay and caching method then record the acceptance decisions and partial CDT ranks of the first generation in the unused space of the signature buffer, so that the original samples are restored from the same random stream with less recomputation. For Cortex-M4, Cortex-M3, MSP430, and AVR, we further design NTT modular multiplication and butterflies that preserve 16-bit storage, and optimize the arithmetic of the FFT and the Gaussian sampler. We validate our implementation on 8 KiB hardware: a single C firmware image on the ATmega2560 runs HAETAE-2 key generation, signing with the generated key, and verification, and uses a total of 8,070 B of SRAM, including the secret key, the signature buffer, static data, and the stack. To the best of our knowledge, this is the first standardization-track PQC-DSA implementation that signs within 8 KiB of total SRAM on physical hardware. Compared with Low-Stack HAETAE on Cortex-M4, Cortex-M3, MSP430, and AVR, our implementation reduces stack usage by up to 34% and runs key generation and signing 1.18–2.51× faster wherever Low-Stack can run them.

Keywords: HAETAE · Post-Quantum Cryptography · resource-constrained devices

1 Introduction

Digital signatures are the primary means of establishing origin and integrity in firmware updates, secure boot, and device authentication. On microcontrollers, the real-time operating system (RTOS), the network stack, and the I/O drivers share the SRAM, so the space left for cryptographic operations is far smaller than the total SRAM. Post-quantum schemes use larger parameters and require more stack space than ECC-based schemes, making feasibility a more immediate concern than speed on low-end devices [HMK+26, KvVBS26].

For this reason, recent work places feasibility on low-end devices at the center of its evaluation. At AfricaCrypt'22, Bos et al. [BRS22] implemented CRYSTALS-Dilithium, and

*Corresponding author

at CHES'23, Bos et al. [BBC+23] implemented FrodoKEM, each within a few kB of stack. At CHES'26, low-memory implementations of HAETAE [KvVBS26], CROSS [HMK+26, SGB+26], and Raccoon [HSZ+26] were presented. These implementations share the same strategy: they recompute large intermediate values instead of storing them to lower peak memory, generate seed-derived objects by streaming, and reuse the same space for buffers whose lifetimes do not overlap.

Among them, HAETAE [CCD+24], proposed at CHES'24, is the most balanced choice among signature schemes for low-end microcontrollers. CROSS is a code-based signature scheme built on zero-knowledge proofs, so even after memory optimization its stack usage on Cortex-M4 is up to 24 kB, with a memory footprint, including the signature, below 128 kB [SGB+26], and only six variants fit below 32 kB [HMK+26]. The work on Raccoon focuses on reducing the memory requirements of high-order masked implementations for side-channel protection on Cortex-M4 [HSZ+26]. Like ML-DSA, HAETAE is a Fiat–Shamir signature scheme over module lattices, but it samples the ephemeral vector from a hyperball rather than a hypercube, yielding shorter public keys and signatures at the same security level [CCD+24]. HAETAE has been selected as a final KpqC algorithm [CCD+26].

However, since the scaling factor of the hyperball depends on the norm of the entire Gaussian vector, the reference implementation keeps several large polynomial vectors live at the same time. Hyoung et al. [HCK+26] applied on-the-fly matrix generation and two-pass hyperball sampling to HAETAE-5 on Cortex-M4, lowering the signing stack from 91,176 B to 8,092 B. In follow-up work presented at CHES'26, Kim et al. [KvVBS26] applied pass decomposition, early rejection, and reverse-order streaming rANS to all three modes, bringing the signing stack on Cortex-M4 down to 5.7–5.9 KiB.

The endpoints of an edge network are very small devices, such as sensor nodes. For example, a Class 1 constrained node in RFC 7228 has about 10 KiB of RAM [BEK14], and the ATmega2560 used in post-quantum protocol research on 8-bit AVR sensor nodes has 8 KiB of SRAM [KS25, DKZ+26]. Bos et al. [BRS22] targeted a signing stack below 8 KiB for Dilithium. Signature-based mutual authentication requires both parties to generate signatures, and sensor readings that pass through untrusted relays can prove their origin and integrity through the node's own signature [OKG+12]. If a device supports only Verify, the protocol must fall back to classical signatures or pre-shared keys on that segment. A segment that uses classical signatures becomes a downgrade path for a quantum adversary, while pre-shared keys still impose the burden of key distribution. Therefore, extending post-quantum authentication to the end devices requires signing to be feasible on devices with 8 KiB of SRAM.

However, in a realistic scenario where the secret key, the signature buffer, and the working stack share one SRAM, none of the post-quantum signature schemes we compared, including HAETAE, can yet sign within 8 KiB of SRAM. When signing with a secret key generated on the device, even HAETAE-2 needs 8,718 B on Cortex-M4: 5.7 KiB of stack plus a 1,408 B secret key and a 1,474 B signature [KvVBS26]. The Low-Stack implementation [KvVBS26] could not execute Sign on an AVR with 8 KiB of SRAM and supported only Verify. The static upper bound on SRAM usage for its HAETAE-2 Sign firmware is 9,028 B (Section 7). In a simulator-based evaluation of Dilithium for 8-bit AVR, Hwang et al. [HKS25] report that Dilithium2 requires 12,059 B for the Sign stack alone, exceeding 8 KiB. Protocol research on 8-bit AVR sensor nodes has opted for KEM-based authentication [KS25, DKZ+26].

If memory is the first constraint, execution time is the second. Regeneration-based designs [BRS22, KvVBS26, HCK+26] compute the same values several times. On Cortex-M4, Low-Stack [KvVBS26] reported key generation and signing times of up to 1.9 and 2.6 times those of the C reference, respectively. On low-end devices, execution time translates into protocol latency and energy cost [BZB+22, TDF+23, KS25], and low-memory work

on other schemes has likewise improved performance within a limited working space after reducing memory [BBB+24, ADE+25, SGB+26, BF26]. HAETAE therefore also needs optimizations that reduce the recomputation cost while keeping a small working space. Accordingly, in this paper, we not only reduce the stack usage of HAETAE so that signing fits in 8 KiB of SRAM but also propose optimizations that lower the regeneration cost.

Contributions Our contributions are as follows.

1. **First PQC-DSA Signing on 8 KiB SRAM** We store the coefficients of the working polynomials in 16 bits instead of the 32-bit representation used by existing implementations [CCD+26, KvVBS26, HCK+26], lifting them to modulo $2q$ only when needed and rearranging buffers by object lifetime, which further reduces peak stack usage without additional regeneration. This idea provides more memory headroom for an RTOS and a network stack on Cortex-M4-class devices, and enables signing even with 8 KiB of SRAM. On a physical 8-bit AVR ATmega2560 with 8 KiB of SRAM, a single C firmware image runs HAETAE-2 KeyGen, Sign, and Verify, signing with a key generated on the device. Its total SRAM usage, including the secret key, the signature buffer, static data, and the call stack, is only 8,070 B. To the best of our knowledge, this is the first standardization-track PQC-DSA implementation that signs within 8 KiB of total SRAM on physical hardware.

2. **Exact Gaussian Replay with Compact Records** Existing low-memory implementations [KvVBS26, HCK+26] repeat the CDT search and the acceptance test every time they regenerate the same Gaussian polynomial. We propose exact Gaussian caching and replay, which store the acceptance decisions and the partial CDT ranks of the first generation as a compact record in the unused space of the signature output buffer and reconstructs the original samples exactly from the same random stream. On Cortex-M4, it reduces Gaussian generation cycles by about 7–13% and total Sign cycles by about 6–12% in our C implementation, compared with the configuration without replay that retains the same y_1 cache.

3. **ISA-Specific Arithmetic Optimizations** We tailor Montgomery, Barrett, and Plantard reductions and their combination with the NTT butterflies to the multiplication instructions and register constraints of Cortex-M4, Cortex-M3, MSP430, and AVR. In the FFT and the Gaussian sampler, we optimize the partial products and accumulation while preserving the original fixed-point rounding. As a result, on all four ISAs the assembly NTT is 2.84× (Cortex-M3) to 5.52× (Cortex-M4) faster than the NTT of our 16-bit C implementation.

4. **Evaluation on Four Architectures** On Cortex-M4, our C implementation reduces stack usage across KeyGen, Sign, and Verify by 21–34% relative to Low-Stack C [KvVBS26]. Our assembly implementation is 1.23–1.54× faster for KeyGen and 1.18–1.32× faster for Sign than the Low-Stack assembly [KvVBS26]. On Cortex-M3 and MSP430, we reduce API stack usage by 20–33% and 5–31%, respectively. Across these two platforms, our assembly implementation is 1.34–1.90× (KeyGen), 1.37–2.51× (Sign), and 1.16–1.32× (Verify) faster than Low-Stack C [KvVBS26]. On AVR, we reduce Verify stack usage by 19–23%, and our assembly Verify is 1.19–1.21× faster than the Low-Stack assembly. For KeyGen and Sign, which Low-Stack cannot execute, our assembly is 1.57× and 1.86× faster than our C implementation, respectively. On Cortex-M4, our assembly Sign still requires 2.10–2.45× as many cycles as the reference C implementation, whereas our assembly Verify is 1.12–1.30× faster.

Table 1: Main parameters of HAETAE 1.2.0.

Parameter	Description	HAETAE-2	HAETAE-3	HAETAE-5
n	Polynomial degree	256	256	256
q	Modulus	64,513	64,513	64,513
(k, ℓ)	Dimensions of z_2 and z_1	$(2, 4)$	$(3, 6)$	$(4, 7)$
d	Truncated bits of the public key	1	1	0
τ	Hamming weight of c	58	80	128
M	Expected number of signing attempts	6.0	5.0	6.0
$\lvert\mathsf{pk}\rvert$	Public key size (bytes)	992	1,472	2,080
$\lvert\mathsf{sk}\rvert$	Secret key size (bytes)	1,408	2,112	2,752
$\lvert\mathsf{sig}\rvert$	Signature size (bytes)	1,474	2,349	2,948

Organization Section 2 introduces HAETAE and its implementation characteristics, and Section 3 analyzes the existing Low-Stack implementation and its remaining costs. Sections 4, 5, and 6 present narrow storage with object lifetimes, exact Gaussian replay, and the ISA-specific arithmetic, and Section 7 reports the evaluation on the four architectures, including the integrated run on an 8 KiB AVR.

