Low-Stack HAETAE for Memory-Constrained Microcontrollers

Gustavo Banegas[1], YoungBeom Kim[2], Seog Chung Seo[2] and Christine van Vredendaal[3]

[1] LIX, CNRS, Inria, École Polytechnique, Institut Polytechnique de Paris, France
gustavo.souza-banegas@inria.fr
[2] Kookmin University, Seoul, Republic of Korea, {darania,scseo}@kookmin.ac.kr
[3] NXP Semiconductors, The Netherlands, christine.cloostermans@nxp.com

Abstract. We present a low-stack implementation of the module-lattice signature scheme HAETAE, targeting microcontrollers with 8 kB–16 kB of available SRAM. On such devices, peak stack usage is often the binding constraint, and HAETAE's hyperball-based sampler, large transient polynomial vectors, and variable-length signature payloads (hint and high-bits arrays) pose a particular challenge. To address this we introduce (i) *Rejection-aware pass decomposition*, which isolates encoding to the post-acceptance path; (ii) *Component-level early rejection*, which short-circuits the response computation when a partial norm already exceeds the bound; and (iii) *Reverse-order streaming entropy coding* using range Asymmetric Numeral Systems (rANS), which eliminates full hint and high-bits staging buffers. Combined with streamed matrix generation, a two-pass hyperball sampler with streaming Gaussian backend, and row-streamed verification, these techniques bring Signing stack from 71 kB–141 kB in the reference implementation down to 5.8 kB–6.0 kB, key generation to 4.7 kB–5.7 kB, and verification to 4.7 kB–4.8 kB across all three security levels. Our pure C implementation covers all three security levels (HAETAE-2/3/5), whose optimization paths differ due to the public-key domain (d>0 vs. d=0) and rejection structure. We implement our optimization on a Nucleo-L4R5ZI and compare to the reference `pqm4` (for HAETAE-2 and -3) and a recently published memory-optimized implementation (targeting HAETAE-5 only). We reduce HAETAE-2, -3, and -5 stack by respectively 75, 86 and 8 % for key generation, 92, 95 and 24 % for signature generation, and 85, 91 and 22 % for verification. Depending on the parameter set, this impacts performance by at most a factor 1.8 and 3.4 for key and signature generation respectively, while even offering a performance improvement up to 18 % for verification. Verification at all security levels fits within 8 kB of RAM (signature buffer + stack) and is 2.34–3.34× faster than ML-DSA m4fstack at each comparable security level. We additionally validate portability under RIOT-OS on ARM Cortex-M4 and RISC-V targets.

Keywords: HAETAE · lattice-based cryptography · small memory signatures · constrained devices

1 Introduction

Embedded systems rely on public-key authentication schemes for firmware updates, secure boot, device attestation, and device provisioning. With the transition to post-quantum

*Author list in alphabetical order; see https://ams.org/profession/leaders/CultureStatement04.pdf. This work was supported by the HYPERFORM consortium, funded by France through Bpifrance, and by the France 2030 program under grant agreement ANR-22-PETQ-0008 PQ-TLS. Date of this document: 2026-04-01.

cryptography, signature schemes based on module lattices are leading candidates for standardization.

On low-end ARM Cortex-M and RISC-V microcontrollers, the stack available to an application may be only 8 kB–16 kB once the real-time operating system (RTOS), network stack, and I/O drivers are loaded. For this class of targets, peak stack usage is often the binding constraint rather than code size or cycle count.

HAETAE [CCD+23, CCD+24], a module-lattice signature scheme presented at CHES'24, achieves shorter signatures and keys than ML-DSA [NIS23] by sampling ephemeral vectors from a hyperball rather than a hypercube. This compactness reduces communication and persistent storage costs, making HAETAE attractive for constrained deployments. For its attractive qualities, it has been identified as a winner in the Korean domestic post-quantum cryptography competition, KpqC [Kor].

Motivation. Although HAETAE offers smaller public keys and signatures than ML-DSA at comparable security levels (Table 1), the reference implementation requires 71 kB to 141 kB of peak stack during the signing operation, depending on the security level, far exceeding the RAM budget of typical embedded targets. The root cause is that multiple large polynomial vectors used in sampling, challenge derivation, and hint computation are kept live simultaneously. The hyperball sampler amplifies the problem: because the scaling factor depends on the squared norm of the full Gaussian sample, a one-pass implementation must buffer the entire sample before producing any output.

This issue extends beyond signing. In modern embedded systems, key generation may also be performed at runtime for ephemeral credentials or key rotation, and verification must complete within tight stack budgets on sensor nodes that only verify signatures. A practical low-memory implementation should therefore address key generation, signing, and verification alike.

Related work. Bos, Renes, and Sprenkels [BRS22] showed that ML-DSA can fit within a few kilobytes of stack by trading recomputation for peak memory, streaming seed-derived objects, and reusing lifetime-disjoint buffers; similar strategies have been applied to FrodoKEM [BBC+23]. For HAETAE, however, only streaming matrix generation and sparse challenge multiplication carry over directly; the hyperball sampler, variable-length rANS encoding, and hint computation remain unaddressed by [BRS22]. To the best of our knowledge, [HCK+26] is the state-of-the-art memory-optimized HAETAE implementation on Cortex-M4. Following [BRS22], it achieves 5,212 B/8,092 B/6,220 B (key generation/signing/verification) for HAETAE-5, approaching the idealized baselines of Section 3. However, it targets HAETAE-5 only, leaving HAETAE-2/3 ($d>0$, different public-key domain and rejection structure) unaddressed, and its verification stack (6,220 B) plus the signature (2,948 B) totals 9,168 B, exceeding the 8 kB budget of many constrained devices. Fitting within this budget requires optimizations beyond general streaming, targeting HAETAE-specific components that prior work leaves unexplored.

1.1 Contributions

Support for all HAETAE security levels. We address both gaps identified above: our implementation supports all three security levels (HAETAE-2/3/5), with separate optimization paths for the $d>0$ (HAETAE-2/3) and $d=0$ (HAETAE-5) cases (Section 4.1), and fits verification within the 8 kB budget at every level. Our pure C implementation achieves 4.7 kB–5.7 kB key generation stack, 5.8 kB–6.0 kB signing stack, and 4.7 kB–4.8 kB verification stack across all levels, with `.data` = 0 and `.bss` = 0.

Novel stack optimization techniques. We introduce optimization techniques tailored to HAETAE's algorithmic structure that push the measured stack below the idealized streaming baselines (Section 3). For signing (Section 4.2), we introduce (i) *Rejection-aware pass decomposition*, which confines large buffers to non-overlapping lifetime phases; (ii) *Component-level early rejection*, which short-circuits the response computation when a partial norm exceeds the bound; and (iii) *Reverse-order streaming entropy coding*, which eliminates full hint and high-bits staging arrays. The signing path additionally employs a streaming two-pass hyperball sampler with a fully streaming Gaussian backend (`.bss` = 0), for which we provide a formal correctness proof (Section 4.4). For verification (Section 4.3), we combine row-streamed matrix multiplication with view-style decoding, replacing a large vector accumulator with a single polynomial and a compact union overlay. For HAETAE-5, these yield 4,816 B key generation (−7.6 % vs. [HCK+26]), 6,136 B signing (−24 %), and 4,840 B verification (−22 %), while key generation, signing, and verification are 25 %, 27 %, and 16 % faster respectively.

Evaluation and practical applicability. We evaluate on the `pqm4` framework and additionally validate portability under RIOT-OS on Cortex-M4 (nRF52840) and RISC-V (ESP32-C6) targets. On 8 kB devices, verification at all HAETAE security levels fits within budget (signature buffer + stack), and is 2.34–3.34× faster than ML-DSA m4fstack [BRS22] at each comparable security level. On 16 kB devices, all HAETAE levels support full signing (secret key + signature + stack), whereas ML-DSA-87 exceeds this budget.

Constant-time implementation. Our implementation avoids branches and memory accesses that depend on long-term secret coefficients, with the exception of the rejection-sampling loop inherent in the Fiat-Shamir-with-Aborts paradigm; verification processes only public inputs. Countermeasures against power analysis or fault attacks are not addressed in this work.

Our implementation will be available upon acceptance of the paper.

2 Background

In this section, we briefly explain the mathematical and algorithmic background of HAETAE, covering its algebraic setting, parameter sets, and core operations. Implementation-oriented algorithm specifications based on the latest HAETAE specification [Kpq26] are provided in Appendix A–C.

2.1 HAETAE

HAETAE is a module-lattice signature scheme operating in the polynomial ring

$$R_q \;=\; \mathbb{Z}_q[X]/(X^N+1),$$

with q a prime modulus. Vectors of ℓ (resp. k) polynomials are called `polyvecl` (resp. `polyveck`) and occupy $\ell \cdot N \cdot 4$ (resp. $k \cdot N \cdot 4$) bytes when stored with 32-bit coefficients. Polynomial multiplication in R_q is performed efficiently via the Number Theoretic Transform (NTT), which maps a polynomial to its evaluation at the $2N$-th roots of unity modulo q and reduces multiplication to a pointwise product in $\mathcal{O}(N \log N)$ time.

Parameter sets. HAETAE is specified at three security levels. Table 1 lists the parameters referenced in this paper. All three sets share the polynomial degree N=256 and modulus q=64,513. The module dimensions (k, ℓ) determine the sizes of the public matrix $\mathbf{A} \in R_q^{k\times\ell}$ and the secret/response vectors, and are the primary driver of memory usage. The

Table 1: Selected HAETAE parameters. $|\texttt{poly}| = N \times 4 = 1{,}024\,\text{B}$.

Parameter	Description	HAETAE-2	HAETAE-3	HAETAE-5
N	polynomial degree	256	256	256
q	modulus	64 513	64 513	64 513
(k, ℓ)	module dimensions	$(2, 4)$	$(3, 6)$	$(4, 7)$
τ	challenge weight	58	80	128
d	PK truncation	1	1	0
α	$\mathbf{z}_1$ compression	256	256	256
α_h	$\mathbf{h}$ compression	512	512	256
$\lvert pk\rvert$	public key (bytes)	992	1 472	2 080
$\lvert sk\rvert$	secret key (bytes)	1 408	2 112	2 752
$\lvert sig\rvert$	signature (bytes)	1 474	2 349	2 948

truncation parameter d governs the public-key format: d=1 (HAETAE-2/3) stores a rounded representation, while d=0 (HAETAE-5) stores the NTT-domain image directly (Section 2.1).

Like ML-DSA [NIS23], HAETAE is constructed in the *Fiat–Shamir with Aborts* (FSwA) framework [Lyu09]. In FSwA, the signer commits to an ephemeral randomness $\mathbf{y}$, receives (or derives) a binary challenge c of low Hamming weight, and computes a masked response $\mathbf{z} = \mathbf{y} + (-1)^b(c \star \mathbf{s})$. An abort (rejection) step is performed to ensure that the distribution of $\mathbf{z}$ does not leak information about the secret $\mathbf{s}$. The key distinction between ML-DSA and HAETAE lies in the distribution from which the ephemeral $\mathbf{y}$ is drawn, as detailed in Section 2.2.

HAETAE makes extensive use of HighBits and LowBits decompositions; we abbreviate these as HB and LB throughout, with superscripts (h, z_1, pk) indicating the decomposition type. For HAETAE-2/3 (d=1), the verification key stores only the high-order bits $\mathbf{b}_1 = \mathsf{HB}^{pk}(\mathbf{b})$ together with a public $\text{seed}_{\mathbf{A}}$ for expanding the matrix. The low-order bits $\mathbf{b}_0 = \mathsf{LB}^{pk}(\mathbf{b})$ are folded into the secret key. For HAETAE-5 (d=0), no rounding is applied; instead $\widehat{\mathbf{b}} = \mathsf{NTT}(-2\mathbf{b})$ is stored directly in the NTT domain.

