Memory-Efficient Implementation of SMAUG-T and HAETAE

Yulim Hyoung[1 [0009−0009−7530−6497]], Subeen Cho[1 [0009−0004−4352−8021]], Uijae Kim[1 [0009−0004−2617−9659]], Minwoo Lee[2 [0000−0002−2356−3055]], Hwajeong Seo[1 [0000−0003−0069−9061]], and Minjoo Sim[2 [0000−0001−5242−214X]] ⋆

[1] Department of Convergence Security, Hansung University, Seoul (02876), South Korea
{yulim4hyoung, chosubin1208, chrisvt424}@gmail.com
[2] Department of Information Computer Engineering, Hansung University, Seoul (02876), South Korea
{minunejip, hwajeong84}@gmail.com

Abstract. SMAUG-T and HAETAE, designated as target algorithms for national standardization via the Korean Post-Quantum Cryptography (KpqC) competition, run efficiently on general-purpose platforms. On ARM Cortex-M4 class microcontrollers, however, peak stack usage becomes a key constraint: while SMAUG-T can be executed on typical Cortex-M4 boards, the baseline HAETAE implementation exceeds the available SRAM (e.g., 91,176 B stack for signing), motivating dedicated memory optimization. To address this problem, we propose a suite of memory optimization techniques for SMAUG-T and HAETAE that enable their practical operation within the strict memory budget of the Cortex-M4. Experimental results demonstrate that, compared to the KpqClean_ver2 baseline, peak stack usage was reduced by 73–83 % for SMAUG-T5 (e.g., 24,300 B→4,240 B in decapsulation) and by about 90 % for HAETAE5 (e.g., 91,176 B→8,092 B in signing). Furthermore, a branchless constant-time design was applied throughout to ensure that the optimized implementations remain robust against side-channel threats such as timing attacks. This work provides a practical methodology for deploying KpqC lattice-based cryptography in memory-constrained embedded environments.

Keywords: Memory Optimization · KpqC · SMAUG-T · HAETAE · Software Optimization · Cortex-M4 · Embedded Systems

1 Introduction

The rapid advancement of quantum computing technology presents a fundamental security threat to existing public-key cryptosystems—such as RSA [1] and ECC [2]—which rely on the mathematical hardness of integer factorization and discrete logarithm problems [3,4]. In response, the transition to post-quantum cryptography (PQC) is accelerating globally [5,6,7]. South Korea has

⋆ Corresponding author. minjoos9797@gmail.com

also designated AIMer [8], HAETAE [9], NTRU+ [10] and SMAUG-T [11] as target algorithms for national standardization via the KpqC competition [12], aiming to strengthen cryptographic sovereignty and address national security needs. Among the evaluated candidates, SMAUG-T and HAETAE are lattice-based schemes grounded in the computational hardness of the Module Learning With Errors (MLWE) and Module Learning With Rounding (MLWR) problems, respectively [11,9]. These lattice-based algorithms are expected to serve as core components of future security infrastructures, providing robust resistance against quantum-era threats.

However, significant technical barriers exist in deploying these PQC algorithms onto embedded devices. Platforms like the Cortex-M4 [13], widely used in IoT devices and security modules, provide only tens to hundreds of kilobytes of SRAM. In contrast, KpqC algorithms can require substantial *transient* memory during computation, and in particular their peak stack usage becomes a major constraint in resource-constrained environments. This motivates dedicated memory optimization to ensure practical and efficient execution on such devices.

The pqm4 project [14] has established a standardized benchmarking framework for evaluating post-quantum cryptographic implementations (including NIST PQC candidates) on the Cortex-M4, providing cycle counts, stack measurements, and code size metrics across a wide range of schemes. However, pqm4 is primarily designed for performance benchmarking and comparability, rather than for reducing stack usage to fit within tight memory limits. While the kpqm4 project [15] provides a benchmarking framework for KpqC algorithms on the Cortex-M4, it similarly focuses on performance measurement rather than memory reduction. Furthermore, neither pqm4 nor kpqm4 addresses implementations whose baseline stack usage already exceeds the device's available stack budget, a condition that makes benchmarking itself infeasible without prior memory optimization. Our work fills this gap by reducing the stack footprint of SMAUG-T and HAETAE to within the operational limits of the Cortex-M4.

While libraries such as KpqClean_ver2 [16] have contributed to improving the portability of these algorithms, they do not offer sufficiently granular optimization strategies to overcome the stringent memory limits of specific microcontrollers. Specifically, in embedded environments, forcibly reducing memory usage can lead to increased computation time. Furthermore, ensuring resistance against side-channel attacks—such as preventing unintended timing information leakage during the optimization process—remains a critical challenge. Consequently, there is an urgent need for research that identifies the optimal trade-off to maximize memory efficiency while preserving both security and computational performance.

We propose a memory optimization methodology and optimized implementations that enable efficient operation of SMAUG-T and HAETAE on ARM Cortex-M4 microcontrollers.

The remainder of this paper is organized as follows. Section 2 reviews the KpqC algorithms, the target Cortex-M4 platform, and related work. Section 3 describes the proposed memory optimization and security enhancement tech-

Table 1: Parameter sets of SMAUG-T

Name	SMAUG-T1	SMAUG-T3	SMAUG-T5
Security level	1	3	5
n	256	256	256
k	2	3	4
(q, p, t)	(1024, 256, 2)	(2048, 512, 2)	(2048, 512, 2)
p' (compression)	32	16	128
h_s (HWT for $\mathbf{s}$)	140	264	348
r (spCBD for $\mathbf{r}$)	1/8	1/4	3/16
σ ($\tilde{D}_\sigma$ for errors)	1.0625	1.0625	1.0625
Public key (bytes)	672	1,088	1,440
Secret key (bytes)	832	1,312	1,728
Ciphertext (bytes)	672	992	1,376

niques in detail. Section 4 presents the experimental results and performance analysis. Finally, Section 5 concludes the paper and suggests directions for future research.

1.1 Contributions

This paper makes the following contributions:

- **Strategic Phased Stack Reduction:** We present a phased stack reduction strategy specialized for embedded environments by precisely analyzing the memory usage patterns of SMAUG-T and HAETAE.
- **Practical and substantial memory savings:** Through the proposed techniques, we achieved substantial peak stack reductions compared to the KpqClean_ver2 baseline [16]. For SMAUG-T5, peak stack usage was reduced by **73–83%** (e.g., **24,300 B→4,240 B** in decapsulation), and for HAETAE5, peak stack usage was reduced by about **90%** (e.g., **91,176 B→8,092 B** in signing), enabling practical operation within strict Cortex-M4 SRAM budgets. We further quantify the resulting time–memory trade-off in Section 4.
- **Side-channel robust security hardening:** We apply constant-time design principles throughout the optimization process to avoid introducing secret-dependent control flow and to mitigate timing side-channel risks (e.g., branchless sparse multiplication and constant-time comparisons).

2 Background

2.1 Korean Post-Quantum Cryptography Competition

The Korean Post-Quantum Cryptography (KpqC) competition[12], initiated in 2022 to establish independent domestic cryptographic standards, has selected

Algorithm 1 Description of SMAUG.PKE[11]

KeyGen(1^λ):
$seed \leftarrow \{0,1\}^{256}$
$(seed_A, seed_{sk}, seed_e) \leftarrow XOF(seed)$
$\mathbf{A} \leftarrow expandA(seed_A) \in \mathcal{R}_q^{k\times k}$
$\mathbf{s} \leftarrow HWT_{h_s}(seed_{sk}) \in S_\eta^k$
$\mathbf{e} \leftarrow dGaussian_\sigma(seed_e) \in \mathcal{R}^k$
$\mathbf{b} = -\mathbf{A}\cdot\mathbf{s} + \mathbf{e} \in \mathcal{R}_q^k$
return $pk = (seed_A, \mathbf{b}), sk = \mathbf{s}$

Enc$(pk, \mu; seed_r)$:
$\mathbf{A} = expandA(seed_A)$
if $seed_r$ is not given **then** $seed_r \leftarrow \{0,1\}^{256}$
$\mathbf{r} \leftarrow HWT_{h_r}(seed_r) \in S_\eta^k$
$\mathbf{c}_1 = \lfloor p/q \cdot \mathbf{A}^\top \cdot \mathbf{r} \rceil \in \mathcal{R}_p^k$
$c_2 = \lfloor p'/q \cdot \langle \mathbf{b}, \mathbf{r}\rangle + p'/t \cdot \mu \rceil \in \mathcal{R}_{p'}$
return $ct = (\mathbf{c}_1, c_2)$

Dec$(sk, \mathbf{c})$:
$\mu' = \lfloor t/p \cdot \langle \mathbf{c}_1, \mathbf{s}\rangle + t/p' \cdot c_2 \rceil \in \mathcal{R}_t$
return μ'

NTRU+[10] and SMAUG-T[11] for Key Encapsulation Mechanisms (KEM), and AIMer[8] and HAETAE[9] for digital signatures as target algorithms for standardization following a three-year national evaluation process. Currently, the formal standardization process for these algorithms is underway.

2.2 SMAUG-T

SMAUG-T is defined on the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^{256}+1)$, with parameters (n, k, q) representing the polynomial degree (256), the rank of the modules and the modulus, respectively. The algorithm is characterized by its use of sparse secrets and a distinct error sampling method to enhance computational efficiency and reduce ciphertext size.