2 Background

2.1 HAETAE 1.2.0

HAETAE [CCD+24] is a module-lattice signature scheme based on Fiat–Shamir with Aborts (FSwA). Its overall FSwA structure [Lyu09] is similar to that of ML-DSA [Nat24]. KeyGen generates a public matrix and a secret vector with small coefficients from seeds and computes the public key. Sign forms a commitment w from an ephemeral vector y sampled uniformly from a hyperball and derives a challenge c from w and the message. As in BLISS [DDLL13], it uses bimodal rejection sampling with a random sign bit β to compute the response $z = y + (-1)^{\beta}(c \star s)$. Signing attempts that fail the norm conditions are rejected; the accepted response and hint are compressed to form the signature. Verify reconstructs the commitment from the signature and public key and checks the norm bounds and the consistency of the challenge. For a detailed description of the HAETAE algorithms and their underlying operations, we refer the reader to [CCD+26].

All parameter sets use the polynomial ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ with $n = 256$ and the prime modulus $q = 64\,513 = 2^{16} - 2^{10} + 1$. Since $q \equiv 1 \pmod{512}$, polynomial multiplication in R_q can be computed using a fully splitting NTT. Table 1 summarizes the main parameters used in this work, based on the HAETAE specification [CCD+26].

2.2 Implementation Challenges in HAETAE

Modulo-2q arithmetic. ML-DSA performs polynomial ring arithmetic modulo q. Since its modulus $q = 8\,380\,417$ is 23 bits wide, coefficients are typically stored in 32-bit words. HAETAE uses intermediate values modulo $2q$ in commitment computation, hint generation, and verification to support bimodal rejection sampling [CCD+26]. Although HAETAE's $q = 64\,513$ fits in 16 bits, $2q = 129\,026$ and the sums and differences in NTT butterflies can exceed the 16-bit range. Thus, the storage width of working coefficients cannot be determined from q alone; the ranges of intermediate values and the requirements of subsequent operations must also be considered.

Hyperball sampling. ML-DSA samples its ephemeral vector y uniformly from a hypercube, allowing each polynomial to be generated independently from XOF output [BRS22].

In contrast, HAETAE samples $y = (y_1, y_2)$ from a hyperball by scaling a discrete Gaussian vector according to its overall norm. This requires first computing the norm of $n(k+\ell)+2$ Gaussian samples [CCD+26]. The reference implementation therefore retains all Gaussian samples until scaling is complete. Since Gaussian sampling is deterministic for a given seed and nonce, the original samples can be regenerated from the same inputs.

Secret-key norm check. ML-DSA key generation does not reject secret-key candidates based on a separate norm check. HAETAE, however, rejects and resamples candidates that do not satisfy $N(s) \leq \gamma^2 n$. This check bounds $\|c \star s\|_2$ for all admissible challenges c. The check applies a 256-point fixed-point FFT to each secret polynomial. KeyGen therefore requires 2 KiB of FFT scratch space and a 32-bit norm accumulator in addition to the workspace for NTT-based polynomial arithmetic [CCD+26]. The secret-key norm check thus directly affects the peak workspace of KeyGen.

3 Low-Stack HAETAE and Remaining Costs

We retain the pass decomposition, streaming, early rejection, and row-streamed verification of Low-Stack [KvVBS26] as our baseline.

3.1 Prior Work on Low-Stack HAETAE

Low-memory implementations of lattice-based schemes combine on-the-fly generation, streaming, recomputation, and buffer reuse based on object lifetimes to avoid storing large intermediate objects simultaneously. Bos et al. applied techniques such as on-the-fly generation and lifetime-based workspace reuse to CRYSTALS-Dilithium [BRS22].

For HAETAE-5, Hyoung et al. used streamed matrix generation and two-pass hyperball sampling [HCK+26]. Kim et al. subsequently extended these techniques to HAETAE-2/3/5 in Low-Stack HAETAE [KvVBS26]. They further reduced peak stack usage in KeyGen, Sign, and Verify through rejection-aware pass decomposition, component-level early rejection, reverse-order streaming rANS encoding, and row-streamed verification. We refer to this implementation as Low-Stack.

3.2 Baseline Organization

KeyGen. Low-Stack uses streaming matrix–vector multiplication in KeyGen, generating the required polynomials from seeds on-the-fly and using them immediately without materializing the full public matrix or secret vector. For HAETAE-2/3, it accumulates the matrix product and secret norm one component at a time. For HAETAE-5, it separates the norm check and matrix–vector multiplication into two passes, regenerating only candidates that pass the check for use in the second pass.

Sign. Low-Stack decomposes the signing loop into **Pass A**, **B**, and **C** to reduce peak stack usage. **Pass A** generates the commitment and challenge from the hyperball sample y. **Pass B** regenerates y from the same seed and nonce, computes the response z, and checks the rejection conditions component by component, applying component-level early rejection. **Pass C** runs only for attempts that pass the rejection tests in **Pass B**, recomputing the required intermediate values and encoding the signature. **Pass C** is split into **C1** for hint generation and encoding, and **C2** for packing z_1. The local workspaces of these subpasses have non-overlapping lifetimes. If encoding fails, a new signing attempt begins.

Verify. Low-Stack performs row-streamed matrix–vector multiplication in Verify without materializing a full polynomial vector. For each row, it decodes the required signature components again and regenerates public matrix entries on-the-fly for immediate use.

Each API reduces peak stack usage by reusing workspace across objects with non-overlapping lifetimes. In KeyGen, regions of the FFT scratch, sampling workspace, and workspaces for norm and row accumulation share storage in this way.

3.3 Remaining Costs

We analyze the memory state of Low-Stack [KvVBS26] in terms of the major polynomial objects that are simultaneously live. We exclude smaller objects such as hash states, scalars, and call frames. The main working polynomials in R_q use 32-bit coefficients, so for $n = 256$, each polynomial occupies $|\mathsf{poly}| = 4 \cdot n = 1\,024\,\mathrm{B}$. Since the signing passes' local workspaces have disjoint lifetimes, the peak working storage in this model is

$$S_{\mathrm{sign}} = S_{\mathrm{driver}} + \max\{S_A, S_B, S_{C_1}, S_{C_2}\} \tag{1}$$

where S_{driver} denotes the common state retained across passes, and each pass term is the maximum additional working storage simultaneously live in that pass and its callees. Both S_{C_1} and S_{C_2} include the live buffers common to **Pass C**.

Two costs remain: storing working objects and regenerating deterministic samples.

Storage cost. Although full vectors are no longer stored, several working polynomials with 32-bit coefficients remain simultaneously live at the peak of each API. In KeyGen, this state includes FFT scratch and accumulators; in Sign and Verify, it includes pass-local rows and intermediate objects. Low-Stack's state model counts each live object as $|\mathsf{poly}|$. Low-Stack considered 16-bit storage but did not adopt it because intermediate values such as $w = A\lfloor y_1 \rceil \bmod 2q$ can exceed 16 bits and the peak is determined by 2 KiB of fixed-point FFT scratch. However, a modulo-$2q$ value can be reconstructed from a modulo-q value and a parity bit when needed, and FFT scratch is needed only in KeyGen.

Regeneration cost. In Sign, Low-Stack regenerates samples from the same seed and nonce rather than storing a large Gaussian vector. An attempt that completes **Pass C1** generates the same y_2 polynomial five times: once for the norm and scaling computation, once each in **Pass A** and **Pass B**, and twice in **Pass C1**. Each generation repeats the CDT search and acceptance test, although the candidates and their acceptance decisions are identical for the same seed and nonce.

We reduce each of these costs without increasing the other. Narrower storage for working polynomials reduces storage cost without increasing regeneration cost (Section 4). ISA-specific arithmetic reduces the computational overhead introduced by narrow storage (Section 6). We also reduce regeneration cost without an additional buffer by storing Gaussian sampling decisions in the unused space of the signature output buffer (Section 5).

4 16-Bit Storage and Object Lifetimes

As discussed in Section 3, Low-Stack [KvVBS26] retains several working polynomials in 32-bit representations even after eliminating full vectors, leaving substantial storage costs. We reduce these costs while preserving the existing streaming and pass decomposition, without additional Gaussian regeneration or secret resampling. The key is to match each object's storage width to the information and range required by subsequent operations, rather than to the widest intermediate value encountered during arithmetic. We therefore store only the necessary information in narrow representations for long-lived state. Wider

representatives, such as those modulo $2q$, are reconstructed coefficient by coefficient at the point of use. We also revisit the lifetimes of the narrowed objects and arrange for states that are not needed simultaneously to share the same workspace.

4.1 Narrow Representations for Intermediate Values

Since HAETAE's $q = 64\,513$ is smaller than 2^{16}, canonical remainders modulo q fit in unsigned 16-bit words. Low-Stack also noted this and used narrower types for some objects with small ranges, such as high bits and hints. However, wider intermediate values, such as those modulo $2q$, led it to retain 32-bit representations for working polynomials in R_q. We decouple the storage width of working polynomials from the maximum width of arithmetic intermediates. Each polynomial retains only the modulo-q information required by subsequent operations in a 16-bit representation. Wider representatives, such as those modulo $2q$, are reconstructed coefficient by coefficient at the point of use. We thus reduce the storage width of working polynomials over the original modulus q from Low-Stack's 32-bit representation to a 16-bit representation. Specifically, we use each coefficient's stored modulo-q remainder and one parity bit to reconstruct its modulo-$2q$ representative.

Lemma 1. *Let q be odd, v an integer, $r = v \bmod q$, and $p = v \bmod 2$. Then the value $w = r + q\big((r \bmod 2) \oplus p\big)$ satisfies $w = v \bmod 2q$.*

Proof. Since $r \in [0, q)$ and $(r \bmod 2) \oplus p \in \{0, 1\}$, we have $0 \le w < 2q$ and $w \equiv r \equiv v \pmod q$. If $r \bmod 2 = p$, then $w = r$, so $w \bmod 2 = p$. Otherwise, $w = r + q$, and since q is odd, $w \bmod 2 = 1 - (r \bmod 2) = p$. The value $v \bmod 2q$ also lies in $[0, 2q)$ and has remainders modulo q and modulo 2 equal to r and p, respectively. Since $\gcd(2, q) = 1$, the Chinese remainder theorem gives a unique such integer in $[0, 2q)$, hence $w = v \bmod 2q$. □

In Sign and Verify, we store the parity of the first row of w in a 256-bit bitplane (32 B). All remaining rows have even coefficients, so $p = 0$. We therefore reconstruct each required coefficient on demand instead of storing a full polynomial of 17-bit representatives.