The signature and key sizes of HAETAE are smaller than those of ML-DSA at comparable security levels, owing to the hyperball-based sampling strategy described below. At HAETAE-2 (targeting NIST security level 2), the signature size is 1 474 bytes, making it competitive with other lattice-based signatures while remaining efficient to verify.

Key Generation. Key generation expands a seed into the public matrix $\mathbf{A} \in R_q^{k \times \ell}$ and secret vectors $(\mathbf{s}_1, \mathbf{s}_2)$, then computes $\mathbf{b} = \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 \bmod q$. Unlike ML-DSA, HAETAE applies an explicit singular-value norm check $\mathcal{N}(\mathbf{s}_1, \mathbf{s}_2) \le \gamma^2 N$ that rejects and resamples until the bound is satisfied; this rejection loop and its FFT-based workspace affect the key-generation stack peak. The implementation differs across security levels: for HAETAE-2/3 (d>0), $\mathbf{b}$ is rounded and the public key stores $\mathbf{b}_1 = \mathsf{HB}^{pk}(\mathbf{b})$, while for HAETAE-5 ($d$=0), $\widehat{\mathbf{b}} = \mathsf{NTT}(-2\mathbf{b})$ is stored directly in the NTT domain. The norm rejection also differs: HAETAE-2/3 interleave it with the matrix–vector multiplication, while HAETAE-5 performs it before expanding $\mathbf{A}$. See Algorithm 3 and 4 in Appendix A for the detailed specifications.

Signing. Unlike ML-DSA, which samples $\mathbf{y}$ uniformly from a hypercube, HAETAE draws $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2)$ from a *discretized hyperball*: coefficients are sampled from a discrete Gaussian $\mathcal{D}_\sigma$, the vector is rescaled so that $\|(\mathbf{y}_1, \mathbf{y}_2)\|_2 \le B_0\Lambda$, and a norm rejection test is applied. This yields shorter signatures but requires a more expensive and memory-intensive sampler (Section 4.4).

In the signature operation, the signer derives $\mu = H(pk, M)$ and repeats the steps of sampling $\mathbf{y}$, computing $\mathbf{w} = \mathbf{A}\lfloor \mathbf{y}_1 \rceil \bmod 2q$, deriving challenge c, and forming $\mathbf{z} = \mathbf{y} +$

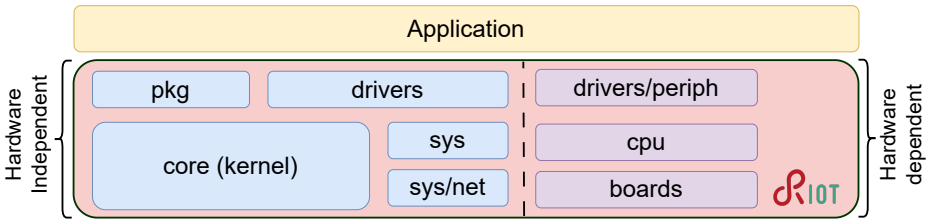

Figure 1: Overview of RIOT-OS modularization packages.

$(-1)^b c \star \mathbf{s}$. If $\mathbf{z}$ passes the norm bounds, the hint $\mathbf{h} = \mathsf{HB}^h(\mathbf{w}') - \mathsf{HB}^h(\mathbf{w}' - 2\lfloor \mathbf{z}_2 \rceil)$ is computed and the signature $\sigma = (\mathsf{HB}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathsf{LB}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathbf{h}, c)$ is encoded. See Algorithm 5 in Appendix B. A naïve implementation keeps $\mathbf{y}$, $\mathbf{w}$, $\mathbf{z}$, and encoding buffers live simultaneously, totaling $(\ell+2k) \cdot |\mathtt{poly}|$ of ring objects (e.g. over 12 kB for HAETAE-5). The hyperball sampler amplifies this: the scaling factor depends on the norm of the full Gaussian sample, so a one-pass implementation must store all $(L+K) \cdot N$ coefficients before producing any output.

Verification. Verification (Algorithm 6 in Appendix C) parses the signature, reconstructs $\tilde{\mathbf{w}} = \mathbf{A}\tilde{\mathbf{z}}_1 \bmod 2q$, recomputes the challenge from $\tilde{\mathbf{h}} + \mathsf{HB}^h(\tilde{\mathbf{w}}')$ and the message digest, and accepts if the recomputed challenge matches and all norm bounds hold. There is no rejection loop or secret-dependent sampler, but materializing the matrix product, decoded signature, and hint buffers simultaneously can still require substantial stack space.

2.2 Applicable ML-DSA Memory Optimization

Due to the similarity of the schemes, the memory-optimization techniques of [BRS22] can be applied to the operations common between HAETAE and ML-DSA.

Both schemes are module-lattice Fiat–Shamir signatures with the same high-level shape: seed expansion to generate a public matrix $\mathbf{A}$; computation of a public commitment involving $\mathbf{A}$ and short secret vectors; a sign-then-reject loop in which an ephemeral sample is masked by a low-weight challenge; and verification by recomputing the challenge from the response and the public key. Both also rely on NTT-based polynomial arithmetic and HB/LB decompositions to compress intermediate values. This structural similarity means that the *streaming* and *liveness-reduction* ideas from [BRS22] carry over to HAETAE with appropriate adaptation.

The primary algorithmic difference is the distribution from which the ephemeral randomness $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2)$ is drawn. ML-DSA samples $\mathbf{y}$ *uniformly from a hypercube*: each coefficient is drawn independently and uniformly from $\{-\gamma_1+1, \ldots, \gamma_1\}$. HAETAE instead samples $\mathbf{y}$ from a *discrete Gaussian conditioned on a hyperball*: coefficients are drawn from a discrete Gaussian, the vector is rescaled so that its Euclidean norm lies in $[\,0, B_0\Lambda\,]$, and a Euclidean norm rejection test is applied. This hyperball sampling yields shorter signatures and keys because it more efficiently fills the acceptance region, but at the price of a more expensive and memory-intensive sampler (see Section 4.4).

In ML-DSA, the public key is $(\rho, \mathbf{t}_1)$, where $\mathbf{t}_1$ is the high-order part of $\mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$. The matrix seed ρ is stored explicitly, and the secret key additionally holds $\mathbf{t}_0$ (the low-order residue) to enable efficient signing. In HAETAE, the public key is $(\mathrm{seed}_{\mathbf{A}}, \mathbf{b}_1)$, where $\mathbf{b}_1$ is the high-order part of a related lattice commitment $\mathbf{b}$. The HAETAE secret key is more compact; it stores a single short vector $\mathbf{s}$ of augmented dimension $k + \ell + 1$ and a nonce K—because the lattice relation allows the signer to recover the full signing basis on-the-fly.

Algorithm 1 Schematic HAETAE (Key generation, Signing, Verification)

procedure KEY GENERATION(1^λ)
  Sample seeds ρ, κ; sample $(\mathbf{s}_1, \mathbf{s}_2)$
  Reject if $\mathcal{N}(\mathbf{s}_1, \mathbf{s}_2) > \gamma^2 N$ ▷ FFT-based singular-value norm
  Expand $\rho \mapsto \mathbf{A}$; compute $\mathbf{b} \leftarrow \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 \bmod q$; output (pk, sk)
end procedure
procedure SIGNING(sk, M)
  Derive $\mu \leftarrow H(pk, M)$
  repeat
    Sample $(\mathbf{y}_1, \mathbf{y}_2)$; compute $\mathbf{w} \leftarrow \mathbf{A}\lfloor \mathbf{y}_1 \rceil \bmod 2q$
    $c \leftarrow \mathsf{SampleBinaryChallenge}_\tau(H(\mathsf{HB}^h(\mathbf{w}'), \mathsf{LSB}(\lfloor y_{1,1} \rceil), \mu))$
    $\mathbf{z} \leftarrow \mathbf{y} + (-1)^b c \star \mathbf{s}$
  until $\mathbf{z}$ passes rejection tests
  $\mathbf{h} \leftarrow \mathsf{HB}^h(\mathbf{w}') - \mathsf{HB}^h(\mathbf{w}' - 2\lfloor \mathbf{z}_2 \rceil)$ ▷ post-acceptance
  $\sigma \leftarrow (\mathsf{HB}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathsf{LB}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathbf{h}, c)$
end procedure
procedure VERIFICATION(pk, M, σ)
  Parse σ into $(\mathsf{HB}^{z_1}, \mathsf{LB}^{z_1}, \mathbf{h}, c)$; reconstruct $\tilde{\mathbf{z}}_1$
  $\tilde{\mathbf{w}} \leftarrow \mathbf{A}\tilde{\mathbf{z}}_1 \bmod 2q$; recompute c' from $(\tilde{\mathbf{h}} + \mathsf{HB}^h(\tilde{\mathbf{w}}'), \mu)$
  Accept iff $c'{=}c$ and norm bounds hold
end procedure

2.3 RIOT-OS

RIOT [RIO] is an open-source operating system targeting resource-constrained IoT devices. Originally introduced in [BHG+13], it is written in C and designed with portability in mind, which allows it to support a broad range of boards and architectures. Figure 1 shows the modular structure of RIOT. The most left part in the figure corresponds to hardware-independent components, whereas the right part contains hardware-specific ones. This design enables RIOT to reuse most of its code across platforms and to favor portable C implementations instead of relying on architecture-specific intrinsics or assembly code.

3 Idealized memory baselines for HAETAE

Before describing the concrete implementation techniques in Section 4, we analyze the idealized memory footprint of each HAETAE routine by counting only the dominant ring objects (polynomials and polynomial vectors) while ignoring constant-size overheads such as hash states, scalar variables, and call frames. This analysis clarifies the relationship between the number of simultaneously live large ring objects and the achievable peak stack.

Dominant objects. All three HAETAE parameter sets use $N{=}256$; a single polynomial with 32-bit coefficients occupies $|\texttt{poly}| = 1\,024$ B. The public matrix $\mathbf{A} \in R_q^{k \times \ell}$ has k rows and ℓ columns; a `polyveck` (k polynomials) and `polyvecl` (ℓ polynomials) scale linearly. For HAETAE-5 ($k{=}4, \ell{=}7$) these are 4 096 B and 7 168 B respectively.

High-level algorithms. We summarize the HAETAE workflow in Algorithm 1; it is intentionally schematic since our implementation keeps the mathematics unchanged while reorganizing the order in which objects are materialized. We use the HB/LB abbreviations introduced in Section 2.1.

Key generation. The reference implementation materializes $\mathbf{A}$, $\mathbf{s}_1$, and $\mathbf{b}$ simultaneously ($\approx (k\ell{+}\ell{+}k) \cdot |\texttt{poly}|$). Since both $\mathbf{A}$ and $\mathbf{s}_1$ are deterministically expandable from seeds, the matrix–vector product can be streamed with $2{\cdot}|\texttt{poly}|$ (one row accumulator plus one sampling scratch). However, the singular-value norm check $\mathcal{N}(\mathbf{s}_1, \mathbf{s}_2) \le \gamma^2 N$ requires an

Table 2: Streaming baselines vs. measured stack on `pqm4` (HAETAE-5, bytes). Baselines count dominant ring objects and staging buffers; constant-size overheads (hash states, scalars, call frames) are excluded. Full performance results are in Table 3.

	Idealized baseline	[HCK+26]	Ours
Key generation	$2 \times 1\,024 + 2\,048 = 4\,096$	5 212	**4 816**
Signing	$(k{+}3) \times 1\,024 + \ell{\cdot}N = 8\,960$	8 092	**6 136**
Verification	$(k{+}2) \times 1\,024 = 6\,144$	6 220	**4 840**

FFT-based computation with a 2 048 B workspace, giving a total baseline of $2{\cdot}|\texttt{poly}| + 2\,048 = 4\,096$ B.