The key features and specifications of SMAUG-T are as follows:

- **Sparse Secret and HWT Sampling:** The secret key s consists of ternary polynomials with coefficients in $\{-1, 0, 1\}$. SMAUG-T employs Fixed-Weight Polynomial Sampling (HWT) to fix the ratio of non-zero components. This enables constant-time implementation based on shuffling, securing the algorithm against timing attacks.
- **LWR-based Encryption:** While key generation is based on the MLWE problem, the encryption process utilizes the MLWR problem. Instead of adding an error term during encryption, it performs a rounding operation, which reduces the size of the ciphertext and improves the speed of the operation.

Table 2: Parameter sets of HAETAE

Name	HAETAE-2	HAETAE-3	HAETAE-5
Security level	2	3	5
n (Degree of $\mathcal{R}$)	256	256	256
(k, ℓ) (Dimensions of $\mathbf{z}_2, \mathbf{z}_1$)	(2,4)	(3,6)	(4,7)
q (Modulus)	64513	64513	64513
η (sk coefficients range)	1	1	1
τ (Weight of c)	58	80	128
γ (Rejection parameter)	48.858	57.707	55.13
Key acceptance rate	0.1	0.1	0.1
d (Truncated bits of vk)	1	1	0
M (Expected repetitions)	6.0	5.0	6.0
B ($\mathbf{y}$ radius)	9846.02	18314.98	22343.66
B' (Rejection radius)	9838.98	18307.70	22334.95
B'' (Verify radius)	12777.52	21906.65	24441.49
α ($\mathbf{z}_1$ compression)	256	256	256
α_h ($\mathbf{h}$ compression)	512	512	256
Public Key (bytes)	992	1,472	2,080
Secret Key (bytes)	1,408	2,112	2,752
Signature (bytes)	1,474	2,349	2,948

– **dGaussian Sampling:** For error vector generation, SMAUG-T uses a dGaussian sampling technique that approximates a discrete Gaussian distribution using only 10-bit Boolean logic circuits. This allows for constant-time error generation without complex floating-point arithmetic.

Algorithm 1 provides a detailed description of the Key Generation (KeyGen), Encryption (Enc), and Decryption (Dec) procedures that constitute the core Public-Key Encryption (PKE) scheme of SMAUG-T[11].

2.3 HAETAE

HAETAE is a lattice-based digital signature algorithm that follows the 'Fiat-Shamir with Aborts (FSwA)' paradigm, similar to NIST's Crystals-Dilithium. It distinguishes itself by introducing uniform sampling in a 'Hyperball' region to optimize signature size and reduce rejection probability during signature generation.

The key features and specifications of HAETAE are as follows:

– **Algorithm Structure:** HAETAE comprises KeyGen based on the MLWE problem, Signing which verifies signature norms within a hyperball region, and Verification that confirms validity through z's size and hash consistency checks.

- **Bimodal Hyperball Sampling:** HAETAE combines a Bimodal distribution (inspired by BLISS) with a high-dimensional Hyperball uniform distribution. To ensure the signature z is independent of the secret key, it performs rejection sampling. By utilizing the Hyperball region, the acceptance space for signatures is expanded, reducing the signature size by up to 39% compared to Dilithium.
- **Fixed-point Arithmetic:** While some lattice signatures like Falcon rely on floating-point arithmetic, which can be vulnerable to side-channel attacks, HAETAE employs only fixed-point arithmetic. This simplifies the design and enhances resistance against side-channel attacks.

Algorithm 2 provides a detailed description of the Key Generation (KeyGen), Signing (Sign), and Verification (Verify) procedures that constitute the core digital signature scheme of HAETAE [9].

Algorithm 2 Uncompressed description of HAETAE[9]

KeyGen(1^λ):
$(\mathbf{A}_{\text{gen}}) \leftarrow \mathcal{R}_q^{k\times(\ell-1)}$ and $(\mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}}) \leftarrow S_\eta^{\ell-1} \times S_\eta^k$
$\mathbf{b} = \mathbf{A}_{\text{gen}} \cdot \mathbf{s}_{\text{gen}} + \mathbf{e}_{\text{gen}} \in \mathcal{R}_q^k$
$\mathbf{A} = (-2\mathbf{b} + q\mathbf{j} \mid 2\mathbf{A}_{\text{gen}} \mid 2\mathbf{Id}_k) \pmod{2q}$
$\mathbf{s} = (1, \mathbf{s}_{\text{gen}}, \mathbf{e}_{\text{gen}})$
if $\mathcal{N}(\mathbf{s}) > \gamma^2 n$ **then** restart
return $sk = (\mathbf{A}, \mathbf{s}), vk = \mathbf{A}$

Sign(sk, M):
$\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},(k+\ell)}(B))$
$\mathbf{w} \leftarrow \mathbf{A}\lfloor\mathbf{y}\rceil$
$c = H(\mathbf{w}, M) \in \mathcal{R}_2$
$\mathbf{z} = (\mathbf{z}_1, \mathbf{z}_2) = \mathbf{y} + (-1)^b c\mathbf{s}$ for $b \leftarrow U(\{0,1\})$
if $\|\mathbf{z}\|_2 \geq B'$ **then** restart
else if $\|2\mathbf{z} - \mathbf{y}\|_2 < B$ **then** restart with probability 1/2
return $\sigma = (\lfloor\mathbf{z}\rceil, c)$

Verify$(vk, M, \sigma = (\mathbf{z}, c))$:
$\tilde{\mathbf{w}} = \mathbf{A}\mathbf{z} - qc\mathbf{j} \pmod{2q}$
return $(c = H(\tilde{\mathbf{w}}, M)) \wedge \left(\|\mathbf{z}\| < B + \frac{\sqrt{n(k+\ell)}}{2}\right)$

2.4 Related Work on Memory Optimization

PQC schemes typically incur larger key/signature sizes and higher computational costs than classical public-key cryptography, making efficient implementations on resource-constrained microcontrollers (e.g., ARM Cortex-M4) essential. Prior studies on memory-optimized PQC largely exploit algorithm-specific structures

Table 3: Comparison of representative memory-optimization strategies for post-quantum cryptography on Cortex-M4.

Related Works	PQC	On-the-fly	Streaming	Recomp.	Alg. Trans.	Mem. Share	Dominant bottleneck
Bos et al. [17]	Dilithium	✓	✓		✓	✓	Poly workspaces (NTT/mul)
Hart et al. [18]	CROSS	✓	✓	✓	✓		Merkle/GGM trees
Aranha et al. [19]	FAEST	✓		✓	✓		VOLE state / VC traversal
Benadjila et al. [20]	MQOM	✓	✓	✓	✓	✓	MQ matrices / tree mat.
Our System	**SMAUG-T**	✓	✓		✓	✓	Matrix A & ciphertext buffers
	HAETAE	✓	✓	✓	✓	✓	A_1 expansion & sign/verify state

(e.g., polynomial arithmetic, tree constructions, or large linear-algebra objects) to reduce peak RAM usage. In this section, we categorize them by high-level optimization principles—on-the-fly generation, streaming of intermediates, recomputation, algorithmic transformations (NTT/Tree/DFS), and memory sharing. Table 3 provides a principle-level comparison, while the discussion below clarifies the dominant bottleneck and the concrete techniques used in each work.

Bos et al. [17] proposed various memory-saving strategies for Crystals-Dilithium[21], a NIST-standard lattice-based signature scheme. Their optimizations focus on reducing stack usage through polynomial compression, the adoption of alternative Number Theoretic Transforms (NTT), and a fallback to schoolbook multiplication for certain operations.

Hart et al. [18] addressed memory bottlenecks in CROSS[22], a code-based signature scheme, where Merkle and GGM trees consume significant resources. They proposed dynamic node calculation to avoid storing entire tree structures, utilizing an $O(\log t)$ buffer for Merkle root computation and on-the-fly generation for GGM trees. Additionally, by introducing just-in-time processing for hash inputs to eliminate unnecessary buffering, they improved feasibility of CROSS on resource-constrained platforms.

Aranha et al. [19] optimized FAEST [23], a VOLE-in-the-Head based signature, by introducing the "VOLE Black Box" concept. Instead of storing large VOLE-related matrices, their approach recomputes required segments on demand from seed trees. They also employed a logarithmic tree-walk algorithm for vector commitment, reducing the memory complexity from linear to logarithmic and improving practicality in memory-limited environments.

Benadjila and Feneuil [20] focused on minimizing the memory footprint of the MQ-based signature MQOM[24] by implementing on-the-fly matrix multiplication. This technique processes large matrices row by row instead of storing them in their entirety. Their work further incorporates bitslicing for parallel operations and a DFS-based streaming approach for GGM tree generation, demonstrating that MQOM can operate effectively within minimal SRAM constraints.

2.5 ARM Cortex-M4

The ARM Cortex-M4 [13] represents a widely adopted standard for benchmarking post-quantum cryptography in embedded environments. It is a 32-bit micro-

Table 4: Memory-intensive variables in SMAUG-T and HAETAE.