Not all intermediate objects can be narrowed in the same way. We normalize the coefficients of modulo-q working polynomials to canonical remainders in $[0, q)$ and store them as `uint16_t`. The rounded values $\lfloor y \rceil$ and $\lfloor z \rceil$ can be stored as `int16_t` because their required coefficient ranges fit in signed 16-bit words across all parameter sets. The coefficients of the small secrets lie in $[-2, 2]$, so `int8_t` suffices. The final challenge comparison in Verify requires only 256 binary values rather than the full coefficient representation, so we use a 256-bit bitset. In contrast, the unrounded y and z used in rejection tests, along with KeyGen's norm accumulator and fixed-point FFT scratch, retain 32-bit representations to preserve the required precision and range.

4.2 KeyGen: Narrow Rows and Reuse of Packed Secrets

Low-Stack's KeyGen uses a fixed-point FFT to check the secret-key norm and row-wise matrix–vector multiplication to compute the public-key polynomial. The FFT scratch and norm accumulator retain 32-bit representations to preserve the required precision and range. For the row accumulator used in matrix–vector multiplication, subsequent operations need only modulo-q values, so each coefficient can be stored in $[0, q)$ as `uint16_t`. This reduces each working row from 1 024 B to 512 B, while wider intermediate values are used only temporarily during arithmetic.

Low-Stack reuses the same workspace for the secret polynomial and FFT scratch. By the end of the norm check, the FFT has overwritten the secret, so the secret required for matrix–vector multiplication is resampled using the same seed and nonce. Our implementation packs the secret into the secret-key buffer before the FFT, then recovers the required coefficients by unpacking the stored secret-key bytes. Since the secret coefficients

are small integers, we avoid rematerializing a separate 32-bit polynomial and eliminate additional secret resampling. We also eliminate the by-value stack copy in Low-Stack's secret-key packing by passing a pointer.

In modes 2 and 3, the 32-bit norm accumulator and row accumulator are simultaneously live, so the 16-bit row representation described above directly reduces the required workspace. In mode 5, the two objects already share the same workspace because their lifetimes do not overlap, so narrowing the row alone does not reduce the size of this shared region. The reduction in KeyGen stack usage therefore results from combining narrow row representations, reuse of the existing secret-key output, and elimination of unnecessary polynomial copies, rather than from type narrowing alone.

4.3 Sign: Pass-Local Representations and Workspace Reuse

Low-Stack decomposes the signing loop into **Pass A**, **B**, and **C**, separating the buffer lifetimes of computation before the rejection decision and post-acceptance encoding. We preserve this pass structure and streaming order. We store the working objects of each pass in narrower representations suited to their required ranges and rearrange the workspace to reflect the reduced object sizes and their lifetimes. Figure 1 illustrates these changes and the dataflow in ①–⑭, for the common C implementation in mode 2.

In the driver workspace shown in Figure 1, Low-Stack retains two 32-bit row accumulators for paired-row processing, requiring 2 048 B. We reduce each accumulator to a 16-bit representation occupying 512 B, storing both rows in 1 024 B. The row pair is no longer needed after ③–⑥ in Figure 1, and the dense challenge is generated only after the last row has been processed. Since the row pair and challenge have non-overlapping lifetimes, we place the two 16-bit rows and the 1 024 B dense challenge in the same workspace. This reduces driver-level storage from 2 048 B to 1 024 B. This shows that narrow representations must be combined with the workspace reuse they enable to reduce peak storage.

Pass A. In **Pass A** (①–⑧ in Figure 1), we regenerate the fixed-point coefficients of y in ①and immediately accumulate their norm contributions as in the baseline. We store only the rounded values required by subsequent matrix–vector multiplication directly in an `int16_t` buffer, rather than storing the regenerated coefficients in a 32-bit fixed-point buffer. This replaces the 1 024 B fixed-point staging buffer with a 512 B buffer of rounded values. In the subsequent paired-row computation in ③–⑥, the modulo-q row accumulators and column workspace use 16-bit representations occupying 512 B each. When wider representatives modulo $2q$ are required, we reconstruct them coefficient by coefficient and use them immediately in subsequent operations such as high-bit extraction, without materializing a separate polynomial.

Pass B. **Pass B** (⑨–⑪ in Figure 1) generates the response z one component at a time and performs the rejection tests. The unrounded y and z retain 32-bit representations to preserve the fixed-point precision required by the rejection criterion. In contrast, the small secret multiplied by the challenge in ⑨requires a much smaller range. In modes 2 and 3, the secret coefficients lie in $[-2, 2]$, so we store them as `int8_t`, reducing the corresponding polynomial workspace from 1 024 B to 256 B. In mode 5, we retain the NTT-domain working polynomials of the challenge and secret in 16-bit modulo-q representations. Given the NTT result $t \in [0, q)$, we use the bound $|c \star s| \leq 2\tau < q/2$ to reconstruct the integer representative required for the rejection test on demand: if $t < q/2$, we use t; otherwise, we use $t - q$. The norm accumulation and rejection criterion in ⑩–⑪ remain unchanged, so **Pass B** reduces wide storage while preserving the precision of the original tests.

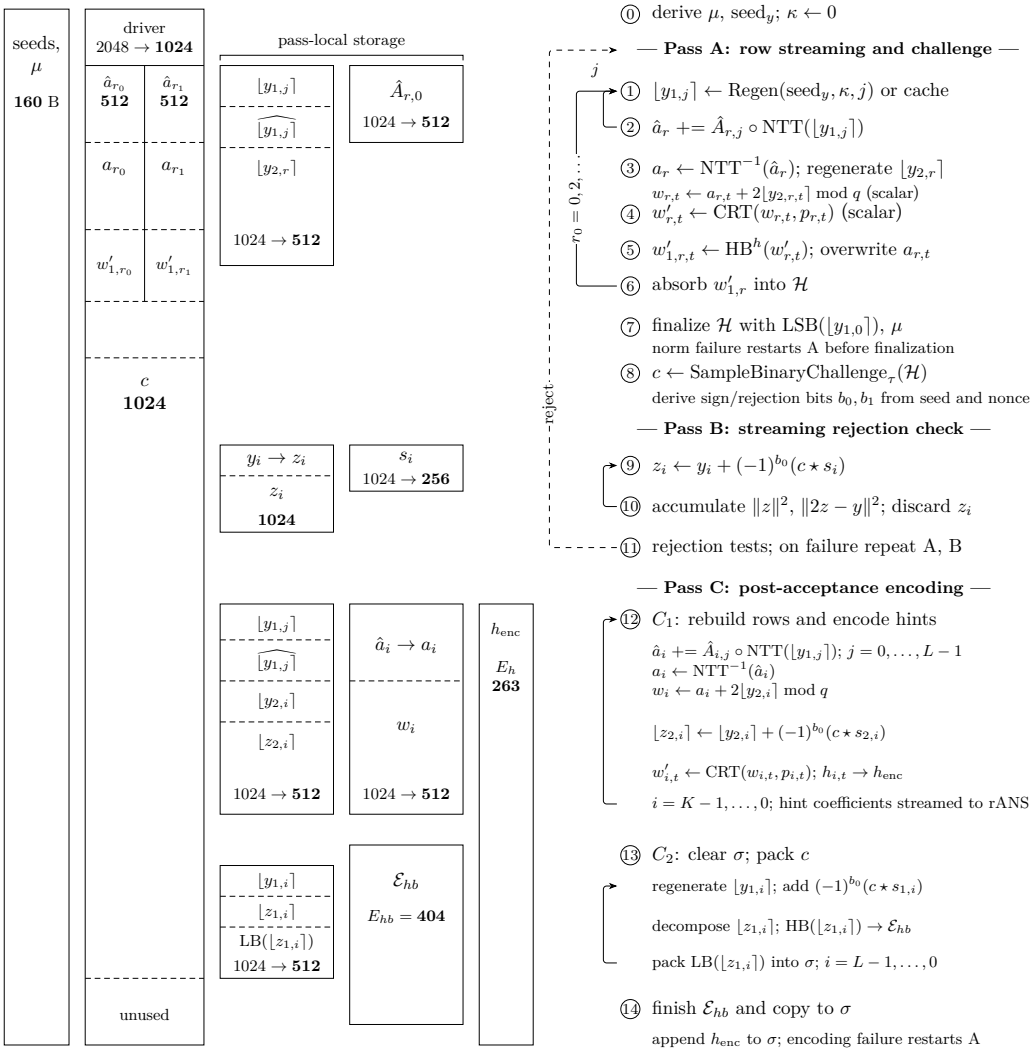


Figure 1: HAETAE-2 signing memory layout. Sizes are in bytes, shown as Low-Stack [KvVBS26] → this work. Bold numbers denote sizes in this work, including unchanged sizes. Dashed lines within boxes mark changes in the values stored in the same memory region; widths are not to scale.

Pass C. **Pass C** (⑫–⑭ in Figure 1) runs for candidates that pass the rejection test in **Pass B**. As in Low-Stack, **C1** generates the hint and **C2** encodes z_1.

C1 (⑫) recomputes the matrix rows as in **Pass A**, but retains the row accumulator and modulo-q working polynomials in 16-bit representations occupying 512 B each. In $z_2 = y_2 + m$, each coefficient of $m = (-1)^\beta(c \star s_2)$ is an integer, so

$$\lfloor z_2 \rceil = \lfloor y_2 + m \rceil = \lfloor y_2 \rceil + m.$$

For each row i, the baseline **C1** regenerates the same $y_{2,i}$ twice: once to add $2\lfloor y_{2,i} \rceil$ to the matrix product and once to form $z_{2,i}$. Our implementation retains $\lfloor y_{2,i} \rceil$ from the first regeneration in a 16-bit buffer and accumulates m_i directly into that buffer using coefficients read from the packed secret key. This forms $\lfloor z_{2,i} \rceil$ in the same buffer, eliminating the second regeneration of $y_{2,i}$ and the materialization of a wide fixed-point $z_{2,i}$. As in **Pass A**, wider representatives modulo $2q$ are reconstructed coefficient by

coefficient and used immediately for hint computation. The hint coefficients are encoded immediately using the existing reverse-order rANS streaming.

C2 (⑬–⑭) applies the same principle to z_1. We regenerate each $\lfloor y_{1,i} \rceil$, then add the challenge product computed using $s_{1,i}$ from the packed secret key to form $\lfloor z_{1,i} \rceil$ directly in a 16-bit buffer. We stream the high bits to the existing rANS encoder as they are produced and pack the low bits into the signature buffer, avoiding materialization of a full z_1 vector or a separate high-bit staging array. Thus, **Pass C** keeps the existing streaming order and reduces storage and redundant regeneration by reusing 16-bit working state.