Signature generation. The reference implementation keeps $\mathbf{y}_1$ (ℓ polys), $\mathbf{y}_2$ and $\mathbf{w}$ ($2k$ polys), and the hint $\mathbf{h}$ live simultaneously, totaling $\approx(\ell{+}2k) \cdot |\texttt{poly}|$ (e.g. 12 288 B for HAETAE-5). With standard streaming, $\mathbf{y}$ and $\mathbf{w}$ can be regenerated row by row from seeds, but the full hint buffer $\mathbf{h} \in R^k$, the high-bits array $\mathsf{HB}^{z_1}(\mathbf{z}_1) \in \mathbb{Z}^{\ell \times N}$, and the challenge polynomial c must still be materialized before the rANS encoder can process them. In particular, c remains live throughout the rejection test and the packing phase, coexisting with the hint and streaming buffers. This gives a streaming baseline of $(k{+}3) \cdot |\texttt{poly}| + \ell{\cdot}N$ bytes (e.g. $7\,168 + 1\,792 = 8\,960$ B for HAETAE-5).

Verification. A column-streamed approach accumulates the matrix product $\tilde{\mathbf{w}} = \widehat{\mathbf{A}} \circ \mathsf{NTT}(\tilde{\mathbf{z}}_1)$ into a full `polyveck` buffer while sweeping $\tilde{\mathbf{z}}_1$ once. The challenge polynomial c must also remain live for the final comparison, giving a baseline of $(k{+}2) \cdot |\texttt{poly}|$ (e.g. 6 144 B for HAETAE-5).

Summary and outlook. Table 2 summarizes the baseline streaming footprints for HAETAE-5; these baselines count dominant ring objects and staging buffers while excluding seeds, hash states, and other implementation-specific overheads. The same analysis applies to HAETAE-2/3 with adjusted (k, ℓ) values, though differences in the public-key domain ($d{>}0$ requires an additional rounding step and $\mathbf{a}$-vector) and the placement of the rejection loop affect the concrete figures. The prior work of [HCK+26] reports HAETAE-5 stack usage close to these baselines: 5,212 B (Key generation), 8,092 B (Signing), and 6,220 B (Verification). In the next section we introduce additional techniques, including pass decomposition with `noinline` boundaries, reverse-order streaming entropy coding, and row-streamed verification, that push the stack footprint below these idealized baselines. Section 5.2 confirms that the resulting implementation achieves lower stack usage while retaining competitive cycle counts relative to [HCK+26].

4 Low-footprint memory techniques

In this section, we describe the main techniques we use to reduce the memory footprint of HAETAE key generation, signing, and verification. Our approach is guided by the principle of trading increased computation time for reduced peak stack footprint, which is essential for predictable execution on memory-constrained microcontrollers. We also leverage the structure of HAETAE's algorithms, such as the use of seed-derived objects and sparse challenges, to enable streaming and in-place computation strategies that minimize stack usage.

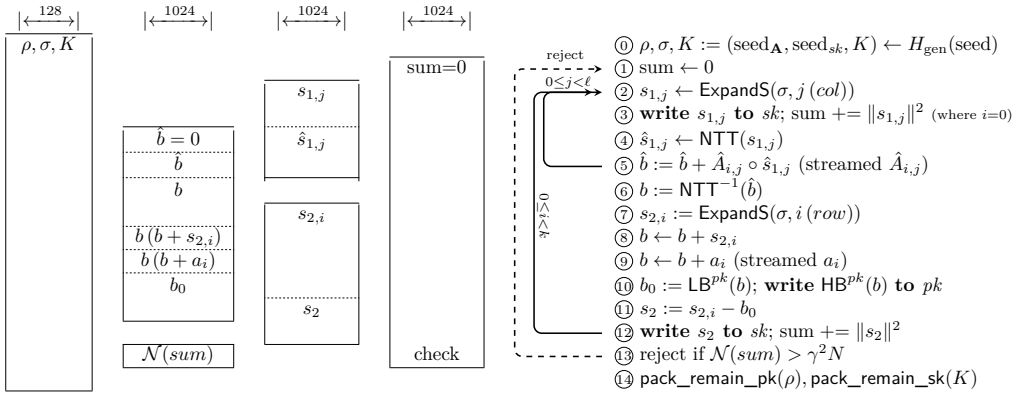

Figure 2: Memory allocation of the proposed memory-optimized HAETAE-2,3 key generation with streaming. This approach streams the matrix $\hat{A}$ on-the-fly and reuses memory slots, keeping only two single polynomials (`poly b, s`) and one norm accumulator (`int32_t sum[N]`) on the stack. The dashed arrow indicates the rejection loop unique to HAETAE-2,3.

4.1 Low-stack key generation via streaming

When key generation for digital signatures on constrained devices is required to be performed at runtime, it can no longer be circumvented with a one-time provisioning step. Practical applications include generating ephemeral keys for firmware integrity verification and implementing key rotation to mitigate compromise.

The baseline key generation is specified in Algorithms 3 and 4 (Appendix A). We describe our stack optimizations in two parts. First we present the streaming techniques applied to HAETAE-2/3 ($d > 0$), which involve rounding and an additional $\mathbf{a}$-vector not present in the d=0 case; this streaming of key generation for $d > 0$ is a new contribution not covered in previous work. Figure 2 illustrates the resulting memory layout. We then describe our HAETAE-5 ($d = 0$) implementation, which follows the same high-level approach as [HCK+26], but further compresses the stack through caller-level union analysis.

Streaming matrix–vector multiplication. Key generation computes $\mathbf{b} = \mathbf{a} + \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2 \pmod q$, where $\mathbf{A} \in R_q^{k\times\ell}$ is derived from a seed and $\mathbf{s}_1 \in R_q^{\ell}$, $\mathbf{s}_2 \in R_q^{k}$ are short secret vectors. The reference implementation materializes $\mathbf{A}$ and $\widehat{\mathbf{s}}_1$ as full polynomial vectors, causing large stack pressure. We reorganize the computation so that $\mathbf{s}_1$, $\mathbf{s}_2$, and $\mathbf{A}$ are all generated and consumed on-the-fly (steps ②–⑫ in Figure 2): for each column j we sample $\mathbf{s}_{1,j}$ into a single scratch polynomial (②), apply the NTT in place (④), and for each row i generate $\mathbf{A}_{i,j}$ from the seed and immediately accumulate the pointwise product into $\mathbf{b}_i$ (⑤). The secret vectors are written to the secret key as they are produced (③, ⑫), and a norm accumulator (`int32_t sum[N]`) tracks the singular-value check incrementally (⑬); $\mathbf{s}_1$ norms are accumulated only at i=0 to avoid redundant computation in the inner loop. The entire key generation thus requires only two single-polynomial scratch slots (`poly b, s`) and one norm accumulator on the stack.

Additional stack reductions. SHAKE-based sampling routines (`poly_uniform`, and `poly_uniform_eta`) are rewritten to squeeze one block at a time, replacing multi-block stack buffers with a single-block buffer without altering the output distribution. Together with the streaming matrix multiplication and the fused frozen-A variant, these techniques remove full-vector temporaries, stack-local expansion buffers, and multi-block squeeze buffers, yielding a substantially smaller and more predictable stack profile. Note that

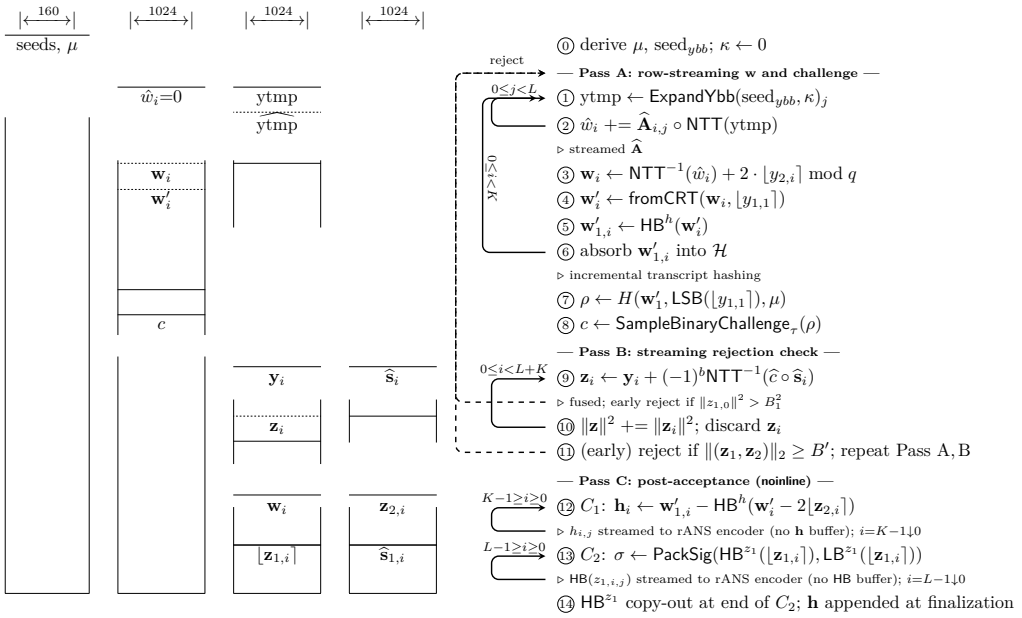

Figure 3: Memory allocation of the proposed pass-decomposed HAETAE signing. The driver holds persistent state (seeds, μ; 160 B) and a single 1 024 B polynomial slot that serves as the row accumulator $\hat{w}_i$ for the matrix–vector product in Pass A and holds the challenge c from Pass B onward. Two additional poly-sized scratch slots (1 024 B each) are reused across passes. The dashed arrow marks the rejection loop (Pass A + B only); Pass C runs once after acceptance. C_1 and C_2 are invoked via `noinline`, so their frames never coexist. Reverse-order loops ($i{\downarrow}, j{\downarrow}$) in C_1/C_2 stream coefficients directly to the rANS encoder, eliminating full $\mathbf{h}$ and $\mathsf{HB}(\mathbf{z}_1)$ staging buffers.

streaming and re-computation alter data-access patterns; a thorough side-channel analysis is left for future work.

HAETAE-5 ($d = 0$) key generation. For HAETAE-5, the key generation follows the same high-level streaming approach as prior work [HCK+26]: on-the-fly matrix generation, row-by-row accumulation of $\widehat{\mathbf{b}} = \mathsf{NTT}(-2\mathbf{b})$, and direct storage in the NTT domain. The two-pass structure is illustrated in Figure 6 (Appendix A): Pass 1 (norm check, steps ①–⑥ in Figure 6) and Pass 2 (matrix–vector product, steps ⑦–⑰). Our implementation additionally introduces two caller-level `union`s that tightly pack all large temporaries into a fixed-size caller frame, eliminating the need for deep callee stack frames. The first union shares the 2 048-byte FFT scratch used for the singular-value norm in Pass 1 (③, ⑤) with the 1 024-byte sampling polynomial (`poly s`) used in both passes (②, ⑦); the FFT is invoked directly in the caller rather than delegated to a callee, so the scratch never appears as a separate stack frame. The second union shares the norm accumulator (`int32_t sum[N]`, 1,024 B) in Pass 1 (①) with the row accumulator (`poly b`, 1,024 B) in Pass 2 (⑩), exploiting the strict lifetime disjointness across the two passes. Together, the two unions compress the dominant temporaries to $2\,048 + 1\,024 = 3\,072$ bytes in the caller frame, with no additional callee overhead for the norm computation. By analyzing the stack frames of each callee and identifying lifetime-disjoint buffers across the two passes, this union-based layout reduces the key-generation stack by approximately 400 B compared to [HCK+26] (4,784 B vs. 5,212 B on the same `pqm4`framework).