Scheme	Operation	Memory-intensive variables
SMAUG-T	KeyGen	$\mathbf{A}$, $\mathbf{s}$, $\mathbf{e}$, $\mathbf{b}$
	Encaps	$\mathbf{A}$, $\mathbf{r}$, $\mathbf{e}_1$, $(\mathbf{c}_1, c_2)$
	Decaps	$\mathbf{s}$, $(\mathbf{c}_1, c_2)$, re-encryption buffer
HAETAE	KeyGen	$\mathbf{A}$, $\mathbf{s}_1$, $\mathbf{s}_2$, $\mathbf{b}$
	Sign	$\mathbf{A}_1$, Gaussian sampling arrays, $\mathbf{y}_1$, $\mathbf{y}_2$
	Verify	$\mathbf{A}_1$, $\mathbf{z}_1$, $\mathbf{w}$, $\mathbf{h}$, $\mathbf{z}_2$

controller based on the ARMv7E-M architecture, featuring a 3-stage pipeline and supporting the Thumb-2 instruction set. Notably, the Cortex-M4 includes Digital Signal Processing (DSP) extensions—such as single-cycle multiply-accumulate (MAC) and SIMD instructions—and an optional single-precision Floating Point Unit (FPU). While most PQC schemes, including those in the `pqm4` library, primarily utilize integer arithmetic and do not leverage the FPU, the DSP capabilities allow the Cortex-M4 to be classified as a high-performance microcontroller suitable for executing complex cryptographic primitives.

The National Institute of Standards and Technology (NIST) has acknowledged the prevalence of the Cortex-M4 in academic literature, and it is widely used as a reference platform for optimization and benchmarking in the PQC community. Following this practice, the `pqm4` project adopted the Cortex-M4 to evaluate the feasibility of various PQC candidates on resource-constrained devices, reporting cycle counts, stack usage, and code size in a consistent manner. By targeting this widely deployed microcontroller class, researchers can assess a broad range of PQC schemes while establishing a baseline for performance under limited RAM and code size constraints.

3 Proposed Method

SMAUG-T and HAETAE are lattice-based cryptosystems founded upon the computational hardness of the Module Learning With Rounding (MLWR) [25] and Module Learning With Errors (MLWE) [26] problems, respectively. The primary cause of stack consumption in lattice-based cryptographic implementations is the transient storage of large-scale polynomial matrices and vectors.

In SMAUG-T, these include the public matrix $\mathbf{A}$, secret key $\mathbf{s}$, error vector $\mathbf{e}$, and public key $\mathbf{b}$ during key generation; the matrix $\mathbf{A}$, ephemeral key $\mathbf{r}$, error vector $\mathbf{e}_1$, and ciphertext components $(\mathbf{c}_1, c_2)$ during encapsulation; and the secret key $\mathbf{s}$, ciphertext components $(\mathbf{c}_1, c_2)$, and re-encryption buffer during decapsulation.

In HAETAE, key memory-intensive variables include the public matrix $\mathbf{A}$, secret keys $\mathbf{s}_1$ and $\mathbf{s}_2$, and public key $\mathbf{b}$ during key generation; the expanded matrix $\mathbf{A}_1$, Gaussian sampling arrays, and masking vectors $\mathbf{y}_1$ and $\mathbf{y}_2$ during

signature generation; and the matrix $\mathbf{A}_1$, response vector $\mathbf{z}_1$, witness $\mathbf{w}$, hint vector $\mathbf{h}$, and verification vector $\mathbf{z}_2$ during verification.

Conventional implementations of SMAUG-T and HAETAE necessitate the simultaneous residency of large-scale structures, such as the public matrix $\mathbf{A}$ and various vectors, within memory. This high memory requirement renders these implementations infeasible for deployment in resource-constrained environments, where available memory is strictly limited.

In this section, we propose stack reduction techniques to facilitate the practical deployment of SMAUG-T and HAETAE in memory-constrained environments. Our implementation builds on the second-round reference code provided in the KpqClean_ver2 project [16], and we focus on the Security Level 5 parameter sets.

3.1 Memory Optimization Strategies for Lattice-based Crypto

Definitions.

- **On-the-fly generation:** Deterministically generate coefficients/matrix elements from a public seed and indices only when needed, avoiding materialization of large public objects.
- **Streaming (JIT handling):** Eliminate staging buffers by integrating packing/unpacking and hash-input preparation directly into the computation pipeline.
- **Recomputation:** Intentionally regenerate values across phases (e.g., multi-pass sampling) when storing them would dominate peak memory.

On-the-fly generation may imply recomputation; we use *recomputation* specifically for planned regeneration in later phases as a time–memory trade-off.

To address this challenge, we identify and exploit structural properties common to lattice-based cryptographic algorithms:

- **Operational Independence:** The operational independence of matrix-vector multiplication, which enables row-by-row or column-by-column processing.
- **Deterministic Reproducibility:** The deterministic reproducibility of pseudo-random coefficients from seeds via functions such as `SHAKE128` [27].
- **Sequential Execution:** The sequential execution of major algorithmic stages with non-overlapping variable lifetimes.

Accordingly, we propose four generic optimization techniques applicable to both SMAUG-T and HAETAE, alongside algorithm-specific memory optimizations tailored to their unique structural characteristics:

- **Seed-based On-the-fly Regeneration:** Large matrices and sampling streams are regenerated on-demand from a seed rather than stored in RAM, replacing bulk storage with a minimal temporary buffer.

- **On-the-fly Streaming and JIT Packing:** Serialization and deserialization are integrated directly into the core computational loops so that each polynomial is packed into the output buffer immediately upon completion and the temporary buffer is released before the next element is processed.
- **Lifetime-based Buffer Reuse:** Variables with non-overlapping lifetimes share the same physical memory via `union` structures or explicit buffer reallocation, reducing the peak stack footprint to the size of the largest simultaneously active variable.
- **Call Graph Shaping:** Complex procedures are decomposed into sub-functions designated with the `noinline` attribute, forcing the compiler to allocate independent stack frames and preventing their local variables from overlapping within a single inflated frame.

Throughout the optimization process, the principle of constant-time execution must be strictly maintained. Detailed discussions regarding this principle are provided in Section 3.4.

Table 5 summarizes the applicability of these proposed techniques to both target algorithms. The four generic techniques are shared between SMAUG-T and HAETAE; the remaining entries correspond to algorithm-specific optimizations detailed in Sections 3.2 and 3.3.

Table 5: Applicability of Optimization Techniques

Technique	**SMAUG-T**	**HAETAE**
On-the-fly Public Matrix Generation	✓	✓
On-the-fly Serialization / Just-in-time Packing	✓	✓
Union-based Reuse	✓	✓
Function Outlining with `noinline` Attribute	✓	✓
Multi-level Function Outlining		✓
Streaming Matrix-Vector Multiplication	✓	✓
Two-Pass Hyperball Sampling		✓
Sparse Schoolbook Multiplication with Small Types	✓	
In-place Ciphertext Processing	✓	
Fused Shift-Multiply	✓	
Streaming Public Key Deserialization	✓	
Fused FO Re-encryption & Comparison	✓	

3.2 SMAUG-T5: Memory-Efficient KEM

This subsection details the application of the optimization framework presented in Section 3.1 to SMAUG-T5 ($k = 4$, $N = 256$, $q = 2{,}048$).

The primary memory bottlenecks in the reference implementation stem from three sources: the `public_key` structure, which stores the full matrix $\mathbf{A}$ in addition to the seed ρ and vector $\mathbf{b}$; sparse key and ephemeral vectors whose

Algorithm 3 On-the-fly Key Generation for SMAUG-T

Require: seed ρ, compact secret key $\mathbf{s}$ (`int8_t`), error vector $\mathbf{e}$
Ensure: serialized public key `pk_bytes`
1: **for** $i \leftarrow 0$ **to** $k-1$ **do**
2: $\quad acc \leftarrow \mathbf{0}$
3: $\quad$ **for** $j \leftarrow 0$ **to** $k-1$ **do**
4: $\quad\quad A_{i,j} \leftarrow$ `SHAKE128`$(\rho,\ i,\ j)$
5: $\quad\quad acc \leftarrow acc +$ `SparseMul`$(A_{i,j},\ \mathbf{s}[j])$
6: $\quad acc \leftarrow acc + \mathbf{e}[i]$
7: $\quad$ `pack`$(acc,$ `pk_bytes`$[i])$
8: $\quad acc \leftarrow \mathbf{0}$

coefficients are allocated as `int16_t` despite being restricted to $\{-1, 0, 1\}$; and the Toom–Cook multiplication buffer required for polynomial arithmetic.

The remainder of this subsection details the per-operation application of these techniques to key generation, encapsulation, and decapsulation, respectively.

Key Generation Key generation computes $\mathbf{b} = -\mathbf{A}^\top \cdot \mathbf{s} + \mathbf{e}$, where $\mathbf{A} \in R_q^{k \times k}$ is the public matrix, $\mathbf{s} \in S_\eta^k$ is the secret key with HWT $= 87$ per polynomial and coefficients in $\{-1, 0, 1\}$, $\mathbf{e}$ is the error vector, and $\mathbf{b}$ is the resulting public key vector. The dominant stack consumers in the reference implementation are the stored matrix $\mathbf{A}$ and the Toom–Cook multiplication buffer. We apply three techniques to eliminate both.