4.4 Verify: Narrow Rows, Columns, and Challenge State

Low-Stack performs row-wise matrix–vector multiplication in Verify without materializing the full matrix product, but retains 32-bit representations for the working polynomials of each row and column. Our implementation preserves Low-Stack's row-wise streaming order while narrowing modulo-q working polynomials to 16-bit representations. In modes 2 and 3, we eliminate the separate sampling scratch used to generate the first column and accumulate the sampled coefficients directly into the row accumulator. Each row accumulator and column workspace stores coefficients in $[0, q)$ as `uint16_t`. As in Sign, representatives modulo $2q$ are reconstructed coefficient by coefficient when needed and used immediately for high-bit extraction and norm computation.

The final challenge comparison in Verify does not require a full 32-bit polynomial representation. Since the challenge coefficients are binary, we store them in a 256-bit bitset, reducing this state from 1 024 B to 32 B. In Low-Stack, the column workspace and final challenge have non-overlapping lifetimes and share a 1 024 B slot. Narrowing only one of them leaves the shared slot at 1 024 B. By reducing the column workspace to 512 B and representing the challenge as a bitset, we reduce the shared slot itself to 512 B. These changes preserve the order of challenge generation and random input consumption, so the final comparison yields the same verification result as in Low-Stack.

4.5 Modeling Peak Working Storage

To summarize the effects of the preceding designs on dominant working state, we calculate the working storage of each API from the storage widths and lifetimes of its objects, using the accounting scope defined in Section 3. We denote the sizes of a 32-bit polynomial, a 16-bit polynomial, and an 8-bit small-secret polynomial by $P = 1\,024$ B, $P_q = 512$ B, and $P_s = 256$ B, respectively. The fixed-point FFT scratch in KeyGen remains $F = 2\,048$ B. The streaming encoder states in Sign occupy $E_h = \{263, 383, 614\}$ B and $E_{hb} = \{404, 648, 776\}$ B for modes 2, 3, and 5, respectively. For Verify, we combine the 512 B high-bits/hint workspace, whose width remains unchanged, with 64 B of auxiliary state and set $V_{\text{aux}} = 576$ B. As in Section 3, we exclude hash states, scalars, small auxiliary arrays, and compiler-generated call frames from this model.

Table 2 shows that reducing coefficient storage width alone does not reduce peak working storage equally across all routines. In KeyGen modes 2 and 3, the 32-bit norm accumulator and row accumulator are simultaneously live, so replacing P with P_q directly reduces working storage. In mode 5, however, the two objects already have non-overlapping lifetimes, so $\max(P, P_q) = P$. As in the pass-aware state model of Section 3, the overall peak for Sign is determined by the sum of the persistent driver storage and the maximum pass-local storage among **Passes A**, **B**, **C1**, and **C2**. The pass that determines the peak may vary with the parameter set. In Verify, we retain the existing workspace sharing and narrow the representations of the row, column, and challenge-related state, reducing dominant polynomial storage from $3P$ to $3P_q$. Actual stack usage also reflects hash states and sampling buffers omitted from this model, the encoder-state layout, and compiler-generated frames. We compare the model with measured stack usage in Section 7.

Table 2: Width-aware minimum-state model for dominant working storage. Hash states, small auxiliary arrays, and call frames are excluded.

Operation	Low-Stack	This work
KeyGen	$F + 3P$, modes 2/3 $F + 2P$, mode 5	$F + P + P_q$, modes 2/3 $F + \max(P, P_q)$, mode 5
Sign driver	$2P$	$\max(P, 2P_q)$
Sign Pass A	$2P$	$2P_q$
Sign Pass B	$2P$	$P + P_s$, modes 2/3 $P + 2P_q$, mode 5
Sign Pass C	$2P + E_h$	$\max(P_q, E_{hb}) + P_q + E_h$
Verify	$3P + V_{\text{aux}}$	$3P_q + V_{\text{aux}}$

5 Exact Gaussian Replay with Compact Records

As described in Section 3.2, Low-Stack [KvVBS26] regenerates the Gaussian vector from the same seed and nonce when needed, rather than storing it. If a signing attempt reaches the post-acceptance stage, the same y_2 polynomial may therefore be generated up to five times during the scaling computation and **Pass A**, **B**, and **C1**. The **C1** reuse in Section 4 reduces this count to four, but each remaining regeneration still repeats the same CDT search and acceptance test. Rather than caching the entire Gaussian polynomial, we record only the candidate acceptance pattern and partial CDT ranks of accepted samples in compact form during the first generation. In subsequent passes, we consume the same randomness to replay it exactly. Following the output-buffer reuse in [BRS22], we place the records in the unused signature buffer, avoiding a additional large recording array.

5.1 Recording and Replaying Gaussian Decisions

The Gaussian sampler in HAETAE 1.2.0 [CCD+26] reads 26 B per candidate from the SHAKE stream to form the CDT input U, the acceptance-test input E, and the fractional part V of the magnitude. The original regeneration path compares all 166 CDT thresholds for each candidate to obtain the rank x, then repeats the acceptance test, which includes squaring and `approx_exp`. During the first generation, we follow this original path and record the acceptance decisions in candidate order, using 1 bit per candidate in a bitmap. For accepted samples, we additionally store only part of the CDT rank. During replay, we consume the same 26 B in the same order, skip computation for candidates whose bitmap bit is 0, and reconstruct magnitudes only for candidates whose bit is 1, using the stored rank information. Figure 2 illustrates this difference across the scaling computation and **Pass A**, **B**, and **C1** for the acceptance pattern $(0, 1, 0, 1, 1)$; if a record exceeds its limit, the original path is used (Section 5.3).

Even when replay skips computation for a rejected candidate, it must consume its randomness exactly as in the original sampler. For example, with the acceptance pattern $(0, 1, 0, 1, 1)$, the first and second samples come from candidates c_2 and c_4, respectively.

If replay does not consume the 26 B corresponding to a rejected candidate, the SHAKE stream position for subsequent candidates shifts, causing them to use different U, E, and V values. We therefore consume 26 B for every candidate in order, skipping only the CDT search and acceptance test when the bitmap bit is 0. This preserves the stream position of the original sampler while removing only the computational cost of rejected candidates.

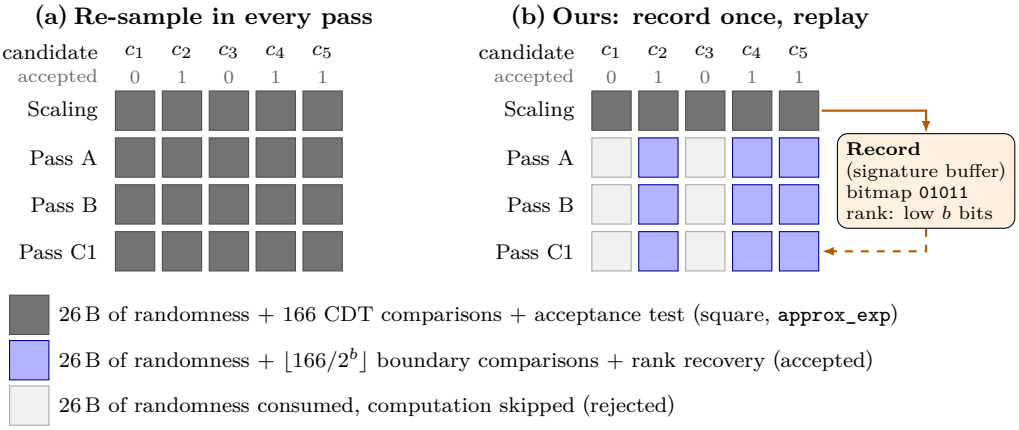


Figure 2: Cost of obtaining the same y_2 polynomial within a signing attempt.

5.2 Partial-Rank Reconstruction

The original sampler computes the rank $x = \sum_{i=0}^{165} \mathbf{1}\{U > T_i\}$, with $0 \le x \le 166$, for CDT thresholds $T_0, \ldots, T_{165}$ in ascending order. During the first generation, we record only the low b bits, $x_{\mathrm{lo}} = x \bmod 2^b$, instead of the full x. During replay, we compute

$$x_{\mathrm{hi}} = \sum_{j=1}^{\lfloor 166/2^b \rfloor} \mathbf{1}\{U > T_{j2^b-1}\}, \qquad \widehat{x} = 2^b x_{\mathrm{hi}} + x_{\mathrm{lo}}$$

by comparing only one boundary per 2^b thresholds. Since the thresholds are sorted, each boundary comparison determines whether $x \ge j2^b$, so $x_{\mathrm{hi}} = \lfloor x/2^b \rfloor$ and $\widehat{x} = x$. This reduces the number of CDT comparisons per accepted sample from 166 to 5, 20, and 10 in modes 2, 3, and 5, respectively ($b = 5, 3, 4$).

Replay performs boundary comparisons by traversing the entire boundary table for each mode in a fixed order, rather than selecting specific thresholds based on the stored x_{lo}. This adds no secret-dependent memory accesses based on the rank, and the number of comparisons is fixed by the public parameters.

5.3 Exactness and Fallback

For the same seed and nonce, replay generates the same sample sequence as the original Gaussian sampler. The bitmap preserves the acceptance decisions from the first generation, and partial-rank reconstruction exactly recovers the original CDT rank x for the same U. Since the randomness for every candidate is consumed in the same order, the SHAKE stream position also remains identical to that of the original regeneration path. However, a carry from rounding V in the magnitude $r = x\,2^{56} + \lfloor V/2^{16} \rceil$ may propagate to the high word. To obtain the rank to record, we remove this carry to recover the original x, rather than simply using the high bits of r. Replay applies the same carry for the same V before reconstructing the magnitude, so downstream computation, including hyperball scaling and coefficient rounding, uses the same values as the original regeneration path. Thus, replay not only preserves the sampling distribution but also reproduces the original sampler's output bit-exactly for the same seed and nonce.

For each mode, we use a record only if 256 samples are obtained within the candidate limit C. If the limit is exceeded, we mark the record as invalid and complete the first generation using the original sampler. Subsequent passes use the original regeneration path with the same seed and nonce instead of replay. Thus, C determines only the availability of the replay fast path, without limiting the sampling distribution.

Table 3: Signature buffer allocation and replay savings by mode. CDT/Gaussian/Sign: reduction over baseline (no y_2 records) in comparisons per Sign / Cortex-M4 (C) cycles.