4.2 Low-stack signing via pass decomposition and reverse-order streaming

Recall HAETAE signing (Algorithm 5 in Appendix B) mainly consists of a rejection-sampling loop: each iteration draws ephemeral vectors $(\mathbf{y}_1, \mathbf{y}_2)$ from a hyperball distribution, computes the matrix–vector product $\mathbf{w} \leftarrow \mathsf{NTT}^{-1}(\widehat{\mathbf{A}} \circ \mathsf{NTT}(\lfloor \mathbf{y}_1 \rceil)) + 2\lfloor \mathbf{y}_2 \rceil \bmod q$, derives a challenge c, forms the response $\mathbf{z} = \mathbf{y} + (-1)^b c \star \mathbf{s}$, and repeats until the norm bounds are satisfied. The reference implementation [CCD+23] keeps $\mathbf{w}$, $\mathbf{z}$, and the hint $\mathbf{h}$ live simultaneously, leading to high peak stack usage.

Our design follows the [BRS22] principle of trading recomputation for reduced peak memory, and adopts the seed-based on-the-fly matrix streaming used by [HCK+26]: each entry of $\widehat{\mathbf{A}}$ is regenerated from the public seed rather than stored in RAM. Beyond these shared foundations, we introduce several techniques not described by [HCK+26]: (i) a *Rejection-aware pass decomposition* that confines encoding and hint computation to a post-acceptance path; (ii) *Component-level early rejection* that short-circuits the response computation in Pass B when a partial norm already exceeds the bound; and (iii) *Reverse-order streaming entropy coding* that eliminates full hint and high-bit staging buffers by aligning computation order with the rANS encoder's backward symbol consumption. Additionally, our signing path achieves zero mutable BSS (`.bss` = 0 B) through a streaming Gaussian backend (Section 4.4). Moreover, while [HCK+26] targets HAETAE-5 only, our design supports all three security levels including HAETAE-2/3 (d>0). Figure 3 illustrates the resulting memory layout.

Rejection-aware pass decomposition. We decompose the signing loop into three passes whose large buffers have strictly non-overlapping lifetimes:

- **Pass A** (steps ①–⑧ in Figure 3) samples $(\mathbf{y}_1, \mathbf{y}_2)$ via the two-pass hyperball sampler (Section 4.4), computes $\mathbf{w}$ row by row using on-the-fly matrix streaming, derives $\mathbf{w}'_1 = \mathsf{HB}^h(\mathsf{fromCRT}(\mathbf{w}, \lfloor y_{1,1} \rceil))$, and obtains the challenge $c \leftarrow \mathsf{SampleBinaryChallenge}_\tau\big(H(\mathbf{w}'_1, \mathsf{LSB}(\lfloor y_{1,1} \rceil), \mu)\big)$. The single-polynomial row accumulator for $\mathbf{w}$ reuses the output location of c (overwritten only at the end of the pass), eliminating a dedicated scratch buffer. For d>0 (HAETAE-2/3), the first column of $\widehat{\mathbf{A}}$ is derived on-the-fly from the packed public key via in-place CRT reconstruction, requiring no additional polynomial buffer.
- **Pass B** (⑨–⑪) regenerates $(\mathbf{y}_1, \mathbf{y}_2)$ from the stored seed and accepted nonce, computes $\mathbf{z} = \mathbf{y} + (-1)^b c \star \mathbf{s}$ one component at a time via fused accumulation (described below), and evaluates the rejection tests. No matrix product is needed, since the ℓ_2-norm and ℓ_∞ tests involve only $\mathbf{z}$, $\mathbf{y}$, and b'. Each component is discarded after its norm contribution is accumulated.
- **Pass C** (⑫–⑭) runs only once, after acceptance. It recomputes $\mathbf{z}$ and $\mathbf{w}$ to derive the hint $\mathbf{h} \leftarrow \mathbf{w}'_1 - \mathsf{HB}^h(\mathbf{w}' - 2\lfloor \mathbf{z}_2 \rceil) \bmod {}^+ \frac{2(q-1)}{\alpha_h}$ and packs the signature. The variable-length payloads $\mathbf{h}$ and $\mathsf{HB}^{z_1}(\mathbf{z}_1)$ are encoded via reverse-order streaming rANS (described below). Pass C is further split into two sub-phases (C_1, hint encoding, and C_2, $\mathbf{z}_1$ packing) with non-overlapping buffer lifetimes.

Because the rejection loop comprises only Pass A and Pass B, all encoding and hint computation are deferred to the post-acceptance path. The peak signing stack is therefore

$$S_{\text{sign}} \;=\; S_{\text{driver}} + \max\big(S_A,\; S_B,\; S_{C_1},\; S_{C_2}\big), \tag{1}$$

where S_{driver} is the persistent driver state (seeds, nonce counter, hash prefix) and the four terms are the transient allocations of each pass. Our decomposition guarantees that the peak follows Equation (1) through explicit `noinline` boundaries between passes, confining encoding and hint computation to the single post-acceptance execution.

In contrast, the idealized streaming baseline from Section 3 requires the full hint and high-bits arrays to be materialized before encoding:

$$S_{\text{sign}}^{\text{conv}} \;=\; (k{+}2)\cdot|\texttt{poly}| + \ell\cdot N.$$

Our pass-aware design replaces this with Equation (1), where for HAETAE-5:

$$S_{C_1} \approx 3\cdot|\texttt{poly}| + B_h, \qquad S_{C_2} \approx 2\cdot|\texttt{poly}| + B_{hb},$$

with B_h and B_{hb} denoting the compact streaming encoder buffers (bounded by the base entropy of $\mathbf{h}$ and $\mathsf{HB}^{z_1}(\mathbf{z}_1)$ respectively), which are substantially smaller than the full staging arrays they replace. This means we are even below the idealized streaming baseline. Since S_{C_1} and S_{C_2} dominate the peak while Pass A requires only a single polynomial accumulator, the pass-aware structure leaves unused stack budget in Pass A. We exploit this by batching two matrix rows per iteration of the inner product, halving the number of ephemeral-vector regenerations from $k\ell$ to $\lceil k/2\rceil\cdot\ell$ at the cost of one additional polynomial accumulator ($+1\,024$ B in S_A), without increasing the overall peak S_{sign}.

Component-level early rejection. The reference implementation computes the full response vector $\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2)$ before evaluating any rejection condition. We introduce component-level early exits that reduce the average cost of rejected iterations in Pass B (steps ⑨–⑩ in Figure 3). The first component $z_{1,0} = y_{1,0} + (-1)^b \cdot c$ involves no secret-key material (the challenge c is derived from public data), so a data-dependent branch on its norm is constant-time safe. Pass B therefore computes $z_{1,0}$ before any NTT-based multiplication (⑨) and rejects immediately if $\|z_{1,0}\|^2 > B_1^2$ (dashed arrow between ⑨ and ⑩), skipping the remaining $\ell{+}k{-}1$ fused multiply-accumulate operations. During the subsequent per-component norm accumulation (⑩) $\|\mathbf{z}\|^2 = \sum_i \|z_i\|^2$, we check the running partial sum after each polynomial and skip the remaining components once the bound is exceeded. Additionally, the hyperball scaling factors, which are deterministic functions of the seed and nonce counter, are computed once in the driver frame and passed by pointer to Passes B, C_1, and C_2, eliminating three redundant recomputations per accepted iteration.

Reverse-order streaming entropy coding. The signature includes two variable-length rANS-encoded payloads: the hint $\mathbf{h} \in \mathbb{Z}^{k\times N}$ and the high-bit decomposition $\mathsf{HB}(\mathbf{z}_1) \in \mathbb{Z}^{\ell\times N}$. Conventional implementations materialize these as full arrays before encoding; for HAETAE-5, a `polyveck` hint buffer ($k\times 1\,024$ B $= 4\,096$ B) and an `int8_t` array ($\ell\times N = 1\,792$ B) for $\mathsf{HB}(\mathbf{z}_1)$.

rANS is a *backward* entropy coder: for a flat array $X[0],\ldots,X[m{-}1]$, the encoder consumes symbols in the order $X[m{-}1], X[m{-}2],\ldots,X[0]$. The conventional approach first materializes the entire array, then iterates backward:

$$\mathsf{RansEncPut}(X[m{-}1]),\ \mathsf{RansEncPut}(X[m{-}2]),\ \ldots,\ \mathsf{RansEncPut}(X[0]).$$

We show that for a row-major $R\times C$ array ($m = R\cdot C$), the identical reverse sequence can be produced without materializing the full array, by traversing rows from $i = R{-}1$ down to 0 and coefficients from $j = C{-}1$ down to 0 within each row:

$$\begin{aligned}&\text{for } i = R{-}1 \text{ downto } 0:\\ &\quad\text{for } j = C{-}1 \text{ downto } 0:\\ &\qquad\mathsf{RansEncPut}\big(X[i\cdot C + j]\big).\end{aligned}$$

This allows each coefficient to be fed to the encoder immediately upon computation, without materializing the full $R\times C$ array. In Pass C_1 (⑫ in Figure 3), each hint row $\mathbf{h}_i$

($R{=}k$, $C{=}N$) is recomputed via row-streaming and immediately encoded; in Pass C_2 (⑬), $\mathsf{HB}^{z_1}(\mathbf{z}_{1,i})$ ($R{=}\ell$, $C{=}N$) is handled analogously. Both full staging buffers are eliminated entirely.

The encoded output is accumulated in a compact stack-local encoder context and copied to the signature buffer only after encoding completes, following an *internal encoding with copy-out* boundary model. To verify correctness, we performed differential testing for each parameter set (HAETAE-2/3/5, 10 000 signatures each): for each randomly generated signature, we extracted $\mathbf{h}$ and $\mathsf{HB}^{z_1}(\mathbf{z}_1)$ from the produced signature, re-encoded them with both the conventional full-array encoder (`encode_h` / `encode_hb_z1`) and the streaming reverse-order encoder, and compared the resulting byte streams. In all cases the two outputs were byte-identical.

Incremental transcript hashing. Both signing (Pass A, steps ⑤–⑥ in Figure 3) and verification (⑧–⑨ in Figure 4) derive the challenge polynomial by hashing a transcript that includes the packed high bits of the matrix–vector product. The reference implementation allocates a contiguous buffer of `POLYVECK_HIGHBITS_PACKEDBYTES + POLYC_PACKEDBYTES` bytes (608–1 184 B depending on the security level), packs all k rows into it, and then feeds it to SHAKE256 in one shot. Because SHAKE256 is a sponge and satisfies $\mathrm{absorb}(A\|B) = \mathrm{absorb}(A);\,\mathrm{absorb}(B)$, we replace this monolithic buffer with *row-by-row incremental absorb*: each row's high bits are packed into a small single-row buffer (`POLY_HIGHBITS_PACKEDBYTES`), immediately absorbed into the SHAKE256 state, and discarded before the next row is processed. After the row loop, the remaining fields ($\mathbf{w}'$ and the message digest μ) are absorbed and the state is finalized. This eliminates the full transcript buffer entirely, yielding measurable stack savings (up to 656 B for HAETAE-5).

Sparse challenge multiplication with fused accumulation. In Pass B (⑨ in Figure 3), for HAETAE-2/3, where the challenge $c(X) = \sum_{t=1}^{\tau} X^{i_t}$ has Hamming weight $\tau{=}60$, we replace NTT-based multiplication by a purely coefficient-domain *signed shift-and-add* rule in the negacyclic ring $R_q = \mathbb{Z}_q[X]/(X^N{+}1)$:

$$(c \star s)[j] \;=\; \sum_{t=1}^{\tau} \varepsilon_t \cdot s[(j - i_t) \bmod N], \qquad \varepsilon_t = \begin{cases} +1 & j \geq i_t, \\ -1 & j < i_t, \end{cases}$$

where ε_t arises from the negacyclic relation $X^N = -1$. For HAETAE-5 ($\tau{=}128$, dense binary challenge), we retain NTT-based multiplication as the dense weight makes a coefficient-domain approach less efficient. In all cases we fuse the accumulation directly into the response: each signed shift (or partial product) is added in place to the corresponding component of $\mathbf{y}$, producing $\mathbf{z}$ without allocating a separate product polynomial (saving 1,024 B per multiplication).