Generating $\mathbf{A}$ *On-The-Fly* Each element $A_{i,j}$ can be deterministically reproduced from the public seed ρ and indices (i, j) via `SHAKE128` [27]. We therefore remove the `polyvec A[MODULE_RANK]` field from the `public_key` structure entirely, generating each element on-demand within the inner loop of the matrix-vector multiplication and discarding it immediately after use. Algorithm 3 formalizes this procedure, combining on-the-fly matrix generation with on-the-fly serialization of the output vector $\mathbf{b}$.

As illustrated in Figure 1, the proposed scheme avoids materializing the full matrix $\mathbf{A}$ by processing one element at a time. In Algorithm 3, the outer loop (line 1) iterates over each row i of $\mathbf{A}$, and the accumulator acc is initialized to zero at line 2 before any element in that row is processed. The inner loop (lines 3–5) generates $A_{i,j}$ on-demand from the public seed ρ and index pair (i, j) via `SHAKE128` at line 4, immediately computes the sparse multiply-accumulate $acc \mathrel{+}=$ `SparseMul`$(A_{i,j}, \mathbf{s}[j])$ at line 5, and discards $A_{i,j}$ upon leaving line 5—at no point is more than one element of $\mathbf{A}$ resident in memory. Once all k column contributions have been accumulated, line 6 folds in the corresponding error term $\mathbf{e}[i]$, and line 7 serializes the completed row result b_i directly into `pk_bytes` via `pack`, realizing on-the-fly serialization of the public key without a separate output buffer for the full vector $\mathbf{b}$. Finally, line 8 resets acc to zero, reclaiming the single polynomial buffer before the next row begins.

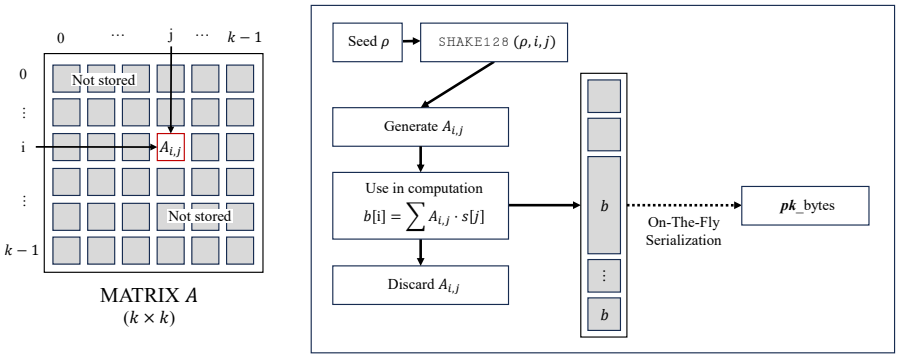

Fig. 1: On-the-fly generation of matrix $\mathbf{A}$ in SMAUG-T key generation. Each element $A_{i,j}$ is generated on-demand from seed ρ and indices (i, j) via `SHAKE128`, used in a single multiply-accumulate step $b[i] = \sum_j A_{i,j} \cdot s[j]$, and immediately discarded.

Compared to the traditional approach, which stores the entire $k \times k$ matrix $\mathbf{A}$ simultaneously, the proposed OTF approach ensures that the full matrix $\mathbf{A}$ and the full vector $\mathbf{b}$ are never simultaneously resident in memory; at most one polynomial element of each is live at any point.

Compact Secret Key Representation The coefficients of $\mathbf{s}$ are restricted to $\{-1, 0, 1\}$, yet the reference implementation allocates `int16_t` (2 bytes) per coefficient. We introduce a compact representation using `int8_t` (1 byte), halving the storage for $\mathbf{s}$ from $k \times N \times 2$ to $k \times N \times 1$ bytes, with coefficients generated directly in this compact form.

Simultaneously, we replace the Toom–Cook polynomial multiplication with a negacyclic sparse schoolbook variant (Algorithm 9) that natively accepts the compact `int8_t` secret and generates each matrix element $A_{i,j}$ on-the-fly, eliminating the Toom–Cook temporary buffer entirely.

Constant-Time Sparse Multiplication The sparse schoolbook multiplication must execute in constant time to prevent timing side-channels. A naïve sparse implementation would skip zero coefficients via a branch (e.g., `if (s[j] == 0) continue`), whose taken/not-taken pattern directly reveals the sparsity structure of the secret key. The branchless design achieving this property is presented in Section 3.4

Buffer Union for HWT Sampling Applying the memory reuse technique from Section 3.1, we merge the two buffers used during HWT sampling into a single `union` structure. The reference implementation allocates a byte array for the raw `SHAKE256` output and a `uint16_t` array for the converted values as two separate stack objects. Since the byte array is fully consumed before the integer array is populated—the little-endian conversion reads from lower addresses

before writing to the same offset—the two buffers have strictly non-overlapping lifetimes and can safely share the same physical memory via a `union`, reducing the combined allocation by half.

Furthermore, we introduce a variant of the sampling function that outputs `int8_t` coefficients directly, eliminating the separate `int16_t` intermediate buffer previously required for type conversion.

Encapsulation The encapsulation process computes a ciphertext $(\mathbf{c}_1, c_2)$ from a public key $(\rho, \mathbf{b})$ and a message μ, via $\mathbf{c}_1 = \mathbf{A}^T \cdot \mathbf{r} + \mathbf{e}_1$ and $c_2 = \mathbf{b}^T \cdot \mathbf{r} + \mathbf{e}_2 + \text{encode}(\mu)$, where $\mathbf{r}$ is an ephemeral random vector and $\mathbf{e}_1, \mathbf{e}_2$ are error terms. The dominant stack consumers are the deserialized public key structure (holding both the full matrix $\mathbf{A}$ and vector $\mathbf{b}$), the ephemeral key $\mathbf{r}$ stored as `int16_t`, and the intermediate ciphertext staging buffer.

Sparse Ephemeral Key Multiplication The ephemeral key $\mathbf{r}$ is sampled from the `sp_cbd2` distribution with coefficients in $\{-1, 0, 1\}$ and an expected Hamming weight of approximately 96, sharing the same sparse structure as $\mathbf{s}$ (HWT = 87). We therefore apply the same pair of optimizations introduced in Section 3.2: the compact `int8_t` representation and the sparse on-the-fly schoolbook multiplication (Algorithm 9).

For $\mathbf{c}_1 = \mathbf{A}^T \cdot \mathbf{r}$, each transposed matrix element $A_{j,i}$ is generated on-demand via `SHAKE128` and immediately multiplied by $\mathbf{r}[j]$, accumulating into $\mathbf{c}_1[i]$ without storing the full matrix or the full vector $\mathbf{r}$ simultaneously. The inner product $\mathbf{b}^T \cdot \mathbf{r}$ for c_2 reuses the same routine, as $\mathbf{r}$ shares the compact type and sparse structure of $\mathbf{s}$.

Streaming Ciphertext Serialization Applying the on-the-fly serialization technique from Section 3.1, we eliminate the intermediate ciphertext staging structure entirely. Instead of computing all of $\mathbf{c}_1$ into a dedicated buffer and then serializing, each element is computed, immediately packed into the output byte array, and the buffer is reused for the next element. Algorithm 4 shows the resulting streaming encapsulation.

The ciphertext staging structure is never allocated; the output byte array is populated incrementally and at most one polynomial is live at any time.

Fused Shift-Multiply The inner product routine for $\mathbf{b}^T \cdot \mathbf{r}$ internally stores a right-shifted copy of each input coefficient into a temporary polynomial before forwarding it to the sparse multiplication kernel. Since the shift is element-wise and depends only on public parameters, it can be fused directly into the multiplication loop itself. We introduce a shift-fused variant `SparseMulShift`($\mathbf{a}$, $\mathbf{s}$, *shift*) that computes $(a[k] \gg \mathit{shift}) \cdot s[j]$ inline, eliminating the temporary buffer without changing the total number of shift operations.

Streaming Public Key Deserialization After the preceding optimizations, the deserialized public key structure —holding the seed ρ and the full vector $\mathbf{b}$—

Algorithm 4 Streaming Ciphertext Serialization in Encapsulation

Require: seed ρ, compact ephemeral key $\mathbf{r}$ (`int8_t`), serialized $\mathbf{b}$, message μ
Ensure: serialized ciphertext `ct_bytes`

```
tmp ← 0                                          ▷ single reusable polynomial buffer
for i ← 0 to k − 1 do
    tmp ← 0
    for j ← 0 to k − 1 do
        A_{j,i} ← SHAKE128(ρ, j, i)
        tmp += SparseMul(A_{j,i}, r[j])
    tmp += e_1[i]
    Rp_to_bytes(ct_bytes, tmp)
tmp ← InnerProduct(b, r, μ)
Rp2_to_bytes(ct_bytes, tmp)
```

remains the largest single allocation in `indcpa_enc`. The seed ρ occupies the first 32 bytes of the serialized public key and can be referenced via a direct pointer rather than a copy, while individual elements b_i are only needed one at a time during the inner product $\mathbf{b}^T \cdot \mathbf{r}$. We exploit this by deserializing each b_i on-the-fly within the accumulation loop via `SparseMulShift` and immediately overwriting the buffer upon completion, eliminating the deserialized public key structure entirely.

The deserialized public key structure is never allocated; the full vector $\mathbf{b}$ is never simultaneously resident in memory.