Mode	b	C	y_1 cache	y_2 record	Used / signature	CDT	Gaussian	Sign
2	5	512	2/4	2/2	1,474 / 1,474 B	21.4%	13.2%	11.8%
3	3	320	4/6	2/3	2,322 / 2,349 B	12.1%	7.6%	6.6%
5	4	512	5/7	2/4	2,946 / 2,948 B	10.7%	6.7%	6.1%

5.4 Caching and Replay Strategy

Since the signature output buffer provided by the caller is unused until **Pass C2** writes the final signature, we reuse this space as temporary storage for replay. In **Pass A**, we cache the rounded coefficients of selected y_1 polynomials as 16-bit values and reuse them in subsequent row batches of **Pass A** and in **Pass C1** to avoid regeneration. For y_2 polynomials, whose unrounded values are required in **Pass B**, we store the replay records described above instead of the values themselves. Thus, we apply value caching to y_1 and exact replay to y_2 in the same output buffer, as each object's downstream use requires.

For a candidate limit C, a replay record for one y_2 polynomial occupies $S(b, C) = 1 + \lceil 256b/8 \rceil + \lceil C/8 \rceil$ bytes. The terms correspond to a 1-byte header, partial ranks for 256 accepted samples, and the acceptance bitmap in candidate order, respectively.

As shown in Table 3, we first allocate the signature buffer to y_1 caches in units of 512 B and place y_2 replay records in the remaining space. This allows us to record both y_2 polynomials in mode 2, whereas in modes 3 and 5, we record only the first two polynomials and use the original regeneration path for the rest. We choose b and the candidate limit C for each mode so that two records fit in the available buffer space. Both caches and records are used only before **Pass C2**, so their storage lifetimes do not overlap with that of the final signature output.

In mode 2, which records all y_2 polynomials, replay reduces CDT comparisons per Sign operation by 21.4% and Sign cycles in the Cortex-M4 C implementation by 11.8% (see Section 7.1 for the measurement setup). The reductions are smaller in modes 3 and 5, where only some polynomials are recorded (Table 3). Thus, replay reduces Gaussian generation cycles by reusing the signature buffer without requiring an additional buffer.

5.5 Side-Channel Considerations

Replay preserves the side-channel model of the original Gaussian sampler and introduces no additional secret-dependent branches or table address selection. Partial-rank reconstruction compares the boundaries specified for each mode in a fixed order, and record accesses are determined by the candidate and sample indices. We tested recording and replay on Cortex-M4 and M3 using dudect [RBV17], with 10^6 measurements per test. No timing difference was detected between fixed and random candidates with the same acceptance outcome ($|t| \leq 1.45 < 4.5$). Since MSP430 and AVR have no caches, execution time is determined by the instruction sequence. We therefore verified in the compiled firmware that the conditional branches introduced by recording and replay depend only on public counters and acceptance history.

6 ISA-Specific Arithmetic Optimizations

6.1 NTT and Modular Multiplication

With narrow representations, coefficients are stored in 16 bits, so butterfly results must be normalized before storage. Since $q = 64\,513 > 2^{15}$, 16-bit storage leaves no headroom

Table 4: Multiplication methods selected for each ISA

ISA	Name	Cycles	Layer merging
Cortex-M4	32-bit Plantard [HZZ+22] + 32-bit Montgomery [GKS21]	2 / 3	3 + 3 + 2
Cortex-M3	16-bit Montgomery [Sei18, GKS21]	8	2 + 2 + 2 + 2
Cortex-M3 (fast)	Lemire [LKK19]	5	2 + 2 + 2 + 2
MSP430	16-bit Montgomery [Sei18, SKK+26]	30	3 + 3 + 2
AVR	NTT-friendly Montgomery [DKZ+26]	31	2 + 2 + 2 + 2

for lazy reduction. The additional reduction overhead caused by narrower coefficient storage has also been noted in [BRS22]. The impact of this overhead varies across ISAs. On 32-bit architectures, existing 32-bit arithmetic efficiently handles wider intermediate values, so normalization for 16-bit storage incurs an additional cost. On 8-bit and 16-bit architectures, however, operations on the original 32-bit coefficients are costly, so the benefits of narrow representations, namely 16-bit arithmetic, can offset the additional normalization cost. We therefore adapt modular multiplication and layer merging to each ISA's multiplication instructions and register count to reduce NTT costs.

Montgomery and Barrett multiplication are commonly used for NTT twiddle products $a\omega \bmod q$. Here, $R = 2^\ell$ is coprime to q, and we refer to multiplication with $R = 2^\ell$ as ℓ-bit multiplication ($\mathrm{high}(x) = \lfloor x/R \rfloor$, $\mathrm{low}(x) = x \bmod R$). Subtractive Montgomery multiplication [Sei18, Hwa26] precomputes $z = \omega R \bmod q$ and computes

$$\mathrm{high}(t) - \mathrm{high}(mq) = \frac{t - mq}{R} \equiv a\omega \pmod q, \qquad t = az, \quad m = \mathrm{low}(t)\, q^{-1} \bmod R$$

using only the difference between the high halves, since $\mathrm{low}(t) = \mathrm{low}(mq)$. Barrett multiplication [BHK+21] computes $a\omega - \lfloor a\omega'/R \rceil q \equiv a\omega \pmod q$ using precomputed ω and $\omega' \approx \omega R/q$. Since $q = 64\,513 = k \cdot 2^8 + 1$ ($k = 252$), we have $q^{-1} \equiv 1 - k \cdot 2^8 = 1 + 2^{10} \pmod{2^{16}}$, so the quotient m in 16-bit multiplication can be obtained using a shift and an addition or a byte multiplication by k.

Table 4 shows the multiplication methods selected for each ISA, considering transform costs in terms of cycle counts, twiddle tables, input ranges, and register usage. All candidates were implemented in assembly, measured on the respective boards, and exhaustively verified. Since $q = 64\,513 > 2^{15}$, the modulus requirements of prior 16-bit Plantard [HZZ+22, HZZ+24] and signed reduction [DKZ+26] are not satisfied. K-RED2X [LN16, DKZ+26] requires additional reduction to store intermediate results within the 16-bit range. Intermediate sums and differences in packed butterflies [BKS19] can also exceed the signed halfword range, making it difficult to apply the prior instruction sequence directly. On MSP430 and AVR, whose registers are at most 16 bits wide, 32-bit Plantard, Montgomery, Barrett, and Lemire products are decomposed into multiple multiplications. The twiddle table size also doubles for methods other than Montgomery.

Cortex-M4. Both Plantard and Lemire can be implemented with two instructions on Cortex-M4. Given the signed intermediate range and the normalization cost, we combine a Plantard variant using `MUL` and `SMMULR` with Montgomery multiplication. For $R = 2^{32}$, if the two products share the signed low word $\bar{m} \equiv -tq^{-1} \pmod R$, the relation

$$P = M + \left\lfloor \frac{R/2 - t}{R} \right\rfloor, \qquad P = M \iff -R/2 < t \le R/2 \tag{2}$$

holds between the Plantard result $P = \mathrm{high}(\bar{m}q + R/2)$ and the Montgomery result $M = (t + \bar{m}q)/R$ [Hwa26]. We therefore use Plantard for the first two layers, where t lies in this interval, and Montgomery multiplication in the next layer to form a group of three layers. This produces the same result as the original Montgomery multiplication.

Based on this grouping and the register allocation for HAETAE in [KRSS19], we apply $3+3+2$ layer merging. Immediately before each store, we apply Barrett reduction with `SMMULR` [AHKS22] to obtain signed 16-bit representatives, so t lies within the interval in Equation (2) in the first two layers of each group. In each group of three layers, we keep eight coefficients in general-purpose registers and seven twiddles in FPU registers.

Cortex-M3. On Cortex-M3, `MUL` takes 1 cycle, whereas the cycle counts of `UMULL`, `SMULL`, `UMLAL`, and `SMLAL` depend on the inputs because of early termination. The default M3 assembly implementation therefore avoids long multiplication instructions for constant-time execution, as in [HKS25]. Since $q < 2^{16}$, both t and mq fit within the range of 32-bit `MUL`, and m is obtained with a shift and an addition. For 16-bit multiplication, subtractive Montgomery and standard Barrett [HKS25] have the same cost per product, so the choice depends on the constant representation. Barrett requires two constants per twiddle, so packed constants must be unpacked at each use when merging layers with limited registers. Montgomery needs only the single constant z, so we choose Montgomery multiplication. M3-fast, which allows variable-time execution, applies Lemire multiplication [LKK19] to fixed twiddles. For $C_\omega = \lceil 2^{32}\omega/q \rceil$ and $R = 2^{32}$, it computes $\mathrm{high}(\mathrm{low}(aC_\omega)\, q) = a\omega \bmod q$ using `MUL` and `UMULL`. Although this value equals the floor Barrett result $a\omega - \lfloor aC_\omega/2^{32} \rfloor q$ with the same C_ω, the remainder is obtained directly from the low half rather than through the quotient, so a single constant suffices to produce a canonical output without ω. For canonical inputs $0 \le a, \omega < q$, $aq \le 2^{32}$ is a sufficient condition for the identity above; this condition always holds for HAETAE's input range. We use $2+2+2+2$ layer merging, with four coefficients, three twiddles, q, a pointer, scratch, and a loop sentinel occupying all 14 registers without spills.

MSP430. Since MSP430's MPY32 is a memory-mapped peripheral, each multiplication is a transaction that writes operands to the multiplier registers and reads the result. The number of transactions and register transfers determine the cost [SKK+26]. We use 16-bit subtractive Montgomery multiplication, which has the shortest instruction sequence among the candidates that retain a one-word twiddle. From t produced by the first transaction, we compute m using a shift and an addition without invoking the multiplier. We then use mq from the second transaction to compute only the high-word difference $\mathrm{high}(t) - \mathrm{high}(mq) \in (-q, q)$. Since $q > 2^{15}$, we form a mask for adding q from the unsigned borrow of the subtraction rather than the sign bit of the difference. No low-word subtraction, 17th-bit handling, or sign reversal is needed, giving the shortest instruction sequence after the transactions. In each group of three layers, eight coefficients, a data pointer, a twiddle cursor, and two scratch registers occupy all 12 general-purpose registers.