Algorithm-level BSS elimination. Our complete signing path uses no mutable BSS (`.bss` = 0 B). The hyperball sampler's static temporary buffers (`tmp_samples[N+1]` and `tmp_signs[(N+7)/8]`, totaling 2,088 B in the reference implementation) are replaced by a fully streaming Gaussian backend that generates samples on-the-fly from the SHAKE sponge state, as detailed in Section 4.4. Our implementation also provides an alternative build configuration (`BRS_BSS_WORKSPACE`) that places large signing vectors ($\mathbf{y}$, $\mathbf{z}$, $\mathbf{Ay}$) in a static BSS workspace for reduced recomputation, trading BSS for lower-latency signing. In the full-streaming configuration (Table 3), the pass decomposition eliminates this workspace entirely, achieving `.data` = 0 and `.bss` = 0.

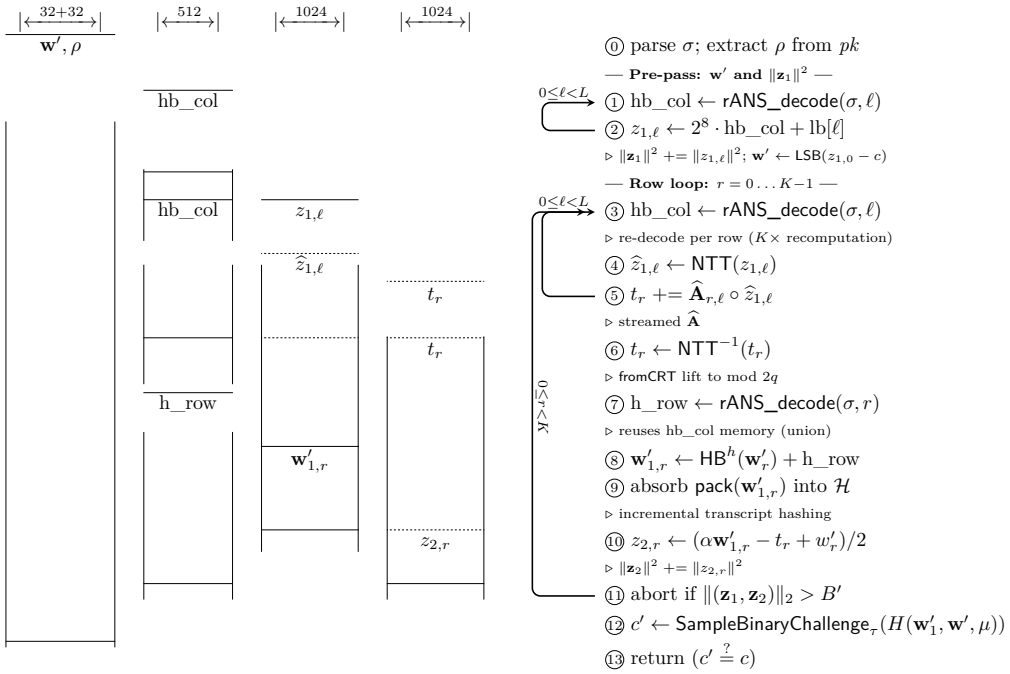

Figure 4: Memory allocation of the row-streamed HAETAE verification. Four memory slots are used: persistent small buffers ($\mathbf{w}'$ packed bits + seed ρ, 64 B, retained through the final challenge recomputation), a `union{int8_t hb_col[N]; uint16_t h_row[N]}` (512 B) that alternates between decoded high-bits columns in the inner ℓ loop and the decoded hint row afterwards, and two polynomials `poly z` and `poly trow` (1 024 B each). The row loop re-decodes $\mathsf{HB}^{z_1}(\mathbf{z}_1)$ via streaming rANS for each row r, trading a factor-k re-computation for the elimination of a full `polyveck` buffer. The incremental SHAKE256 state $\mathcal{H}$ absorbs each row's packed high bits immediately.

4.3 Low-stack verification via view-style decoding and row streaming

HAETAE verification (Algorithm 6 in Appendix C) reconstructs $\tilde{\mathbf{w}}'_1 = \tilde{\mathbf{h}} + \mathsf{HB}^h(\tilde{\mathbf{w}}')$ from $\tilde{\mathbf{w}} = \widehat{\mathbf{A}} \circ \mathsf{NTT}(\tilde{\mathbf{z}}_1)$ (up to the CRT lift mod $2q$), derives the auxiliary component $\tilde{\mathbf{z}}_2$, and re-hashes the packed transcript $(\tilde{\mathbf{w}}'_1, w')$ to recompute the challenge. Rather than following the column-streamed approach of [HCK+26], our verification is built on the [BRS22] method of trading re-computation for transient memory, combining (i) *view-style decoding* with union overlays, (ii) *row-streamed* matrix–vector multiplication, and (iii) *incremental transcript hashing* that eliminates the full-transcript buffer (the same technique introduced for signing in Section 4.2). This design supports all three security levels, including HAETAE-2/3 (D>0), which is not addressed by [HCK+26], and achieves a uniform verification stack of 3,800 B across all security levels, a 39 % reduction compared to the 6,220 B reported by [HCK+26] for HAETAE-5 alone. Figure 4 illustrates the resulting memory layout.

View-style decoding. We avoid materializing the decoded signature vectors (steps ⓪–② in Figure 4). Concretely, we keep $\mathsf{LB}^{z_1}(\lfloor \mathbf{z}_1 \rceil)$ as a pointer into the signature buffer, decode only $\mathsf{HB}^{z_1}(\lfloor \mathbf{z}_1 \rceil)$ into an `int8` array (①), and decode $\mathbf{h}$ into a compact `uint16` array (⑦). These two decoded arrays share a `union` overlay (`int8_t hb_col[N]` / `uint16_t h_row[N]`), exploiting lifetime disjointness: the high-bits columns are consumed during the inner column loop, after which the same memory is reused for the decoded hint row. We also represent $w' \in \mathcal{R}_2$ as a packed bitstring of $N/8$ bytes. A pre-pass (①–②) over

coefficients recomposes $\tilde{z}_{1,\ell} = \mathsf{HB}^{z_1} \cdot 256 + \mathsf{LB}^{z_1}$ on-the-fly, accumulates $\|\tilde{\mathbf{z}}_1\|_2^2$, and derives $w' = \mathsf{LSB}(\tilde{z}_{1,1} - c)$. Together, the working set for decoded data is a single 512-byte union plus a 32-byte packed bitstring, independent of the module dimensions k and ℓ. Signatures that fail format checks (length, zero-padding, or rANS decoder errors) are rejected before any transcript recomputation.

Row-streamed multiplication. To minimize peak stack, we compute $\tilde{\mathbf{w}}$ one row at a time rather than column-by-column (steps ③–⑪ in Figure 4). For each output row r, we recompute an NTT-domain accumulator $\tilde{w}_r \leftarrow \sum_{\ell=0}^{L-1} \widehat{\mathbf{A}}[r,\ell] \circ \mathsf{NTT}(\tilde{z}_{1,\ell})$ (③–⑤) by (re)composing and transforming one column $\tilde{z}_{1,\ell}$ at a time and streaming $\widehat{\mathbf{A}}[r,\ell]$ from the public seed ρ. After an inverse NTT and CRT lift via $\mathsf{fromCRT}$ (⑥), we decode the hint row $\tilde{\mathbf{h}}_r$ (⑦) and form $\tilde{w}'_{1,r} = \tilde{\mathbf{h}}_r + \mathsf{HB}^h(\tilde{w}'_r)$ (⑧), then immediately absorb the packed high bits into the incremental SHAKE256 transcript state (⑨). Within the same loop we compute $\tilde{z}_{2,r}$ (⑩), accumulate $\|\tilde{z}_{2,r}\|_2^2$, and abort early as soon as the bound test fails (⑪). This reduces the working set to a constant number of polynomials plus small decoded arrays, eliminating concurrent `polyveck` temporaries. The tradeoff is a factor-k increase in re-computation: each row re-decodes the ℓ columns of $\mathsf{HB}^{z_1}(\lfloor \mathbf{z}_1 \rceil)$ via streaming rANS and re-applies ℓ forward NTTs.

In contrast, [HCK+26] uses a *column-streamed* approach that accumulates $\tilde{\mathbf{w}}$ into a full `polyveck` buffer ($k \times 1\,024$ B) while sweeping the columns of $\tilde{\mathbf{z}}_1$ once. Our row-streamed design trades this k-fold re-decode cost for a $(k-1) \times 1\,024$ B reduction in peak live memory. Combined with the incremental transcript hashing described in Section 4.2, which absorbs each row's high bits directly into the SHAKE256 state, the full `POLYVECK_HIGHBITS_PACKEDBYTES` transcript buffer (up to 1 184 B) is eliminated entirely, an optimization not applied by either the reference implementation or [HCK+26].

For $D>0$ (HAETAE-2/3), the first column of $\widehat{\mathbf{A}}$ is derived on-the-fly from the public seed and the packed public key, adding one polynomial expansion per row but no additional persistent storage.

In the second pass, the sampler restarts the pseudorandom stream from the same $(\mathsf{seed}, \mathsf{nonce})$ pair, so that $\mathtt{SampleGauss}(\mathsf{st})$ reproduces the identical sequence $(x_{i,j})$. This time, each sample is immediately rescaled and rounded,

$$y_{1,i,j} = \mathtt{Round}(\alpha x_{i,j}) \quad (0 \le i < \ell), \qquad y_{2,i,j} = \mathtt{Round}(\alpha x_{\ell+i,j}) \quad (0 \le i < k),$$

and written to the output polynomials $\mathbf{y}_1$ and $\mathbf{y}_2$. The algorithm then evaluates the squared norm $\left\|(\mathbf{y}_1, \mathbf{y}_2)\right\|_2^2$ and rejects if it exceeds $B_0^2\Lambda^2$, repeating the entire two-pass procedure until acceptance.

4.4 A memory friendly sampler

The hyperball sampler is invoked in Pass A of signing (① in Figure 3) to draw the ephemeral vectors $(\mathbf{y}_1, \mathbf{y}_2)$ used in each rejection-sampling iteration. We require a sampler that outputs a random vector

$$(\mathbf{y}_1, \mathbf{y}_2) \in \mathbb{Z}^{\ell \times N} \times \mathbb{Z}^{k \times N},$$

whose distribution is (approximately) Gaussian, conditioned on the norm constraint

$$\left\|(\mathbf{y}_1, \mathbf{y}_2)\right\|_2 \le B_0\Lambda,$$

for some bound $B_0 > 0$ and scaling factor Λ (in our case $\Lambda = \ell+k$). A naïve implementation samples all $(\ell+k)N$ Gaussian coefficients, stores them in memory, computes their squared norm, rescales the entire vector, and finally applies a rejection test. This requires $\Theta((\ell+k)N)$ words of transient storage. The two-pass approach that avoids this large buffer was first

Algorithm 2 Streamed hyperball sampler

Require: Gaussian dimension N, integers L, K; seed $\mathsf{seed} \in \{0,1\}^{\mathsf{CRHBYTES}}$; nonce $\mathsf{nonce} \in \{0,1\}^{16}$; hyperball bound B_0 and scaling constant Λ (e.g. $\Lambda = L + K$)
Ensure: $(\mathbf{y}_1, \mathbf{y}_2) \in \mathbb{Z}^{L\times N} \times \mathbb{Z}^{K\times N}$ with $\|(\mathbf{y}_1, \mathbf{y}_2)\|_2 \le B_0\Lambda$

repeat
▷ Pass 1: compute the (unnormalized) squared norm S
  `InitStream(st, seed, nonce)`
  $S \leftarrow 0$ ▷ S is a scalar accumulator (e.g. 64-bit integer)
  for $i \leftarrow 0$ to $L + K - 1$ **do**
    for $j \leftarrow 0$ to $N - 1$ **do**
      $x_{i,j} \leftarrow$ `SampleGauss(st)`
      $S \leftarrow S + x_{i,j}^2$
    end for
  end for
▷ Compute scaling factor $\alpha \approx \dfrac{B_0\Lambda}{\sqrt{S}}$
  $\alpha \leftarrow$ `InvSqrt`(S, B_0, Λ)
▷ Pass 2: regenerate samples and scale immediately
  `InitStream(st, seed, nonce)`
  for $i \leftarrow 0$ to $L - 1$ **do**
    for $j \leftarrow 0$ to $N - 1$ **do**
      $x_{i,j} \leftarrow$ `SampleGauss(st)`
      $y_{1,i,j} \leftarrow$ `Round`$(\alpha \cdot x_{i,j})$
    end for
  end for
  for $i \leftarrow 0$ to $K - 1$ **do**
    for $j \leftarrow 0$ to $N - 1$ **do**
      $x_{L+i,j} \leftarrow$ `SampleGauss(st)`
      $y_{2,i,j} \leftarrow$ `Round`$(\alpha \cdot x_{L+i,j})$
    end for
  end for
until $\|(\mathbf{y}_1, \mathbf{y}_2)\|_2^2 \le B_0^2\Lambda^2$

described by [LWKP24] for BLISS and is also adopted by [HCK+26] for HAETAE-5. We apply the same high-level structure but further eliminate all static (BSS) scratch buffers through a fully streaming Gaussian backend.