Decapsulation The decapsulation of SMAUG-T recovers the shared secret from a ciphertext $(\mathbf{c}_1, c_2)$ using the secret key $\mathbf{s}$, via $\mu' = \text{decode}(c_2 - \mathbf{s}^T \cdot \mathbf{c}_1)$, followed by a Fujisaki–Okamoto (FO) [28] re-encryption comparison to ensure ciphertext integrity. The dominant stack consumers are the deserialized ciphertext structure, redundant temporary copies of its components, and the full re-encryption buffer materialized for the FO comparison.

In-place Ciphertext Processing Applying the memory reuse technique from Section 3.1, we eliminate redundant temporary copies. The reference `indcpa_dec` deserializes the ciphertext into a structure `ctxt_tmp`, then copies its components into separate temporary variables—a vector for $\mathbf{c}_1$ and a polynomial for c_2—solely to apply coefficient shifts before the inner product. A lifetime analysis reveals that `ctxt_tmp` is never read after the copy, so the shifts can be applied in-place directly to its fields, eliminating both temporaries—reducing three sequential single-use allocations to one.

Compact Secret Key with Sparse Multiplication As in key generation and encapsulation, the secret key $\mathbf{s}$ is loaded in the compact `int8_t` representation, and the inner product $\mathbf{s}^T \cdot \mathbf{c}_1$ is computed using the sparse schoolbook variant (Algorithm 9). This eliminates both the `polyvec` temporary buffer and the Toom–Cook internal allocation from `indcpa_dec`.

Algorithm 5 Streaming Decryption in `indcpa_dec`

Require: ciphertext bytes `ct_bytes`, compact secret key $\mathbf{s}$ (`int8_t`)
Ensure: message μ'

$acc \leftarrow \mathbf{0}$
$elem \leftarrow \mathbf{0}$
for $i \leftarrow 0$ **to** $k-1$ **do**
    $elem \leftarrow$ `bytes_to_poly`(`ct_bytes`, i)
    $acc \mathrel{+}= $ `SparseMul`($elem$, $\mathbf{s}[i]$)
    overwrite $elem$
`left_shift`(acc)
$elem \leftarrow$ `bytes_to_poly`(c_2 portion of `ct_bytes`)
$\mu' \leftarrow$ decode($elem - acc$)

Streaming Decryption After the preceding optimizations, the deserialized ciphertext structure `ctxt_tmp` remains the largest allocation in `indcpa_dec`. We observe that the left-shift applied to all elements of $\mathbf{c}_1$ and the compensating right-shift applied inside the sparse inner product cancel each other; both can therefore be dropped, allowing decryption to operate directly on the unshifted, byte-serialized ciphertext. Algorithm 5 shows the resulting streaming procedure.

The ciphertext structure is replaced by two single-polynomial buffers; the full vector $\mathbf{c}_1$ is never simultaneously resident in memory.

Algorithm 6 Streaming FO Re-encryption and Constant-Time Comparison (`indcpa_enc_cmp`)

Require: recovered message μ', received ciphertext `ct_bytes`, public key $(\rho, \mathbf{b})$
Ensure: constant-time equality bit $result \in \{0, 1\}$

$rc \leftarrow 0$
$chunk \leftarrow \mathbf{0}$
$offset \leftarrow 0$
for $i \leftarrow 0$ **to** $k-1$ **do**
    Recompute $\mathbf{c}_1[i]$ into $chunk$
    `Rp_to_bytes`($chunk$)
    for $k \leftarrow 0$ **to** `CTPOLY1_BYTES`-1 **do**
        $rc \mid= chunk[k] \oplus$ `ct_bytes`$[offset + k]$
    $offset \mathrel{+}= $ `CTPOLY1_BYTES`
Recompute c_2 into $chunk$; `Rp2_to_bytes`($chunk$)
for $k \leftarrow 0$ **to** `CTPOLY2_BYTES`-1 **do**
    $rc \mid= chunk[k] \oplus$ `ct_bytes`$[offset + k]$
return $\neg\big((\texttt{uint64_t}(rc)) \gg 63\big)$

Streaming Re-encryption Comparison The FO transform requires re-encrypting the recovered message μ' and comparing the result byte-by-byte against the received ciphertext. The reference implementation allocates a full re-encryption

buffer, materializes the ciphertext in its entirety, and then performs the comparison in a single call. We introduce `indcpa_enc_cmp`, a fused function that performs re-encryption and comparison simultaneously in a streaming fashion, as shown in Algorithm 6.

Each component of the re-encrypted ciphertext is serialized into the chunk buffer and immediately compared against the received ciphertext via bitwise XOR accumulation; the full re-encryption buffer is never allocated, and the comparison never short-circuits regardless of where a mismatch occurs.

Callee Separation Applying the technique from Section 3.1, we prevent temporal overlap of stack frames. After all preceding optimizations, `crypto_kem_dec` invokes `indcpa_dec` and `indcpa_enc_cmp` sequentially: `indcpa_dec` completes and its stack frame is deallocated before `indcpa_enc_cmp` begins. We therefore apply `__attribute__((noinline))` to both callees, ensuring the compiler allocates their stack frames independently rather than merging them during inlining. The worst-case peak for `crypto_kem_dec` is thereby

$$frame(\texttt{crypto_kem_dec}) + \max\Big\{frame(\texttt{indcpa_dec}),\; frame(\texttt{indcpa_enc_cmp}) + frame(\texttt{genAx_element})\Big\},$$

rather than the sum of all callee frames.

3.3 HAETAE5: Memory-Efficient Digital Signature

This section provides a comprehensive overview of the memory management and optimization methods integrated into each phase of the HAETAE5 digital signature scheme.

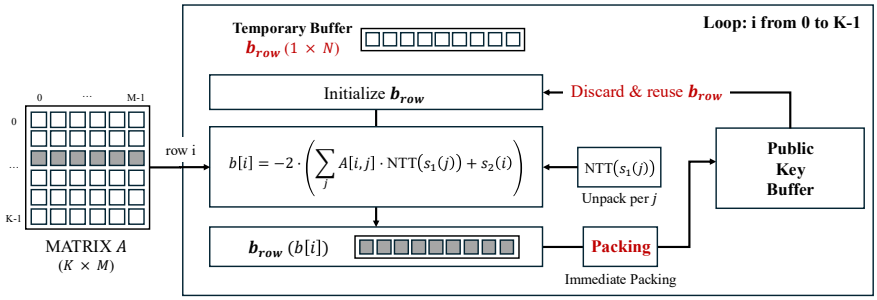

Fig. 2: The key generation process for $b = -2(As_1 + s_2)$ in terms of memory optimization. By generating vectors row by row and reusing buffers, the memory footprint is reduced. (In HAETAE-5, K=4 and M=6 and N=256.)

Key Generation The public vector $\mathbf{b} = -2(\mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2)$ is produced one row at a time. No full matrix $\mathbf{A}$ or full vectors $\mathbf{s}_1$ and $\mathbf{s}_2$ are kept in memory; instead, only the current row of $\mathbf{A}$ (or its contribution) and a single 1 KB buffer for the current row of $\mathbf{b}$ are utilized. Each row is packed into the public key format as soon as it is computed(See Fig 2).

Signature Generation(Alg. 7) Algorithm 7 illustrates the memory-efficient signature generation process. This stage is designed to avoid keeping the complete sample vector, the full matrix $\mathbf{A}_1$, or the entire response vectors $\mathbf{z}_1$ and $\mathbf{z}_2$ in memory. The specific strategies are as follows:

Algorithm 7 Memory-Efficient HAETAE Signature Generation

Input: Secret key sk, message M
Output: Signature σ (bit string)

Pass 1: (Norm Calculation)
$temp, sqsum \leftarrow 0$
for each block ℓ **do**
  $samples \leftarrow \text{Generate}(\ell)$
  $sqsum \leftarrow sqsum + \|samples\|^2$
  ▷ **Memory Opt:** Avoid storing full $\mathbf{y}$ in this pass
Store $norm_factor$ derived from $sqsum$

Pass 2: (Matrix-Vector Multiplication)
Regenerate blocks with same seed → scale by $norm_factor \rightarrow \mathbf{y}_1, \mathbf{y}_2$
Check $\|\mathbf{y}\|^2$ condition ▷ Rejection sampling

for $i = 0$ **to** $K - 1$ **do** ▷ Row-by-row processing
  $Ay[i] \leftarrow 0$
  for $j = 0$ **to** $L - 1$ **do**
    $\mathbf{A}_{1,ij} \leftarrow \text{UnpackOrGen}(\rho', i, j)$
    $Ay[i] \leftarrow Ay[i] + \mathbf{A}_{1,ij} \cdot \text{round}(\mathbf{y}_1)[j]$
    discard $\mathbf{A}_{1,ij}$ ▷ **Memory Opt:** Immediate buffer release
  highbits$(Ay[i])$ → pack → absorb into H

for $j = 0$ **to** $L - 1$ **do** ▷ On-the-fly $\mathbf{z}$ generation
  $\mathbf{y}_{1,j} \leftarrow \text{Regenerate}(seed, norm_factor, j)$
  $\mathbf{z}_{1,j} \leftarrow \mathbf{y}_{1,j} + \text{NTT}^{-1}(\text{NTT}(\mathbf{c}) \cdot \text{NTT}(\mathbf{s}_{1,j}))$
  Decompose and rANS encode $\mathbf{z}_{1,j} \rightarrow sig$
  discard $\mathbf{z}_{1,j}$ ▷ **Memory Opt:** Discard after encoding

Finish:
for $i = 0$ **to** $K - 1$ **do**
  Encode $\mathbf{h}[i]$ row-wise and absorb into H
return $\sigma \leftarrow (c, sig, \dots)$

Hyperball Sampling Lines 3–9 (Pass 1) calculate the squared sum for the norm calculation without full storage, while lines 10–24 (Pass 2) regenerate the samples using the stored $norm_factor$ (lines 10–12). This two-pass strategy replaces conventional single-pass sampling to eliminate the need for a multi-KB sample buffer.