AVR. AVR constructs 16-bit products from partial products of 8-bit `MUL` instructions and carry chains, so the number of `MUL` instructions and the carry chain length determine the cost. We use 16-bit subtractive Montgomery multiplication adapted from the NTT-friendly Montgomery method in [DKZ+26]. Using the byte structure of q, we obtain $\mathrm{high}(mq)$ with `MUL` operations involving k. The forward NTT, inverse NTT, and pointwise products use the same multiplication instruction sequence. In both transform directions, we apply $2+2+2+2$ layer merging, keeping four coefficients (8 bytes) in registers for each pair of layers and using all 32 registers without spills. Let $t = (t_3 t_2 t_1 t_0)_{2^8}$ and $kt_0 = (p_1 p_0)_{2^8}$. From $q^{-1} \equiv 1 - k \cdot 2^8$, the low byte of m is t_0 and the high byte is $m_1 = (t_1 - p_0) \bmod 2^8$. For byte 1 of mq, we have $m_1 + p_0 \equiv t_1$, and the carry from this addition equals the borrow c from the subtraction above. Thus, the relation

$$\mathrm{high}(mq) = k\, m_1 + p_1 + c \tag{3}$$

holds. The NTT-friendly Montgomery method in [DKZ+26] computes m_1 and high(mq) separately and uses three `MUL` instructions for reduction. We compute kt_0 only once and obtain both m_1 and c with a single `sub` that subtracts p_0 from t_1. We then subtract the borrow directly from high(t) through a `sbc` chain, reducing the number of `MUL` instructions for reduction to two. Since AVR's `MUL` overwrites the carry flag, the combined computation must consume this borrow before the next `MUL`. Equation (3) also holds for other moduli $q = k \cdot 2^8 + 1$ ($0 < k < 2^8$). Compared with our baseline implementation, which computes m using a shift and an addition, we reduce the cost per product from 37 cycles to 31 cycles by combining the formation of intermediate values with borrow handling.

6.2 FFT and Sampler

The FFT and Gaussian sampler use integer fixed-point arithmetic, so the modular reductions above do not apply, and the original rounding must be reproduced exactly. The core FFT multiplication is $\lfloor ab/2^{16} \rceil = \lfloor (ab + 2^{15})/2^{16} \rfloor$, and this rounding is applied separately to each integer product within a complex multiplication. Since $\mathrm{round}(x) + \mathrm{round}(y) \neq \mathrm{round}(x + y)$ can hold, we cannot combine two products into a single MAC and round only once. On M4 and M3, we merge two FFT layers; on M4, we also use FPU registers to store integer twiddles. On MSP430, we also merge two layers but store some intermediate values in an array because the registers cannot hold all four complex numbers. On AVR, we implement the same rounded multiplication using byte partial products.

The `smulh48` function in the HAETAE 1.2.0 [CCD+26] sampler computes $S(a, b) = \lceil ab/2^{48} \rceil$ for $-2^{48} \leq a < 2^{48}$ and $0 \leq b < 2^{48}$, rather than rounding to nearest. Let $s \in \{0, 1\}$ be the sign bit of a and set $U = a + s \cdot 2^{48} = a \bmod 2^{48}$. The identity

$$S(a, b) = \left\lfloor \frac{Ub + 2^{48} - 1}{2^{48}} \right\rfloor - sb \tag{4}$$

shows that the result is $\lfloor ab/2^{48} \rfloor$, incremented by 1 only when the discarded low 48 bits are nonzero. Always adding 1 after truncating the low part would be incorrect when that part is zero, so the carry from the low part must be preserved throughout the computation. The official reference implementation without `int128` and the previous M4 assembly implementation (`SMULL`/`SMLAL`) compute $S(a, b)$ by splitting each of a and b into two signed 24-bit limbs and forming four 64-bit partial products. This uses variable-time `UMULL` on M3 and 64-bit integer arithmetic on MSP430 and AVR. We separate the sign as in Equation (4) and split U into unsigned limbs matching each ISA's multiplier width (32 bits for M4, 16 bits for M3 and MSP430, and 8 bits for AVR). We accumulate partial products column by column and distribute $2^{48} - 1$, the bias for rounding up, among the column carries to reproduce exactly the rounding contribution from the discarded low part. The order of coefficient magnitudes fixes the signs of the Horner states in `approx_exp` to $+, -, +, \ldots$, so we apply sign corrections without branches only at predetermined stages, while preserving the 49-bit result of the final stage. The assembly implementations of CDT search and partial-rank reconstruction evaluate $U > T_i$ using the borrow from $T_i - U$, so they produce the same rank as the original sampler even at the boundary $U = T_i$.

6.3 Arithmetic Performance

Table 5 reports the results of arithmetic optimization on each ISA. In the C implementations before assembly optimization, 16-bit storage has opposite effects across ISAs. On Cortex-M4 and M3, the NTT of Ours (C) is slower than the 32-bit C implementation in [KvVBS26] because of normalization costs. On MSP430 and AVR, however, the NTT of Ours (C) is faster because multiplications and carry chains are shorter with 16-bit coefficients. Assembly optimization for each ISA yields NTT speedups of 2.84–5.52 over Ours

Table 5: Arithmetic cycles per ISA (board measurement, median of 100 runs, `-O3`). NTT, iNTT, pointwise, FFT: one polynomial; `approx_exp`: 64 calls. Ours (C)/(asm): our 16-bit C/assembly; in parentheses, speedup over Ours (C). Ours (M3-fast): variable-time.

	Impl.	NTT	iNTT	Pointwise	FFT	`approx_exp`
Cortex-M4	[KvVBS26] (C)	36,180	38,124	4,978	72,226	21,917
	[KvVBS26] (asm)	8,161	8,507	–	–	–
	Ours (C)	49,904	52,639	5,997	71,972	28,703
	Ours (asm)	9,034 (×5.52)	9,088 (×5.79)	2,390 (×2.51)	27,298 (×2.64)	13,811 (×2.08)
Cortex-M3	[KvVBS26] (C)	45,604	51,618	7,255	93,243	32,585
	Ours (C)	53,957	56,201	6,485	93,243	35,141
	Ours (asm)	19,024 (×2.84)	20,114 (×2.79)	5,437 (×1.19)	49,428 (×1.89)	31,456 (×1.12)
	Ours (M3-fast)	15,756 (×3.42)	16,591 (×3.39)	5,437 (×1.19)	41,963 (×2.22)	22,078 (×1.59)
MSP430	[KvVBS26] (C)	742,186	1,035,576	176,518	1,083,363	1,051,975
	Ours (C)	198,614	245,507	29,339	1,083,363	1,465,011
	Ours (asm)	52,137 (×3.81)	51,649 (×4.75)	13,033 (×2.25)	129,754 (×8.35)	88,184 (×16.61)
AVR	[KvVBS26] (C)	483,829	860,466	158,020	1,325,012	1,030,188
	[KvVBS26] (asm)	305,052	433,409	82,494	–	–
	Ours (C)	209,963	327,619	31,306	1,325,012	518,995
	Ours (asm)	62,971 (×3.33)	76,699 (×4.27)	11,930 (×2.62)	227,051 (×5.84)	159,694 (×3.25)

[KvVBS26] (asm): Cortex-M4 and AVR only (–: no separate assembly code).

(C). On M4, however, the NTT and iNTT require more cycles than their counterparts in the previous 32-bit assembly implementation in [KvVBS26]. Although our multiplications are cheaper, [KvVBS26] stores 32-bit butterfly outputs without reduction, whereas we reduce every coefficient before each 16-bit store, and on M4 narrower coefficients do not shorten multiplication. On AVR, by contrast, the NTT and iNTT achieve speedups of 4.84 and 5.65, respectively, over those in [KvVBS26]. Ours (asm) also achieves speedups of 1.89–8.35 for the FFT and 1.12–16.61 for `approx_exp` over Ours (C). For `approx_exp`, the two C implementations differ in `smulh48`. As in the reference implementation (Section 6.2), [KvVBS26] forms four 64-bit products of 24-bit limbs, whereas our common C splits both operands into unsigned 16-bit limbs and forms nine 16×16-bit products. As a result, Ours (C) is slower than [KvVBS26] (C) on Cortex-M4, M3, and MSP430 but faster on AVR.

7 Evaluation

This section evaluates our implementation. Following [KvVBS26], we divide the devices into mid-range (Cortex-M4 and M3) and low-end (MSP430 and AVR) groups and compare stack usage and execution time on each device. Table 6 lists the evaluated devices.

7.1 Experimental Setup

Baselines. HAETAE version 1.2.0 was recently released [CCD+26]. Since Low-Stack Implementation [KvVBS26] was based on an earlier version, we ported it to HAETAE 1.2.0 and remeasured it using the same harness. All measured implementations, including ours, produce outputs byte-for-byte identical to those of the HAETAE 1.2.0 reference implementation. Low-Stack's sampler assembly also targets the previous CDT, so we rewrote it for the 1.2.0 CDT using the same approach (a constant-time full scan). On each device, ref, Low-Stack, and Ours use the same Keccak implementation. We use the assembly implementation of [KRSS19] on Cortex-M4 and a C implementation elsewhere.

Table 6: Target devices. Clock: frequency of the cycle measurements.

	32-bit Cortex-M4	32-bit Cortex-M3	16-bit MSP430	8-bit AVR
Board	NUCLEO-L4R5ZI	NUCLEO-F207ZG	MSP-TS430PEU128*	Arduino Mega 2560
Clock (MHz)	20	16	8.192	16
Flash (KiB)	2,048	1,024	512	256
SRAM (KiB)	640	128	32	8
Compiler	Arm GNU 13.3.1	Arm GNU 13.3.1	MSP430 GCC 9.3.1.11	AVR GCC 15.2.0

*MCU: MSP430F67791.

Measurement. On Cortex-M4 and M3, we follow the pqm4 measurement methodology and report the median over 100 random inputs for comparison with [KvVBS26]. Since MSP430 and AVR lack a common measurement framework such as pqm4, we follow the method of [RPA+26], which accounts for timing variation due to rejection. For KeyGen and Sign, we report the mean over 10 inputs whose average repetition counts match the expected values (10 for KeyGen and 6/5/6 for Sign). For Verify, we report the median when verifying the outputs produced from these inputs. The times in ms in the tables are converted from cycle counts using the clock frequencies in Table 6. As in [KvVBS26], all cycle, stack, and code-size measurements use `-O3` on every device. On mid-range devices, we measure the peak stack of the API call. On low-end devices, we measure the peak SRAM demand of the API (e.g., $|\mathsf{sk}| + |\mathsf{sig}|$ + Sign stack), as in [KvVBS26].