Algorithm 2 implements what we call the *two-pass* sampler with *streaming* Gaussian sampling and it only uses $O(1)$ additional memory. The sampler is parameterized by a seed $\mathsf{seed} \in \{0,1\}^{\mathsf{CRHBYTES}}$ and a nonce nonce, which are used to initialize a pseudorandom stream st (e.g. based on SHAKE or stream256).

In the first pass, the algorithm draws $(\ell + k)N$ independent discrete Gaussian samples $(x_{i,j})_{0\le i<\ell+k,\ 0\le j<N}$ from `SampleGauss(st)`, and maintains only the scalar accumulator

$$S \;=\; \sum_{i=0}^{\ell+k-1} \sum_{j=0}^{N-1} x_{i,j}^2.$$

No sample is stored beyond its contribution to S, so the memory footprint of this pass is independent of N, ℓ, and k. After the loop, the algorithm computes a scaling factor

$$\alpha \approx \frac{B_0\Lambda}{\sqrt{S}},$$

using a fixed-point approximation of the reciprocal square root. The following proposition proves that the two-pass streaming implementation produces an identical output distribution to the one-pass reference procedure using the same fixed-point `InvSqrt` routine.

Proposition 1. *Conditioned on acceptance, the joint distribution of* $(\mathbf{y}_1, \mathbf{y}_2)$ *produced by Algorithm 2 is identical to that of the one-pass reference procedure that (i) draws all* $(L + K)N$ *Gaussian samples into a single array, (ii) computes the scaling factor* $\alpha = \texttt{InvSqrt}(\sum x_{i,j}^2, B_0, \Lambda)$*, (iii) maps each* $x_{i,j} \mapsto \texttt{Round}(\alpha x_{i,j})$*, and (iv) accepts if and only if* $\|(\mathbf{y}_1, \mathbf{y}_2)\|_2^2 \le B_0^2 \Lambda^2$*.*

Proof. Fix any seed and nonce. Since `InitStream`(st, seed, nonce) is deterministic and `SampleGauss`(st) is a pure function of the stream state, restarting the stream with the same (seed, nonce) reproduces the identical sequence $(x_{i,j})$ in both passes. Consequently, the scalar $S = \sum_{i,j} x_{i,j}^2$ and the scaling factor $\alpha = \texttt{InvSqrt}(S, B_0, \Lambda)$ computed in Pass 1 are identical to those that would be computed from the full array in the one-pass variant. The output coefficients $\texttt{Round}(\alpha x_{i,j})$ and the acceptance predicate $\|(\mathbf{y}_1, \mathbf{y}_2)\|_2^2 \le B_0^2 \Lambda^2$ are therefore also identical. The two procedures thus define the same mapping from (seed, nonce) to $(\mathbf{y}_1, \mathbf{y}_2)$ (or to rejection), and hence the same conditional distribution given acceptance. □

Therefore the only difference is implementation strategy: the two-pass sampler never stores more than a bounded number of Gaussian samples at any one time. A further quantitative analysis of the fixed-point approximation error is left to future work.

Streaming Gaussian backend. The reference HAETAE sampler allocates temporary arrays `samples[N·(ℓ+k)]` and `signs[N·(ℓ+k)/8]` in stack or BSS, requiring $\Theta((\ell + k)N)$ words of transient storage. Our implementation replaces this with a fully streaming backend: each discrete Gaussian sample is generated from the SHAKE sponge state, consumed, and discarded immediately. Since squeezing one sample at a time produces the identical output as batch squeezing, this has no effect on the output distribution. This eliminates all static sampler buffers, reducing the scheme-wide `.bss` to zero bytes.

5 Implementation and Evaluation

This section evaluates our low-stack HAETAE implementation. We report primary results on the `pqm4` benchmarking framework [KPR+] (Nucleo-L4R5ZI, ARM Cortex-M4), which enables direct comparison with the prior work of [HCK+26] and with ML-DSA. We additionally validate portability under RIOT-OS on two further targets (nRF52840 and ESP32-C6).

5.1 Build configuration

Our implementation is controlled by three compile-time knobs:

```
SAMPLER=2  STREAM_MATRIX=1  FROZEN_A=2
```

`SAMPLER=2` selects the two-pass streaming hyperball sampler (Section 4.4); `STREAM_MATRIX=1` regenerates matrix entries on demand from the public seed; and `FROZEN_A=2` fuses rejection sampling with pointwise multiplication to eliminate temporary polynomial buffers. When all three are enabled, the full-streaming signing path (`FULL_STREAM_SIGN`) is activated automatically, applying the *Rejection-aware pass decomposition* and *Reverse-order streaming entropy coding* techniques described in Section 4.2.

5.2 Results on `pqm4` framework (Nucleo-L4R5ZI)

All measurements are performed on the **NUCLEO-L4R5ZI** board (STM32L4R5ZI, ARM Cortex-M4F, 2 MB Flash, 640 kB RAM), the default `pqm4` target and the platform used

Table 3: Performance comparison on Nucleo-L4R5ZI (pqm4). Cycles: `-O3`, 20 MHz/0 WS, median of 100 runs ($k = 10^3$). Stack/size: `-Os`. Ratio relative to **ref** (<1: faster). (C) = pure C; (asm) = assembly. [HCK+26]: published numbers (HAETAE-5 only). **ours (C)**/**ours (asm)**: this work. ML-DSA from `pqm4`; **m4fstack** followed [BRS22] strategy.

	Key generation			Signing			Verification				
Impl.	Cycles	Ratio	Stack	Cycles	Ratio	Stack	Cycles	Ratio	Stack	.text	.total
HAETAE-2											
ref (C)	9,253 k	1.00×	23,844	43,710 k	1.00×	73,112	1,732 k	1.00×	33,448	26,490	29,098
m4f (asm)	6,980 k	0.75×	19,772	18,616 k	0.43×	55,684	998 k	0.58×	23,296	30,104	30,624
ours (C)	11,630 k	1.26×	5,848	115,534 k	2.64×	5,968	1,417 k	0.82×	4,936	30,416	30,416
ours (asm)	11,518 k	1.24×	5,816	81,416 k	1.86×	5,968	1,071 k	0.62×	4,824	38,100	38,100
HAETAE-3											
ref (C)	12,007 k	1.00×	41,236	53,546 k	1.00×	113,328	3,170 k	1.00×	54,232	26,436	29,204
m4f (asm)	13,584 k	1.13×	29,484	28,275 k	0.53×	83,436	1,909 k	0.60×	31,776	30,078	30,758
ours (C)	20,998 k	1.75×	5,848	181,792 k	3.40×	6,152	2,897 k	0.91×	4,840	30,780	30,780
ours (asm)	21,889 k	1.82×	5,920	138,771 k	2.59×	6,152	2,134 k	0.67×	4,840	38,464	38,464
HAETAE-5											
ref (C)	16,491 k	1.00×	52,500	65,896 k	1.00×	144,408	3,968 k	1.00×	68,856	25,658	28,786
m4f (asm)	20,115 k	1.22×	34,196	34,995 k	0.53×	103,988	2,532 k	0.64×	37,292	29,756	30,796
[HCK+26] (C)	15,328 k	0.93×	5,212	300,881 k	4.57×	8,092	4,187 k	1.06×	6,220	26,494	27,534
ours (C)	11,561 k	0.70×	4,816	218,957 k	3.32×	6,136	3,510 k	0.88×	4,840	30,150	30,150
ours (asm)	10,423 k	0.63×	4,888	163,125 k	2.48×	6,136	2,736 k	0.69×	4,952	37,834	37,834
ML-DSA-44											
m4f (asm)	1,418 k	1.00×	38,312	3,017 k	1.00×	44,832	1,496 k	1.00×	8,912	19,592	19,592
m4fstack (asm)	1,797 k	1.27×	4,408	10,330 k	3.42×	5,064	3,816 k	2.55×	2,720	24,844	24,844
ML-DSA-65											
m4f (asm)	2,526 k	1.00×	60,840	5,963 k	1.00×	68,896	2,533 k	1.00×	9,888	19,328	19,328
m4fstack (asm)	3,422 k	1.35×	4,408	21,668 k	3.63×	6,608	6,771 k	2.67×	2,720	24,120	24,120
ML-DSA-87											
m4f (asm)	4,267 k	1.00×	97,704	7,145 k	1.00×	107,912	4,381 k	1.00×	12,064	19,500	19,500
m4fstack (asm)	5,770 k	1.35×	4,408	27,150 k	3.80×	8,144	11,713 k	2.67×	2,720	24,516	24,516

by [HCK+26]. Speed measurements use `CLOCK_BENCHMARK` (20 MHz, 0 WS flash) for cycle-accurate results; stack measurements use `CLOCK_FAST` (120 MHz). Stack measurements follow the `pqm4` framework's built-in stack-benchmarking infrastructure. Compiler optimization follows the `pqm4` convention: `-Os` for stack and code-size measurements, `-O3` for cycle-count benchmarks (separate builds). We report the median over 100 executions. All reported stack peaks are empirical measurements obtained with `arm-none-eabi-gcc` 11.3.1 under the default `pqm4` build configuration (`-Os`, no link-time optimization). The `noinline` pass boundaries rely on compiler support for the `__attribute__((noinline))` annotation.

We benchmark four HAETAE configurations: **ref (C)** (latest C reference code [Kpq26]), **m4f (asm)** (pqm4 assembly, older spec [CCD+24]), **ours (C)** (this work, pure C), and **ours (asm)** (this work with assembly NTT and sampler from [CCD+24]). Both **ours** variants use the build knobs from Section 5.1. Since the implementation of [HCK+26] is not publicly available, we report their published numbers directly. Table 3 presents the results.