$\mathbf{A}_1 \cdot \mathbf{y}$ *Operation* Lines 13–19 compute the matrix-vector product row-by-row. Instead of keeping the full matrix $\mathbf{A}_1$ in memory, line 16 generates each element on-the-fly, and line 18 discards the buffer immediately after accumulation to minimize memory occupancy.

Response Vector $\mathbf{z}$ *Generation* Line 21 regenerates $\mathbf{y}_{1,j}$ only at the specific moment it is required for computation. The resulting $\mathbf{z}_{1,j}$ is computed in the NTT domain (line 22), encoded, and then discarded immediately at line 24 to ensure the peak memory footprint remains minimal.

Signature Verification(Alg. 8) Algorithm 8 presents the memory-efficient pipeline for signature verification. The process maximizes memory efficiency by organically connecting three data flows:

Algorithm 8 Memory-Efficient HAETAE Verification

1: **Input:** Public key pk, message M, signature σ
2: **Output:** Accept or Reject (e.g. 0 / -1)

3: **for** $j = 0$ **to** $L - 1$ **do**
4: $\quad \mathbf{z}_{1,j} \leftarrow \text{rANS_decode}(sig)$ ▷ Streaming decode
5: $\quad$ **if** $j = 0$ **then**
6: $\quad\quad \mathbf{Ay} \leftarrow \text{Unpack}(\mathbf{b}) \cdot \mathbf{z}_{1,j}$
7: $\quad$ **else**
8: $\quad\quad \mathbf{Ay} \leftarrow \mathbf{Ay} + \text{FusedMAC}(\rho', j-1, \mathbf{z}_{1,j})$ ▷ Column-by-column
9: $\quad$ **discard** $\mathbf{z}_{1,j}$ ▷ **Memory Opt:** Immediate buffer release
10: **for** $i = 0$ **to** $K - 1$ **do**
11: $\quad \mathbf{h}[i] \leftarrow$ decode hint into work buffer ▷ Row-wise decode
12: $\quad w_i \leftarrow \text{highbits}(Ay[i]) + \mathbf{h}[i]$
13: $\quad$ absorb $\text{highbits}(w_i)$ into H
14: $\quad w' \leftarrow$ restore from 32-byte bitmap
15: $\quad \mathbf{z}_2[i] \leftarrow (w_i \cdot \alpha - Ay[i] + w') \cdot 2^{-1} \pmod q$ ▷ **Memory Opt:** In-place in $\mathbf{Ay}[i]$
16: $c' \leftarrow H.\text{final}()$
17: **return** $(c == c')$ **in constant time**

$\mathbf{z}_1$ *Decoding and Discarding* In line 4, $\mathbf{z}_{1,j}$ is decoded on-the-fly from the rANS stream. To minimize memory occupancy, each component is discarded immediately at line 9 after its contribution to the matrix-vector product is accumulated.

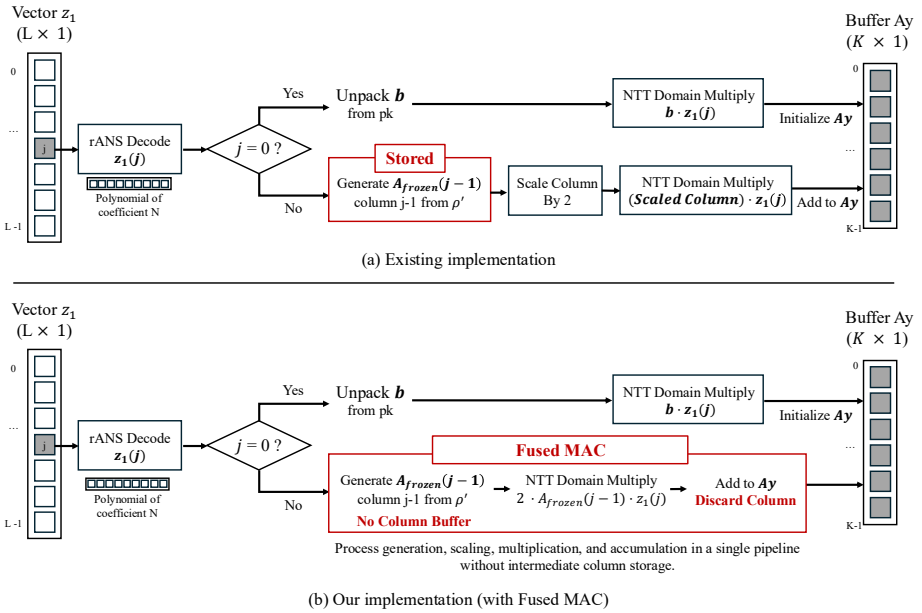

Fig. 3: **(a) Existing Implementation[16]:** Intermediate column buffers were required to store the results of sequential steps, including column generation, scaling, multiplication, and addition. **(b) Our implementation:** The entire process is executed as a single Fused MAC step without intermediate buffers, thereby reducing memory usage. (In HAETAE-5, L=7 and K=4 and N=256.)

Column-wise **Ay** *Operation (Fused MAC)* Lines 3–9 perform the $\mathbf{Ay} = \mathbf{A}_1\mathbf{z}_1$ operation column-by-column. By applying the Fused MAC(see Fig 3) technique in line 8, the implementation integrates generation and accumulation, effectively eliminating the intermediate column buffer.

Row-wise Post-processing and In-place Operations For each row (lines 10–15), the hint $\mathbf{h}[i]$ is decoded into a work buffer at line 11. The implementation forms $w_i = \text{highbits}(Ay[i]) + \mathbf{h}[i]$ (line 12), absorbs the high bits into the hash state H (line 13), and restores w' from the bitmap (line 14). Crucially, $\mathbf{z}_2[i]$ is computed and written in-place into the existing **Ay** buffer at line 15, significantly reducing the total stack memory by reusing storage for the final response components.

Challenge Comparison The final step of verification at Line 17 is performed in a constant-time manner using XOR operations over 32 bytes, OR-accumulation, and a single branch to ensure security.

Lifetime-based Buffer Reuse in HAETAE We map large temporary arrays with disjoint lifetimes to a limited set of shared stack regions, a design

summarized in Table 6. In KeyGen, one buffer is shared by the NTT-domain secret (e.g., NTT($\mathbf{s}_{1,j}$)) and the current row of the public vector ($\mathbf{b}_{\text{row}}$). In Sign, the Gaussian sampler, matrix-vector product, frozen columns, and $\mathbf{c} \cdot \mathbf{s}$ share a unified workspace, and the Hyperball resampling buffer is reused as a general polynomial working buffer. In Verify, the execution order of $\mathbf{z}_1$, $\mathbf{w}$, highbits, $\mathbf{h}$, $\mathbf{z}_2$, and $\mathbf{w}'$ is scheduled so that their lifetimes do not overlap and at most one occupies the physical buffer at any time.

Table 6: Detailed mapping of memory reuse strategies in HAETAE.

Phase	**Sub-step**	**Memory Sharing & Reuse Mapping (Sequential)**
KeyGen	Key Generation	$\hat{s}_1$ (Secret NTT[29]) $\leftrightarrow$ b_row (Public-vec Int.)
Sign	Rejection Loop	Gaussian $\leftrightarrow$ $\mathbf{Av}$ (Matrix-vec) $\leftrightarrow$ col_{frozen} $\leftrightarrow$ $c \cdot s$
	h Encoding	Resampling Buffer $\leftrightarrow$ Working Poly Buffer
Verify	Phase I	z_1 (Restoration) $\leftrightarrow$ w (Computation)
	Phase II	$(lowbits, c)$ (Restoration) $\leftrightarrow$ $highbits$ (Matrix Result)
	Phase III	h (Encoding) $\leftrightarrow$ z_2 (Restoration)
	Phase IV	Poly-read Buffer $\leftrightarrow$ w' (Final Verification)

3.4 Constant-time Implementation

Memory optimization techniques that alter algorithmic structure or control flow may inadvertently introduce timing side-channels. In this section, we analyze the constant-time properties of our optimized implementation and describe the measures taken to ensure that no secret-dependent timing variations are introduced.

Sparse Schoolbook Multiplication While Schoolbook multiplication is preferred in memory-constrained environments due to its smaller memory footprint, all such operations must be executed in constant time to prevent timing side-channel attacks. Note that HAETAE achieved its memory savings without relying on Schoolbook multiplication; we therefore exclusively adopted the NTT-based approach for HAETAE, which is inherently more advantageous for ensuring constant-time execution.

Sparse Schoolbook Multiplication in SMAUG-T As HAETAE achieves its memory savings without relying on Schoolbook multiplication and exclusively adopts the NTT-based approach, the following branchless design applies specifically to SMAUG-T.