7.2 Results on Mid-range Microcontrollers

Cortex-M4. Table 7 reports performance on Cortex-M4. Our implementation reduces stack usage relative to [KvVBS26] by 20.6–25.2% for KeyGen, 29.0–33.8% for Sign, and 31.9–32.2% for Verify. These reductions are the same in C and assembly and result from 16-bit storage and shorter object lifetimes (Section 4). Compared with [KvVBS26] (asm), Ours (asm) achieves speedups of 1.23–1.54 for KeyGen and 1.18–1.32 for Sign, while Verify performance is nearly identical. Our NTT is slower than that of [KvVBS26] (asm) (Table 5), but Gaussian generation dominates our Sign cycles (85.6–88.9%), which we reduce with fewer regenerations, replay, and the assembly sampler. Compared with ref (C), Ours (asm) reduces stack usage by 82.0–92.8% for KeyGen, 95.5–97.4% for Sign, and 91.3–95.7% for Verify. However, Sign still requires 2.10–2.45 times as many cycles, reflecting the cost of regenerating the Gaussian vector in each pass to reduce memory usage. Our implementation reduces part of this overhead through replay (Section 5). Verify also achieves a speedup of 1.12–1.30 over ref (C). [CSS26] optimizes HAETAE on Cortex-M4 but is based on HAETAE 1.1.2, so we list its reported results for reference only.

Cortex-M3. Since [KvVBS26] provides no assembly implementation for Cortex-M3, we compare against its C implementation. Long multiplication instructions on Cortex-M3 (e.g., `UMULL`) are variable-time, so we provide two assembly configurations. M3 avoids long multiplication on secret values, including in the common fixed-point C code, whereas M3-fast permits it. As shown in Table 8, Ours reduces stack usage by 19.9–24.3% for KeyGen, 27.9–32.6% for Sign, and 30.4–30.6% for Verify. Ours (asm) achieves speedups of 1.34–1.39 for KeyGen, 1.37–1.57 for Sign, and 1.16–1.19 for Verify. M3-fast achieves speedups of 1.04–1.07 for KeyGen and Sign and 1.02 for Verify over M3, so avoiding long multiplication incurs at most 7% overhead at the API level.

Table 7: Performance on Cortex-M4 (K = 10^3). Ratio: cycles / ref (C) cycles. `.text`: library code including read-only data. (C)/(asm): arithmetic in C/assembly.

			KeyGen			Sign			Verify			
		Impl.	Cycles	Ratio	Stack	Cycles	Ratio	Stack	Cycles	Ratio	Stack	`.text`
HAETAE	2	ref (C)	6,693	1.00	24,084	43,373	1.00	85,372	1,102	1.00	37,816	26,224
		[CSS26] (asm)	3,566	–	18,816	23,327	–	55,724	926	–	23,328	–
		[KvVBS26] (C)	11,054	1.65	5,808	156,496	3.61	5,836	1,100	1.00	4,824	37,830
		[KvVBS26] (asm)	9,219	1.38	5,808	107,191	2.47	5,840	855	0.78	4,824	45,158
		Ours (C)	11,559	1.73	4,344	132,437	3.05	3,864	1,206	1.09	3,272	29,982
		Ours (asm)	7,028	1.05	4,344	91,104	2.10	3,864	848	0.77	3,272	43,274
	3	ref (C)	10,476	1.00	41,492	49,633	1.00	128,756	1,996	1.00	60,640	26,672
		[CSS26] (asm)	5,759	–	30,096	29,810	–	79,348	1,772	–	31,916	–
		[KvVBS26] (C)	20,394	1.95	5,808	232,815	4.69	6,020	2,273	1.14	4,848	38,390
		[KvVBS26] (asm)	16,550	1.58	5,808	160,054	3.22	6,024	1,724	0.86	4,848	45,718
		Ours (C)	21,791	2.08	4,344	180,150	3.63	4,088	2,529	1.27	3,296	31,278
		Ours (asm)	13,409	1.28	4,344	121,740	2.45	4,088	1,734	0.87	3,296	44,906
	5	ref (C)	9,551	1.00	52,748	80,318	1.00	161,868	2,460	1.00	76,280	26,736
		[CSS26] (asm)	5,197	–	29,016	43,636	–	103,988	2,341	–	37,224	–
		[KvVBS26] (C)	10,977	1.15	4,768	357,799	4.45	6,004	2,764	1.12	4,840	38,238
		[KvVBS26] (asm)	10,128	1.06	4,768	241,205	3.00	6,008	2,214	0.90	4,840	45,566
		Ours (C)	11,401	1.19	3,776	278,844	3.47	4,264	3,007	1.22	3,296	30,074
		Ours (asm)	6,587	0.69	3,784	183,353	2.28	4,264	2,197	0.89	3,296	43,294

Table 8: Performance on Cortex-M3 (K = 10^3). M3-fast: variable-time `UMULL`/`SMULL` on secret data.

			KeyGen			Sign			Verify		
		Impl.	Stack	Cycles	ms	Stack	Cycles	ms	Stack	Cycles	ms
HAETAE	2	[KvVBS26] (C)	6,028	17,289	1,081	6,060	224,047	14,003	5,044	1,865	117
		Ours (C)	4,564	16,663	1,041	4,084	192,532	12,033	3,500	1,888	118
		Ours (asm)	4,564	12,541	784	4,108	163,674	10,230	3,500	1,581	99
		Ours (M3-fast)	4,564	12,017	751	4,084	153,973	9,623	3,500	1,551	97
	3	[KvVBS26] (C)	6,028	32,468	2,029	6,244	332,743	20,796	5,060	3,787	237
		Ours (C)	4,564	31,883	1,993	4,316	257,170	16,073	3,516	3,887	243
		Ours (asm)	4,564	24,315	1,520	4,316	215,533	13,471	3,524	3,192	200
		Ours (M3-fast)	4,564	23,412	1,463	4,316	202,563	12,660	3,516	3,131	196
	5	[KvVBS26] (C)	4,988	15,635	977	6,228	514,924	32,183	5,060	4,722	295
		Ours (C)	3,996	15,763	985	4,484	397,212	24,826	3,524	4,788	299
		Ours (asm)	3,996	11,280	705	4,492	328,322	20,520	3,524	4,080	255
		Ours (M3-fast)	3,996	10,586	662	4,484	307,953	19,247	3,524	4,016	251

7.3 Results on Low-end Microcontrollers

MSP430. Since [KvVBS26] provides no assembly implementation for MSP430, we compare against its C implementation. As shown in Table 9, Ours reduces Stack by 2.5–9.2% for KeyGen, 14.0–21.2% for Sign, and 15.0–20.1% for Verify. The Stack values include I/O buffers whose sizes remain unchanged; the reductions in API stack usage alone are 5.4–13.3% for KeyGen, 26.9–31.2% for Sign, and 29.4–29.8% for Verify. In [KvVBS26], the 1 KiB copy passed by value to secret-key packing remains in the caller's frame throughout KeyGen on Cortex-M4, but exists only during packing on MSP430. Removing this copy on MSP430 shifts the peak to the C Keccak path for matrix generation, resulting in only a small reduction in KeyGen stack usage. Ours (asm) achieves speedups of 1.55–1.90 for KeyGen, 2.17–2.51 for Sign, and 1.27–1.32 for Verify. For HAETAE-2 Sign, the best observed time (one repetition) and the worst observed time (for the input with the largest repetition count among 2^{20} inputs) are 0.24 and 11.4 times the mean, respectively. The

Table 9: Performance on MSP430 (K = 10^3). Flash: Verify image.

			KeyGen			Sign			Verify		
	Impl.	Flash	Stack	Cycles	ms	Stack	Cycles	ms	Stack	Cycles	ms
HAETAE 2	[KvVBS26] (C)	84,442	7,736	365,438	44,609	8,942	8,037,147	981,097	7,560	32,833	4,008
	Ours (C)	95,224	7,024	290,688	35,484	7,050	6,827,560	833,442	6,044	26,876	3,281
	Ours (asm)	90,806	7,024	224,194	27,367	7,050	3,695,483	451,109	6,044	25,448	3,106
HAETAE 3	[KvVBS26] (C)	83,042	8,920	762,806	93,116	10,703	12,721,660	1,552,937	8,949	66,274	8,090
	Ours (C)	95,442	8,208	606,731	74,064	8,809	9,889,619	1,207,229	7,441	53,427	6,522
	Ours (asm)	92,796	8,208	491,633	60,014	8,809	5,156,518	629,458	7,441	50,276	6,137
HAETAE 5	[KvVBS26] (C)	83,678	9,118	269,529	32,902	11,912	18,492,373	2,257,370	10,154	84,905	10,364
	Ours (C)	96,076	8,888	251,413	30,690	10,240	14,275,816	1,742,653	8,628	70,222	8,572
	Ours (asm)	91,008	8,888	141,746	17,303	10,240	7,369,384	899,583	8,628	66,798	8,154

worst observed case has an occurrence probability of approximately 2^{-19}.

AVR. Our implementation supports KeyGen and Sign in mode 2. As shown in Table 10, Ours runs KeyGen and Sign within 8 KiB, with at least 771 B of Headroom for KeyGen and 1,173 B for Sign. In contrast, the static SRAM bounds for [KvVBS26] (C) in the same per-API firmware are 8,991 B for KeyGen and 9,028 B for Sign, both exceeding 8 KiB. For Verify, Stack is 18.8–23.2% lower than in [KvVBS26], and Headroom in HAETAE-5, the most constrained case, increases from 139 B to 1,636 B. For Verify, Ours (asm) achieves a speedup of 1.19–1.21 over [KvVBS26] (asm). For KeyGen and Sign, assembly provides speedups of 1.57 and 1.86, respectively, over Ours (C). For HAETAE-2 Sign, the best and worst observed times are 0.24 and 13.3 times the mean, respectively, and the worst observed case has an occurrence probability of approximately 2^{-23}. To confirm that the implementation fits within 8 KiB on a physical device, the integration firmware generates a fresh key, signs, and verifies the signature in a single run. The total SRAM usage, including 549 B of static data, is 8,070 B for Ours (C) and 8,074 B for Ours (asm), leaving at least 118 B of the 8 KiB SRAM free. For the same firmware, the static SRAM bounds are 9,571 B for the port of [KvVBS26], 9,290 B for that port with only object lifetime changes, and 9,368 B with only 16-bit storage. All three exceed 8 KiB, so both techniques must be combined to guarantee that the implementation fits within 8 KiB. To our knowledge, this is the first demonstration of HAETAE execution from key generation through verification on a physical device with 8 KiB of SRAM. The AVR Dilithium2 implementation in [HKS25] was evaluated in a simulator, and its reported Sign stack usage alone is 12,059 B, exceeding 8 KiB.