Stack usage. Compared to the reference, **ours (C)** reduces peak Signing stack by 91.8–95.8 % across all levels, bringing it to 5,968 B (HAETAE-2), 6,152 B (HAETAE-3), and 6,136 B (HAETAE-5). Verification stack ranges from 4,840 B to 4,936 B, an 85–93 % reduction. Key generation ranges from 4,816 B (HAETAE-5) to 5,848 B (HAETAE-2/3).

ours (asm) achieves comparable stack across all operations; the small differences (e.g. 4,824 B vs. 4,936 B for HAETAE-2 verification) reflect differing callee frame sizes in the assembly routines. Note that the **m4f** implementation is based on an older version of the specification and includes ARM assembly optimizations, so a direct comparison with our implementation is not entirely fair; we include it for completeness. For HAETAE-5, the only level reported by [HCK+26], our implementation achieves lower stack across all three operations:

- Key generation: 4,816 B vs. 5,212 B (−7.6 %). Caller-level union analysis (Section 4.1) packs the FFT workspace and sampling scratch into shared memory slots.
- Signing: 6,136 B vs. 8,092 B (−24 %). The pass decomposition with `noinline` boundaries (Section 4.2) ensures the peak is bounded by $S_{\text{driver}} + \max(S_A, S_B, S_{C_1}, S_{C_2})$, and the *Reverse-order streaming entropy coding* eliminates the full hint and HB^{z_1} staging buffers.
- Verification: 4,840 B vs. 6,220 B (−22 %). Our row-streamed design (Section 4.3) replaces the column-streamed `polyveck` accumulator with a single polynomial, combined with view-style decoding and incremental transcript hashing.

Both variants have `.data` = 0 and `.bss` = 0, whereas [HCK+26] reports a 1,040 B gap between `.text` and `.total`, whose breakdown is not disclosed.

Cycle counts. Key generation for HAETAE-5 is 30 % faster than the reference in **ours (C)** (0.70×) and 37 % faster with **ours (asm)** (0.63×). Since d=0 key generation (Algorithm 4) performs the singular-value norm rejection before expanding the matrix $\mathbf{A}$, with an acceptance rate of roughly 10 %, most iterations only execute the lightweight norm check, and the matrix–vector product runs only once after acceptance. For HAETAE-2/3 (d>0), the norm check is interleaved with the matrix–vector product, so each rejected sample also pays the streaming matrix cost, resulting in 1.24–1.82× slower key generation.

Signature generation requires 1.86–3.40× the cycles of the reference (**ours (C)**: 2.64–3.40×; **ours (asm)**: 1.86–2.59×), reflecting the cost of seed-based recomputation in every pass. For HAETAE-5, **ours (C)** achieves 218,957 k cycles, 27 % faster than [HCK+26] (300,881 k), with 24 % lower stack; *Component-level early rejection* (Section 4.2) reduces the average cost of rejected iterations by skipping unnecessary multiply-accumulate operations, contributing to this speedup. **Ours (asm)** further reduces this to 163,125 k (−46 % vs. [HCK+26]).

Verification is faster than the reference at all levels for both variants (**ours (C)**: 0.82–0.91×; **ours (asm)**: 0.62–0.69×). For HAETAE-5, [HCK+26] achieves 1.06× with 6,220 B stack, whereas **ours (C)** achieves 0.88× with 4,840 B, and **ours (asm)** reaches 0.69×. We attribute the speedup primarily to the smaller working set, which reduces memory traffic on the Cortex-M4.

Code size. **Ours (C)** shows a modest `.text` increase compared to the reference (30,150 vs. 28,786 for HAETAE-5, +4.7 %), reflecting the streaming rANS encoder, fused multiplication, and pass decomposition logic. **Ours (asm)** is larger (37,834 for HAETAE-5, +31 % vs. ref) due to the inlined NTT and CDT sampler routines. Both variants place all constant tables in read-only memory, resulting in `.data` = 0 and `.bss` = 0; the reported `.text` and `.total` are therefore identical. ML-DSA scheme-only code sizes (19 kB to 24 kB) are smaller than HAETAE, reflecting the simpler structure of ML-DSA.

5.3 Results on RIOT-OS (nRF52840 and ESP32-C6)

To validate portability beyond the `pqm4` bare-metal environment, we ran the same low-stack build under RIOT-OS [RIO] on two targets: the **Nordic nRF52840** (ARM Cortex-M4F,

Table 4: Low-stack evaluation on RIOT-OS (**this work**). nRF52840: ARM Cortex-M4F @ 64 MHz, 256 kB RAM. ESP32-C6: RISC-V RV32IMAC @ 80 MHz, 512 kB RAM. Cycles measured via `xtimer` function from RIOT-OS.

	Key generation		Signing		Verification	
Platform	Cycles	Stack	Cycles	Stack	Cycles	Stack
HAETAE-2						
nRF52840	29,039 k	5,988	262,721 k	6,096	2,023 k	3,984
ESP32-C6	49,371 k	6,100	476,223 k	6,068	3,619 k	4,004
HAETAE-3						
nRF52840	46,520 k	6,208	829,928 k	6,288	3,838 k	3,984
ESP32-C6	80,811 k	6,580	1,519,566 k	6,260	7,146 k	4,004
HAETAE-5						
nRF52840	13,612 k	5,096	193,913 k	6,406	4,970 k	3,976
ESP32-C6	21,513 k	5,254	526,047 k	6,244	9,536 k	4,004

Table 5: RAM budget for Verification and Signing (bytes). Verify total: $|sig|$ + stack (public key in flash). Sign total: $|sk| + |sig|$ + stack. Cycles: $k = 10^3$, from `pqm4` (Table 3).

				Verification			Signing		
Scheme	$\|pk\|$	$\|sk\|$	$\|sig\|$	Stack	Total	Cycles	Stack	Total	Cycles
HAETAE-2 (**ours (C)**)	992	1 408	1 474	4 936	6 410	1,417 k	5 968	8 850	115,534 k
HAETAE-3 (**ours (C)**)	1 472	2 112	2 349	4 840	7 189	2,897 k	6 152	10 613	181,792 k
HAETAE-5 (**ours (C)**)	2 080	2 752	2 948	4 840	7 788	3,510 k	6 136	11 836	218,957 k
HAETAE-5 [HCK+26]	2 080	2 752	2 948	6 220	9 168	4,187 k	8 092	13 792	300,881 k
ML-DSA-44 (m4fstack)	1 312	2 560	2 420	2 720	5 140	3,816 k	5 064	10 044	10,330 k
ML-DSA-65 (m4fstack)	1 952	4 032	3 309	2 720	6 029	6,771 k	6 608	13 949	21,668 k
ML-DSA-87 (m4fstack)	2 592	4 896	4 627	2 720	7 347	11,713 k	8 144	17 667	27,150 k

64 MHz, 256 kB RAM) and the **Espressif ESP32-C6** (RISC-V RV32IMAC, 80 MHz, 512 kB RAM). Stack usage is measured via a high-watermark technique: the unused stack region below the current stack pointer is painted with a canary pattern before each operation, and the deepest overwrite is recorded after the operation returns. Timing uses RIOT's `xtimer` layer converted to cycles via `coreclock_hz`. Results are averaged over 100 iterations.

As Table 4 shows, peak stack figures are consistent between the two RIOT-OS targets. For HAETAE-2, signature generation peaks at 6,096 B (nRF52840) and 6,068 B (ESP32-C6), confirming that the memory footprint is determined by the algorithm and build knobs, not by the ISA or OS layer.

Cycle-count gap between Cortex-M4F and RV32IMAC. Despite a 25 % higher clock frequency, the ESP32-C6 requires substantially more cycles for every operation. The slowdown is most pronounced for signing and other operations, the main reason is the absence of DSP/SIMD hardware on the in-order RV32IMAC core: the Cortex-M4F benefits from a single-cycle 32-bit multiplier suited to the NTT-heavy polynomial arithmetic in HAETAE.

6 Discussion

To provide a fair basis for comparison, the discussion below is based exclusively on `pqm4` measurements (Table 3) using the same board, clock configuration, and measurement infrastructure for all schemes.

Verification on 8 KB devices. On many constrained platforms, the public key is stored in flash memory, so the RAM budget for verification consists of the received signature plus the stack required to run the algorithm (Table 5), as is the case in, e.g., [BZB+22, BDKR23]. On devices with as little as 8 kB of SRAM, the available RAM is shared among `.data`, `.bss`, and the runtime stack, meaning that static data sections directly reduce the stack budget. Our implementation has `.data` = 0 and `.bss` = 0, leaving the full RAM available for the stack. In contrast, [HCK+26] reports a 1,040 B gap between `.text` and `.total`, but does not disclose the section-level breakdown. **Ours (C)** fits all three HAETAE security levels within 8 kB (6,410 B to 7,788 B), whereas [HCK+26] requires 9,168 B for HAETAE-5 verification alone. Compared to ML-DSA m4fstack, HAETAE verification is 2.34–3.34× faster at each comparable security level (1,417 k vs. 3,816 k at level 2, 2,897 k vs. 6,771 k at level 3, 3,510 k vs. 11,713 k at level 5). This makes HAETAE attractive for signature-verification-centric deployments such as firmware authentication and IoT attestation.

Signature and key generation on 16 KB devices. For signing, the device must hold the secret key, the output signature, and the signing working set simultaneously (Table 5). All HAETAE levels fit within 16 kB (8,850 B to 11,836 B), while ML-DSA-87 exceeds this budget (17,667 B) due to the large secret key (4,896 B) and signing stack (8,144 B). At security level 5, HAETAE-5 requires 11,836 B for signing, roughly 5.7 kB less than ML-DSA-87. Key generation is comfortable for both schemes: our implementation requires at most 5,848 B of stack, well within the 16 kB budget.

References

[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.

[BDKR23] Joppe W. Bos, Alexander Dima, Alexander Kiening, and Joost Renes. Post-quantum secure over-the-air update of automotive systems. Cryptology ePrint Archive, Paper 2023/965, 2023.

[BHG+13] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 79–80, 2013.

[BRS22] Joppe W. Bos, Joost Renes, and Amber Sprenkels. Dilithium for memory constrained devices. In Lejla Batina and Joan Daemen, editors, *Progress in Cryptology - AFRICACRYPT 2022: 13th International Conference on Cryptology in Africa, AFRICACRYPT 2022, Fes, Morocco, July 18-20, 2022, Proceedings*, volume 13503 of *Lecture Notes in Computer Science*, pages 217–235. Springer Nature Switzerland, 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 Giuseppe Ateniese and Daniele Venturi, editors, *Applied Cryptography and Network Security - 20th International Conference, ACNS 2022, Rome, Italy, June 20-23, 2022, Proceedings*, Lecture Notes in Computer Science, pages 872–891. Springer, 2022.

[CCD+23] Jung Hee Cheon, Hyeongmin Choe, Julien Devevey, Tim Güneysu, Dongyeon Hong, Markus Krausz, Georg Land, Junbum Shin, Damien Stehlé, and MinJune Yi. HAETAE. Technical report, National Institute of Standards and Technology, 2023. available at https://csrc.nist.gov/Projects/pqc-dig-sig/round-1-additional-signatures.

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

[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, Paper 2026/442, 2026.

[Kor] Korean Post-Quantum Cryptography Research Group. KpqC Algorithms final specification documents. Accessed: 2026-03-30.

[Kpq26] Korean Post-Quantum Cryptography Standardization Committee. *HAETAE: Shorter Lattice-Based Fiat-Shamir Signatures*, 2026. Final specification. available at https://www.kpqc.or.kr/images/pdf2/HAETAE.pdf.

[KPR+] Matthias J. Kannwischer, Richard Petri, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. pqm4: Post-quantum crypto library for the ARM Cortex-M4. https://github.com/mupq/pqm4.

[LWKP24] Seungwoo Lee, Joo Woo, Jonghyun Kim, and Jong Hwan Park. Generalized centered binomial distribution for bimodal lattice signatures. *IEEE Access*, 13:2203–2214, 2024.

[Lyu09] Vadim Lyubashevsky. Fiat-Shamir with aborts: Applications to lattice and factoring-based signatures. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 598–616. Springer, 2009.

[NIS23] National Institute of Standards and Technology. *Module-Lattice-Based Digital Signature Standard (ML-DSA)*, 2023. Federal Information Processing Standards Publication 204 (Initial Public Draft), https://doi.org/10.6028/NIST.FIPS.204.ipd.