The most significant algorithmic change in our implementation is the replacement of Toom-Cook 4-way multiplication with negacyclic sparse schoolbook multiplication (Section 3.2). While the Toom–Cook 4-way multiplication used in the

Algorithm 9 Constant-Time Negacyclic Sparse Schoolbook Multiplication

Require: polynomial $\mathbf{a} \in R_q$, sparse polynomial $\mathbf{s}$ (`int8_t`, coefficients in $\{-1, 0, 1\}$)
Ensure: $\mathbf{r} = \mathbf{a} \circledast \mathbf{s}$ (negacyclic product)

```
r ← 0
for j ← 0 to N − 1 do
    s_j ← (int16_t) s[j]
    for k ← 0 to N − 1 − j do
        r[k + j] += s_j · a[k]
    for k ← N − j to N − 1 do
        r[k + j − N] −= s_j · a[k]
```

reference implementation [16] follows a fixed execution path regardless of input values, a naïve sparse multiplication that skips zero coefficients—e.g., `if (s[j] == 0) continue`—introduces a branch whose taken/not-taken pattern directly reveals the Hamming weight and sparsity structure of the secret key $\mathbf{s}$. Our implementation eliminates all secret-dependent branches. For each coefficient index j, the product $s_j \cdot a_k$ is computed unconditionally for all $k \in \{0, \ldots, N-1\}$. When $s_j = 0$,this yields a multiplication by zero that follows the identical instruction sequence as the non-zero case, as formalized in Algorithm 9.

This branchless design ensures the following constant-time properties:

- **Uniform control flow.** The loop bounds depend only on the public index j and the public parameter N, never on the value of s_j. No conditional branches are predicated on secret data.
- **Uniform memory access pattern.** Every element `a[k]` is accessed for every j regardless of s_j, ensuring that the cache access pattern is independent of the secret key.
- **Fixed operation count.** The total cost is $N^2 = 65{,}536$ multiply-accumulate operations per polynomial pair, invariant of the Hamming weight.

The same branchless multiplication is applied uniformly to the secret key $\mathbf{s}$ (HWT = 87) in key generation and decapsulation, and to the ephemeral key $\mathbf{r}$ (expected HW $\approx$ 96) in encapsulation. The shift-fused variant `poly_mul_acc_sparse_shift()` (Section 3.2) preserves the identical control flow, differing only in the addition of a public shift parameter applied to each `a[k]` operand.

Constant-time Comparison All ciphertext and challenge comparisons are implemented in a branchless, constant-time manner to prevent timing leakage. To eliminate this vulnerability, our implementation adopts a branchless, constant-time comparison method that ensures a fixed execution path regardless of the underlying data values.

Constant-time FO Comparison in SMAUG-T The FO re-encryption [28] comparison in `indcpa_enc_cmp()` (Section 3.2) compares the re-encrypted ciphertext against the received ciphertext in a streaming fashion. Each byte comparison is performed via bitwise XOR accumulation as formalized in Algorithm 6.

The accumulator `rc` is processed through all chunks—including all k elements of $\mathbf{c}_1$ and the scalar c_2—before the final result is derived as `(-(uint64_t)rc) >> 63`. This ensures that: (i) the comparison does not short-circuit on the first mismatch, (ii) all bytes are compared regardless of prior differences, and (iii) the final 0/1 result is derived through arithmetic rather than branching.

Constant-time challenge comparison in HAETAE Conventional early-exit (short-circuiting) comparison of the challenge bytes (c and c') is vulnerable to timing side-channel attacks, as the execution time varies depending on the mismatch position. To mitigate this risk, we implement a constant-time comparison that performs bitwise XOR across all 32 bytes and accumulates the results using bitwise OR into a single status flag. By processing the entire data segment regardless of where the mismatch occurs and executing only one final branch at the end, this method effectively eliminates timing leakage and ensures the security of the verification stage.

4 Evaluation

4.1 Experimental Setup

Experiments were conducted on two ARM Cortex-M4 platforms: STM32F4 Discovery (STM32F407VGT6) and Nucleo-L4R5ZI (STM32L4R5ZI). Both microcontrollers implement the ARMv7E-M architecture and include a single-precision floating-point unit (FPv4-SP-D16).

STM32F4 Discovery provides 1 MB Flash, 128 KB SRAM (112 KB main SRAM + 16 KB SRAM2), and 64 KB CCM RAM. Nucleo-L4R5ZI provides 2 MB Flash and 640 KB SRAM. Table 7 summarizes the specifications of the target Cortex-M4 platforms used in our experiments.

Clock configurations followed the default pqm4 settings. Speed measurements were performed using `CLOCK_BENCHMARK` (24 MHz on STM32F4 Discovery, 20 MHz on Nucleo-L4R5ZI), which eliminates flash wait states (0WS) to ensure stable cycle measurements.

Stack measurements were conducted under `CLOCK_FAST` (168 MHz on STM32F4 Discovery, 120 MHz on Nucleo-L4R5ZI). The clock frequency does not affect the measured stack depth, as stack usage depends solely on control flow and memory allocation.

4.2 Measurement Methodology

Performance evaluation was conducted using the `pqm4` benchmarking framework for ARM Cortex-M4 platforms. Throughout Section 4, **Ref** denotes the unoptimized `clean` implementation from `KpqClean_ver2` [16], and **Opt** denotes our memory-optimized implementation.

Stack usage was measured at runtime using the `pqm4` stack watermark method. Prior to each operation, the unused stack region was filled with a sentinel pattern

Table 7: Experimental Platform Specifications

	STM32F4 Discovery	**Nucleo-L4R5ZI**
MCU	STM32F407VGT6	STM32L4R5ZI
Core	ARM Cortex-M4F	ARM Cortex-M4F
ISA	ARMv7E-M	ARMv7E-M
FPU	FPv4-SP-D16	FPv4-SP-D16
Flash	1 MB	2 MB
SRAM	128 KB + 64 KB CCM	640 KB
Clock (stack)	168 MHz (`CLOCK_FAST`)	120 MHz (`CLOCK_FAST`)
Clock (speed)	24 MHz (`CLOCK_BENCHMARK`)	20 MHz (`CLOCK_BENCHMARK`)

(`0xDEADBEEF`) via `hal_spraystack()`. After execution, `hal_checkstack()` was used to count the overwritten bytes, yielding the worst-case stack depth. Stack measurements were performed under `CLOCK_FAST`.

Code size was obtained from the final linked ELF binary using `arm-none-eabi-size`.

Cycle counts were collected using the Cortex-M4 DWT cycle counter as integrated in `pqm4` under `CLOCK_BENCHMARK`. For SMAUG-T5, each operation was executed 10,000 times. For HAETAE5, 1,000 iterations were used due to its significantly higher signing cost. Median and mean values are reported.

Compiler settings and fairness. We intentionally use different compiler optimization levels depending on the metric of interest. For memory-oriented measurements (stack usage and code size), we compile with `-Os` to reflect realistic deployment builds on constrained Cortex-M4 devices, where feasibility is primarily limited by Flash and RAM pressure. For performance measurements (cycle counts), we compile with `-O3` following the `pqm4` benchmarking convention, which reports best-effort throughput under aggressive optimization. Importantly, we do not mix optimization levels within a comparison: stack and code-size results are always compared under `-Os` (Ref vs. Opt), and cycle results are always compared under `-O3` (Ref vs. Opt). This separation prevents conflating compiler effects with our memory-optimization transformations and ensures that all reported improvements and overheads are evaluated fairly and reproducibly within each metric.

4.3 SMAUG-T5

Memory usage Stack usage and code size are summarized in Table 8.

Across both Cortex-M4 devices, the optimized implementation reduces stack usage by 73–83% for all operations. The reduction ratio remains consistent across platforms, indicating that the memory optimization strategy is largely architecture-independent.

The code size increases by approximately 32.5% (`.text` section), reflecting additional recomputation logic and control flow introduced to eliminate large temporary buffers.

Table 8: Stack usage and code size of SMAUG-T5 on Cortex-M4 platforms (bytes, identical across STM32F4 and Nucleo-L4R5ZI)

Operation	Ref	Opt	Δ (%)
KeyGen	20,148	5,328	-73.6%
Encaps	22,820	3,832	-83.2%
Decaps	24,300	4,240	-82.5%
Code size (.text)	6,438	8,534	+32.5%

Cycle Performance Cycle results are summarized in Table 9.

The optimized version incurs an overhead of 2.4–2.9× across all operations. The overhead primarily stems from recomputation of intermediate values that were previously stored in memory.

Importantly, the overhead factor is stable across both platforms, suggesting that the performance trade-off is dominated by algorithmic restructuring rather than microarchitectural effects.

Table 9: Cycle performance of SMAUG-T5 on Cortex-M4 platforms (-O3, 10,000 runs, median cycles)

	STM32F4		Nucleo-L4R5ZI		Ratio (Opt/Ref)	
Operation	Ref	Opt	Ref	Opt	STM32F4	L4R5ZI
KeyGen	5,084,424	12,402,313	5,438,870	13,645,867	2.44×	2.51×
Encaps	5,500,719	15,422,863	5,856,955	16,890,088	2.80×	2.88×
Decaps	6,623,798	18,483,274	7,054,764	20,244,852	2.79×	2.87×

4.4 HAETAE5

Memory usage Memory results are summarized in Table 10.