Table 10: Performance on AVR (K = 10^3). Stack: I/O buffers in SRAM + API stack (public key in Flash). Headroom: SRAM left after the static data of the harness. *Static SRAM bound: 8,991 B (KeyGen), 9,028 B (Sign).

		API	Impl.	Flash	Stack	Headroom	Cycles	ms
HAETAE	2	KeyGen	[KvVBS26] (C)			exceeds 8 KiB*		
			Ours (C)	51,112	6,941	775	204,621	12,789
			Ours (asm)	61,132	6,945	771	130,357	8,147
		Sign	[KvVBS26] (C)			exceeds 8 KiB*		
			Ours (C)	91,824	6,883	1,173	3,990,465	249,404
			Ours (asm)	87,050	6,883	1,173	2,145,048	134,065
		Verify	[KvVBS26] (C)	49,586	6,446	1,642	19,888	1,243
			[KvVBS26] (asm)	51,126	6,446	1,642	17,415	1,088
			Ours (C)	49,120	4,948	3,140	16,133	1,008
			Ours (asm)	50,806	4,950	3,138	14,621	914
	3	Verify	[KvVBS26] (C)	52,046	7,343	745	39,627	2,477
			[KvVBS26] (asm)	53,586	7,343	745	34,450	2,153
			Ours (C)	51,742	5,848	2,240	31,660	1,979
			Ours (asm)	53,428	5,850	2,238	28,419	1,776
	5	Verify	[KvVBS26] (C)	52,694	7,949	139	50,525	3,158
			[KvVBS26] (asm)	54,480	7,944	144	44,410	2,776
			Ours (C)	51,912	6,452	1,636	40,977	2,561
			Ours (asm)	53,578	6,454	1,634	37,468	2,342

References

[ADE+25] Diego F. Aranha, Johan Degn, Jonathan Eilath, Kent Nielsen, and Peter Scholl. FAEST for memory-constrained devices. Cryptology ePrint Archive, Paper 2025/1261, 2025.

[AHKS22] Amin Abdulrahman, Vincent Hwang, Matthias J Kannwischer, and Amber Sprenkels. Faster kyber and dilithium on the cortex-m4. In *International Conference on Applied Cryptography and Network Security*, pages 853–871. Springer, 2022.

[BBB+24] Slim Bettaieb, Loïc Bidoux, Alessandro Budroni, Marco Palumbi, and Lucas Pandolfo Perin. Enabling perk and other mpc-in-the-head signatures on resource-constrained devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):84–109, 2024.

[BBC+23] Joppe W. Bos, Olivier Bronchain, Frank Custers, Joost Renes, Denise Verbakel, and Christine van Vredendaal. Enabling FrodoKEM on embedded devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2023(3):74–96, 2023.

[BEK14] Carsten Bormann, Mehmet Ersue, and Ari Keränen. Terminology for constrained-node networks. RFC 7228, Internet Engineering Task Force, May 2014.

[BF26] Ryad Benadjila and Thibauld Feneuil. Breaking the myth of mpcith inefficiency: Optimizing mqom for embedded platforms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(3):279–305, 2026.

[BHK+21] Hanno Becker, Vincent Hwang, Matthias J. Kannwischer, Bo-Yin Yang, and Shang-Yi Yang. Neon ntt: Faster dilithium, kyber, and saber on cortex-a72 and apple m1. *IACR Transactions on Cryptographic Hardware*

and Embedded Systems, 2022(1):221–244, Nov. 2021. Artifact available at https://artifacts.iacr.org/tches/2022/a2.

[BKS19] Leon Botros, Matthias J Kannwischer, and Peter Schwabe. Memory-efficient high-speed implementation of kyber on cortex-m4. In *International Conference on Cryptology in Africa*, pages 209–228. Springer, 2019.

[BRS22] Joppe W Bos, Joost Renes, and Amber Sprenkels. Dilithium for memory constrained devices. In *International Conference on Cryptology in Africa*, pages 217–235. Springer, 2022.

[BZB+22] Gustavo Banegas, Koen Zandberg, Emmanuel Baccelli, Adrian Herrmann, and Benjamin Smith. Quantum-resistant software update security on low-power networked embedded devices. In *International Conference on Applied Cryptography and Network Security*, pages 872–891. Springer, 2022.

[CCD+24] Jung Hee Cheon, Hyeongmin Choe, Julien Devevey, Tim Güneysu, Dongyeon Hong, Markus Krausz, Georg Land, Marc Möller, Damien Stehlé, and Min-June Yi. Haetae: Shorter lattice-based fiat-shamir signatures. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(3):25–75, Jul. 2024.

[CCD+26] Jung Hee Cheon, Hyeongmin Choe, Julien Devevey, Tim Güneysu, Dongyeon Hong, Minhyeok Kang, Taekyung Kim, Jeongbeen Ko, Markus Krausz, Georg Land, Marc Möller, Junbum Shin, Damien Stehlé, and MinJune Yi. HAETAE: Shorter lattice-based fiat-shamir signatures, Sep. 2026. HAETAE Specification, Version 1.2.0.

[CSS26] JunHyeok Choi, DongHyun Shin, and Seog Chung Seo. Optimizing haetae and smaug-t on cortex-m4 for resource-constrained iot devices. *IEEE Internet of Things Journal*, 2026.

[DDLL13] Léo Ducas, Alain Durmus, Tancrède Lepoint, and Vadim Lyubashevsky. Lattice signatures and bimodal gaussians. In *Annual Cryptology Conference*, pages 40–56. Springer, 2013.

[DKZ+26] Yifan Dong, YoungBeom Kim, Jieyu Zheng, Zhichuang Liang, Boyue Fang, Seog Chung Seo, Maire O' Neill, and Yunlei Zhao. Lightweight pq kem and hybrid mqtt protocol for 8-bit avr sensor nodes. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):372–408, 2026.

[GKS21] Denisa OC Greconici, Matthias J Kannwischer, and Amber Sprenkels. Compact dilithium implementations on cortex-m3 and cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–24, 2021.

[HCK+26] Yulim Hyoung, Subeen Cho, Uijae Kim, Minwoo Lee, Hwajeong Seo, and Minjoo Sim. Memory-efficient implementation of smaug-t and haetae. *Cryptology ePrint Archive*, 2026.

[HKS25] Vincent Hwang, YoungBeom Kim, and Seog Chung Seo. Multiplying polynomials without powerful multiplication instructions. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2025(1):160–202, 2025.

[HMK+26] Harry Hart, Puja Mondal, Suparna Kundu, Supriya Adhikary, Angshuman Karmakar, and Chaoyun Li. Lightening cross: Memory optimized implementations of cross. *IACR transactions on cryptographic hardware and embedded systems*, 2026(2):630–658, 2026.

[HSZ+26] Junhao Huang, Yiteng Sun, Jipeng Zhang, Weijia Wang, Donglong Chen, Haiyang Xue, and Guomin Yang. Efficient and compact high-order masking Raccoon on memory constrained devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(4):369–396, 2026.

[Hwa26] Vincent Bert Hwang. *Software Implementations of Polynomial Multiplications for Lattice-Based Cryptosystems*. PhD thesis, Radboud University, 2026.

[HZZ+22] Junhao Huang, Jipeng Zhang, Haosong Zhao, Zhe Liu, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Improved plantard arithmetic for lattice-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022(4):614–636, 2022.

[HZZ+24] Junhao Huang, Haosong Zhao, Jipeng Zhang, Wangchen Dai, Lu Zhou, Ray CC Cheung, Çetin Kaya Koç, and Donglong Chen. Yet another improvement of plantard arithmetic for faster kyber on low-end 32-bit iot devices. *IEEE Transactions on Information Forensics and Security*, 19:3800–3813, 2024.

[KRSS19] Matthias J Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Testing and benchmarking nist pqc on arm cortex-m4. *Cryptology ePrint Archive*, 2019.

[KS25] YoungBeom Kim and Seog Chung Seo. An optimized instantiation of post-quantum mqtt protocol on 8-bit avr sensor nodes. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 248–266, 2025.

[KvVBS26] YoungBeom Kim, Christine van Vredendaal, Gustavo Banegas, and Seog Chung Seo. Low-stack haetae for memory-constrained microcontrollers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(4):293–315, Sep. 2026.

[LKK19] Daniel Lemire, Owen Kaser, and Nathan Kurz. Faster remainder by direct computation: Applications to compilers and software libraries. *Software: Practice and Experience*, 49(6):953–970, 2019.

[LN16] Patrick Longa and Michael Naehrig. Speeding up the number theoretic transform for faster ideal lattice-based cryptography. In *International Conference on Cryptology and Network Security*, pages 124–139. Springer, 2016.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In *International conference on the theory and application of cryptology and information security*, pages 598–616. Springer, 2009.

[Nat24] National Institute of Standards and Technology. Module-Lattice-Based Digital Signature Standard (ML-DSA). Technical Report FIPS 204, National Institute of Standards and Technology, 2024.

[OKG+12] Leonardo B Oliveira, Aman Kansal, Conrado PL Gouvêa, Diego F Aranha, Julio López, Bodhi Priyantha, Michel Goraczko, and Feng Zhao. Secure-tws: Authenticating node to multi-user communication in shared sensor networks. *The Computer Journal*, 55(4):384–396, 2012.

[RBV17] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1697–1702. IEEE, 2017.

[RPA+26] Sebastien Riou, Jong-Yeon Park, Liga Anwar, Axel Poschmann, and Michael Hutter. Apples, oranges, and signatures: Pitfalls and methodology in ml-dsa benchmarking. In *International Workshop on Migration and Agility in Cryptographic Systems*, pages 121–142. Springer, 2026.

[Sei18] Gregor Seiler. Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography. *Cryptology ePrint Archive*, 2018.

[SGB+26] Jonas Schupp, Marco Gianvecchio, Alessandro Barenghi, Patrick Karl, Gerardo Pelosi, and Georg Sigl. Optimizing the post quantum signature scheme cross for resource constrained devices. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(3):224–246, Jul. 2026.

[SKK+26] DongHyun Shin, YoungBeom Kim, Ayesha Khalid, Maire O' Neill, and Seog Chung Seo. Optimized implementations of keccak, kyber, and dilithium on the msp430 microcontroller. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):409–440, 2026.

[TDF+23] George Tasopoulos, Charis Dimopoulos, Apostolos P Fournaris, Raymond K Zhao, Amin Sakzad, and Ron Steinfeld. Energy consumption evaluation of post-quantum tls 1.3 for resource-constrained embedded devices. In *Proceedings of the 20th ACM International Conference on Computing Frontiers*, pages 366–374, 2023.