[RIO] RIOT Community. Riot – the friendly operating system for the internet of things. Accessed: March 31, 2026.

A Key Generation

Algorithm 3 KeyGen(1^λ) for $d > 0$

Ensure: (pk, sk)
1: $\text{seed} \leftarrow \{0,1\}^{\rho_0}$
2: $(\text{seed}_{\mathbf{A}}, \text{seed}_{sk}, K) \leftarrow H_{\text{gen}}(\text{seed})$
3: $(\mathbf{a}_{\text{gen}}, \widehat{\mathbf{A}}_{\text{gen}}) \leftarrow \mathsf{ExpandA}_d(\text{seed}_{\mathbf{A}})$
4: $(\text{counter}_{sk}, \text{flag}) \leftarrow (0, \text{true})$
5: **while** flag **do**
6: $\quad(\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \leftarrow \mathsf{ExpandS}(\text{seed}_{sk}, \text{counter}_{sk})$
7: $\quad\mathbf{b} \leftarrow \mathbf{a}_{\text{gen}} + \mathsf{NTT}^{-1}(\widehat{\mathbf{A}}_{\text{gen}} \circ \mathsf{NTT}(\mathbf{s}_{\text{gen}})) + \mathbf{e}_{\text{gen}} \bmod q$
8: $\quad(\mathbf{b}_0, \mathbf{b}_1) \leftarrow (\mathsf{LowBits}^{pk}(\mathbf{b}), \mathsf{HighBits}^{pk}(\mathbf{b}))$
9: $\quad(\mathbf{s}_1, \mathbf{s}_2) \leftarrow (\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}} - \mathbf{b}_0)$
10: $\quad\text{counter}_{sk} \leftarrow \text{counter}_{sk} + 1$
11: $\quad$**if** $\mathcal{N}(\mathbf{s}_1, \mathbf{s}_2) \le \gamma^2 n$ **then**
12: $\quad\quad\text{flag} \leftarrow \text{false}$
13: $\quad$**end if**
14: **end while**
15: $tr \leftarrow H(\text{seed}_{\mathbf{A}}, \mathbf{b}_1)$
16: **return** $(pk = (\text{seed}_{\mathbf{A}}, \mathbf{b}_1), sk = (\mathbf{s}_1, \mathbf{s}_2, K, tr, \text{seed}_{\mathbf{A}}, \mathbf{b}_1))$

Algorithm 4 KeyGen(1^λ) for $d = 0$

Ensure: (pk, sk)
1: $\text{seed} \leftarrow \{0,1\}^{\rho_0}$
2: $(\text{seed}_{\mathbf{A}}, \text{seed}_{sk}, K) \leftarrow H_{\text{gen}}(\text{seed})$
3: $(\text{counter}_{sk}, \text{flag}) \leftarrow (0, \text{true})$
4: **while** flag **do**
5: $\quad(\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \leftarrow \mathsf{ExpandS}(\text{seed}_{sk}, \text{counter}_{sk})$
6: $\quad(\mathbf{s}_1, \mathbf{s}_2) \leftarrow (\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}})$
7: $\quad\text{counter}_{sk} \leftarrow \text{counter}_{sk} + 1$
8: $\quad$**if** $\mathcal{N}(\mathbf{s}_1, \mathbf{s}_2) \le \gamma^2 n$ **then**
9: $\quad\quad\text{flag} \leftarrow \text{false}$
10: $\quad$**end if**
11: **end while**
12: $\widehat{\mathbf{A}}_{\text{gen}} \leftarrow \mathsf{ExpandA}_d(\text{seed}_{\mathbf{A}})$
13: $\widehat{\mathbf{b}} \leftarrow -2\left(\widehat{\mathbf{A}}_{\text{gen}} \circ \mathsf{NTT}(\mathbf{s}_{\text{gen}}) + \mathsf{NTT}(\mathbf{e}_{\text{gen}})\right) \bmod q$
14: $tr \leftarrow H(\text{seed}_{\mathbf{A}}, \widehat{\mathbf{b}})$
15: **return** $(pk = (\text{seed}_{\mathbf{A}}, \widehat{\mathbf{b}}), sk = (\mathbf{s}_1, \mathbf{s}_2, K, tr, \text{seed}_{\mathbf{A}}, \widehat{\mathbf{b}}))$

Figure 5: Implementation specification of HAETAE key generation [CCD+23, Kpq26]: $d > 0$ applies to HAETAE-2 and HAETAE-3, while $d = 0$ applies to HAETAE-5.

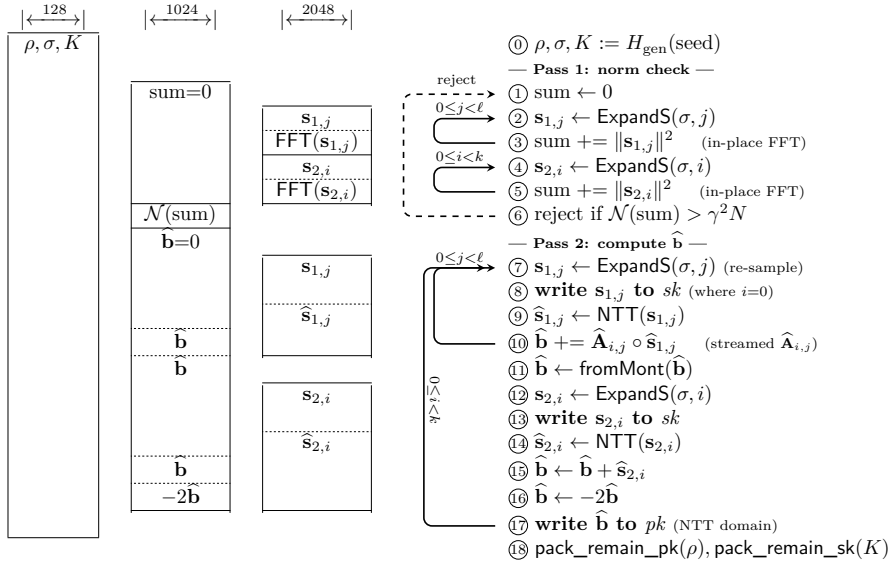

Figure 6: Memory allocation of HAETAE-5 Key generation (d=0). Two caller-level unions share memory across passes: `union{sum[N]; poly b}` (1 024 B) and `union{fft_input[FFT_N]; poly s}` (2 048 B). Pass 1 checks the norm before expanding $\widehat{\mathbf{A}}$; Pass 2 computes $\widehat{\mathbf{b}} = \mathsf{NTT}(-2\mathbf{b})$.

B Signature Generation

Algorithm 5 Implementation Specification of HAETAE Signing [CCD+23, Kpq26]

Require: $sk = (\mathbf{s}_1, \mathbf{s}_2, K, tr, \text{seed}_{\mathbf{A}}, \psi)$, message M
Ensure: signature σ
1: $\widehat{\mathbf{A}} \leftarrow \mathsf{UnpackA}_d(\text{seed}_{\mathbf{A}}, \psi)$
2: $\mu \leftarrow H_{\mathtt{gen}}(tr, M)$
3: $\text{seed}_{ybb} \leftarrow H_{\mathtt{gen}}(K, \mu)$
4: $(\kappa, \sigma) \leftarrow (0, \perp)$
5: **while** $\sigma = \perp$ **do**
6: $(\mathbf{y}_1, \mathbf{y}_2, b, b', \kappa) \leftarrow \mathsf{ExpandYbb}(\text{seed}_{ybb}, \kappa)$
7: $\mathbf{w} \leftarrow \mathsf{NTT}^{-1}(\widehat{\mathbf{A}} \circ \mathsf{NTT}(\lfloor \mathbf{y}_1 \rceil)) + 2 \cdot \lfloor \mathbf{y}_2 \rceil \bmod q$
8: $\mathbf{w}' \leftarrow \mathsf{fromCRT}(\mathbf{w}, \lfloor y_{1,1} \rceil)$
9: $\mathbf{w}'_1 \leftarrow \mathsf{HighBits}^h(\mathbf{w}')$
10: $\rho \leftarrow H(\mathbf{w}'_1, \mathsf{LSB}(\lfloor y_{1,1} \rceil), \mu)$
11: $c \leftarrow \mathsf{SampleBinaryChallenge}_\tau(\rho)$
12: $\widehat{c} \leftarrow \mathsf{NTT}(c)$
13: $z_{1,1} \leftarrow y_{1,1} + (-1)^b \cdot c$
14: $(\mathbf{z}_1)_{2..\ell} \leftarrow (\mathbf{y}_1)_{2..\ell} + (-1)^b \mathsf{NTT}^{-1}(\widehat{c} \circ \widehat{\mathbf{s}}_1)$
15: $\mathbf{z}_2 \leftarrow \mathbf{y}_2 + (-1)^b \mathsf{NTT}^{-1}(\widehat{c} \circ \widehat{\mathbf{s}}_2)$
16: **if** $\|(\mathbf{z}_1, \mathbf{z}_2)\|_2 < B'$ **and** $(\|2(\mathbf{z}_1, \mathbf{z}_2) - (\mathbf{y}_1, \mathbf{y}_2)\|_2 > B$ **or** $b' = 1)$ **then**
17: $\mathbf{h} \leftarrow \mathbf{w}'_1 - \mathsf{HighBits}^h(\mathbf{w}' - 2\lfloor \mathbf{z}_2 \rceil) \bmod^+ \frac{2(q-1)}{\alpha_h}$
18: $\sigma \leftarrow \mathsf{PackSig}(\mathsf{HighBits}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathsf{LowBits}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathbf{h}, c)$
19: **end if**
20: **end while**

C Verification

Algorithm 6 Implementation Specification of HAETAE Verification [CCD+23, Kpq26]

Require: $pk = (\text{seed}_{\mathbf{A}}, \psi)$, message M, signature σ
Ensure: Accept or Reject
1: $\widehat{\mathbf{A}} \leftarrow \mathsf{UnpackA}_d(\text{seed}_{\mathbf{A}}, \psi)$
2: $(\mathsf{HighBits}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathsf{LowBits}^{z_1}(\lfloor \mathbf{z}_1 \rceil), \mathbf{h}, c) \leftarrow \mathsf{UnpackSig}(\sigma)$ ▷ Can fail and rejection
3: $\tilde{\mathbf{z}}_1 \leftarrow \mathsf{HighBits}^{z_1}(\lfloor \mathbf{z}_1 \rceil) \cdot 256 + \mathsf{LowBits}^{z_1}(\lfloor \mathbf{z}_1 \rceil)$
4: $w' \leftarrow \mathsf{LSB}(\tilde{z}_{1,1} - c)$
5: $\tilde{\mathbf{w}} \leftarrow \widehat{\mathbf{A}} \circ \mathsf{NTT}(\tilde{\mathbf{z}}_1) \bmod q$
6: $\tilde{\mathbf{w}}' \leftarrow \mathsf{fromCRT}(\tilde{\mathbf{w}}, w')$
7: $\tilde{\mathbf{w}}'_1 \leftarrow \tilde{\mathbf{h}} + \mathsf{HighBits}^h(\tilde{\mathbf{w}}') \bmod^+ \frac{2(q-1)}{\alpha_h}$
8: $\tilde{\mathbf{z}}_2 \leftarrow [\tilde{\mathbf{w}}'_1 \cdot \alpha_h + w'\mathbf{j} - \tilde{\mathbf{w}}' \bmod 2q] \,/\, 2$ ▷ Addition with w' only for first vector element
9: $\tilde{\mu} \leftarrow H_{\mathtt{gen}}(\text{seed}_{\mathbf{A}}, \psi, M)$
10: **return** $(c = \mathsf{SampleBinaryChallenge}_\tau(H(\tilde{\mathbf{w}}'_1, w', \tilde{\mu}))) \ \wedge \ (\|(\tilde{\mathbf{z}}_1, \tilde{\mathbf{z}}_2)\| < B'')$