The optimized implementation reduces stack usage by approximately 90% across all operations. In particular, the signing procedure decreases from 91 KB to 8 KB, enabling execution on devices with severely constrained SRAM resources.

Code size increases by 37–39%, reflecting additional recomputation and buffer reuse logic introduced to eliminate large temporary buffers.

Cycle Performance Cycle measurements are summarized in Table 11.

Key generation and verification exhibit negligible overhead (0.98× and 1.04×, respectively), whereas signing incurs a 4.18× overhead.

Table 10: Stack usage and code size of HAETAE5 on Cortex-M4 platforms (bytes, identical across STM32F4 Discovery and Nucleo-L4R5ZI, `-Os`)

Operation	Ref	Opt	Δ (%)
KeyGen	52,712	5,212	−90.1%
Sign	91,176	8,092	−91.1%
Verify	62,196	6,220	−90.0%
Code size (.text)	19,086	26,494	+38.8%
Total size	20,126	27,534	+36.8%

Table 11: Cycle performance of HAETAE5 on Cortex-M4 platforms (-O3, 1,000 runs, median cycles)

	STM32F4		Nucleo-L4R5ZI		Ratio (Opt/Ref)	
Operation	Ref	Opt	Ref	Opt	STM32F4	L4R5ZI
KeyGen	10,734,588	10,538,704	12,711,748	15,327,648	0.98×	1.21×
Sign	49,609,518	207,306,694	73,425,375	300,880,635	4.18×	4.10×
Verify	2,411,460	2,497,256	4,082,575	4,187,497	1.04×	1.03×

The significant overhead in signing stems from recomputation introduced by memory-saving restructuring, including on-the-fly matrix generation/unpacking, repeated hash absorption, and the two-pass sampling strategy used during signing. This result highlights the inherent time–memory trade-off in memory-constrained post-quantum signature implementations.

5 Conclusion

We proposed optimized memory management techniques for SMAUG-T and HAETAE, which are prominent candidates in KpqC. While baseline implementations can run on Cortex-M4 platforms, peak stack usage remains a key bottleneck in highly constrained environments—particularly for HAETAE signing—and fine-grained optimization strategies tailored to strict SRAM thresholds remain underdeveloped. In this paper, we propose a memory-efficient implementation strategy and demonstrate that stable execution and signature generation are attainable under severe stack-memory constraints (e.g., SMAUG-T5 decapsulation peak stack reduced from 24,300 B to 4,240 B, and HAETAE5 signing peak stack reduced from 91,176 B to 8,092 B).

Furthermore, all optimizations were designed under constant-time implementation principles to avoid introducing secret-dependent control flow and early-exit behavior, mitigating timing side-channel risks.

Regarding performance, we analyzed the trade-off between memory efficiency and computational speed. For SMAUG-T, we achieved effective memory opti-

mization with a performance slowdown of approximately 2.4× to 2.9× compared to the reference implementation. In contrast, HAETAE exhibited a more pronounced performance penalty, primarily attributed to the computational overhead of on-the-fly matrix generation/unpacking and the two-pass sampling strategy used during signing, which increases recomputation in the presence of rejection sampling.

Furthermore, the methodology presented in this paper can serve as a technical guideline for memory optimization in other KpqC implementations. By significantly reducing peak stack usage, our approach paves the way for practical adoption of KpqC in memory-constrained embedded devices and hardware security modules.

References

1. E. Milanov, "The RSA algorithm," *RSA laboratories*, vol. 1, no. 11, 2009.
2. D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)," *International Journal of Information Security*, vol. 1, no. 1, pp. 33–63, 2001.
3. L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pp. 212–219, 1996.
4. P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th annual symposium on foundations of computer science*, pp. 124–134, Ieee, 1994.
5. N. I. of Standards and T. (NIST), "Report on the post-quantum cryptography (PQC) transition strategy," Tech. Rep. NIST IR 8547 (Initial Public Draft), U.S. Department of Commerce, 2024.
6. C. Tjhai, M. Tomlinson, G. Bartlett, S. Fluhrer, D. V. Geest, and O. Garcia-Vidal, "Multiple key exchanges in the internet key exchange protocol version 2 (IKEv2)." RFC 9370, 2023.
7. IETF, "Post-quantum use in protocols (pquip) working group." https://datatracker.ietf.org/wg/pquip/about/, 2026. Accessed: 2026-02-24.
8. S. Kim, J. Ha, and J. Lee, "AIM: symmetric primitive for shorter signatures with stronger security," in *ACM CCS 2023: ACM Conference on Computer and Communications Security*, ACM (Association for Computing Machinery), 2023.
9. J. H. Cheon, H. Choe, J. Devevey, T. Güneysu, D. Hong, M. Kang, T. Kim, J. Ko, M. Krausz, G. Land, M. Möller, J. Shin, D. Stehlé, and M. Yi, "HAETAE: shorter lattice-based fiat-shamir signatures," tech. rep., KpqC, Nov 2023. Version 2.0.
10. J. Kim and J. H. Park, "Ntru+: compact construction of ntru using simple encoding method," *IEEE Transactions on Information Forensics and Security*, 2023.
11. J. H. Cheon, H. Choe, D. Hong, J. Hong, H. Seong, J. Shin, and M. Yi, "SMAUG: the key exchange algorithm based on Module-LWE and Module-LWR," tech. rep., Seoul National University and CryptoLab Inc., Oct 2023. Version 2.0.
12. "KpqC competition." https://kpqc.or.kr/competition.html, 2025. Accessed on 4 Mar 2026.
13. Arm Limited, *Cortex-M4 Technical Reference Manual*. Arm Limited, r0p1 ed., 2020. Available at https://developer.arm.com/documentation/ddi0439/latest/.

14. M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking NIST PQC on ARM Cortex-M4," in *Second PQC Standardization Conference*, July 2019. Available at https://github.com/mupq/pqm4.
15. S.-C. Seo, Y.-B. Kim, and Y.-R. Choi, "kpqm4: Testing and benchmarking KpqC algorithms on ARM Cortex-M4," 2024. Available at https://github.com/KMU-CSE/kpqm4. Accessed: 2026-02-28.
16. kpqc-cryptocraft, "KpqClean_ver2: Benchmark on Korean Post Quantum Cryptography version 2." https://github.com/kpqc-cryptocraft/KpqClean_ver2, 2025. Accessed: 2026-02-20.
17. J. W. Bos, J. Renes, and A. Sprenkels, "Dilithium for memory constrained devices," in *Smart Card Research and Advanced Applications (CARDIS)*, Springer, 2022.
18. H. Hart, P. Mondal, S. Kundu, S. Adhikary, A. Karmakar, and C. Li, "Lightening CROSS: Memory optimized implementations of CROSS," *Cryptology ePrint Archive*, 2024. Paper 2024/1929.
19. D. F. Aranha, J. Degn, J. Eilath, K. Nielsen, and P. Scholl, "Faest for memory-constrained devices with side-channel protections," *Cryptology ePrint Archive*, 2025. Paper 2025/1261.
20. R. Benadjila and T. Feneuil, "Breaking the myth of mpcith inefficiency: Optimizing mqom for embedded platforms," *Cryptology ePrint Archive*, 2025. Paper 2025/111.
21. S. Bai, L. Ducas, E. Kiltz, T. Lepoint, V. Lyubashevsky, P. Schwabe, G. Seiler, and D. Stehlé, "CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Round 3)." Available at https://pq-crystals.org/dilithium/, 2020. NIST Post-Quantum Cryptography Standardization Process.
22. M. Baldi, A. Barenghi, M. Battagliola, S. Bitzer, M. Gianvecchio, P. Karl, F. Manganiello, A. Pavoni, G. Pelosi, P. Santini, J. Schupp, E. Signorini, F. Slaughter, A. Wachter-Zeh, and V. Weger, *CROSS: Codes and Restricted Objects Signature Scheme*. NIST Post-Quantum Cryptography Standardization Process, January 2025. Algorithm Specifications and Supporting Documentation, Version 2.
23. C. Baum, W. Beullens, L. Braun, C. D. de Saint Guilhem, M. Klooß, C. Majenz, S. Mukherjee, E. Orsini, S. Ramacher, C. Rechberger, L. Roy, and P. Scholl, *FAEST v2: Algorithm Specifications*. NIST Post-Quantum Cryptography Standardization Process, October 2024. Algorithm Specifications and Supporting Documentation, Version 2.0.
24. J.-C. Deneuville, T. Debris-Alazard, L. Ducas, L. Ferreira, P. Gaborit, T. Gougeon, M. Rivain, and J.-P. Tillich, *MQOM: MQ on my Mind - Algorithm Specifications*. NIST Post-Quantum Cryptography Standardization Process, February 2025. Algorithm Specifications and Supporting Documentation, Version 2.0.
25. A. Banerjee, C. Peikert, and A. Rosen, "Pseudorandom functions and lattices," in *Advances in Cryptology–EUROCRYPT 2012*, pp. 253–270, Springer, 2012. Also available at https://eprint.iacr.org/2011/401.
26. O. Regev, "On lattices, learning with errors, random linear codes, and cryptography," *Journal of the ACM (JACM)*, vol. 56, no. 6, pp. 1–40, 2009.
27. N. I. of Standards and T. (NIST), "SHA-3 standard: Permutation-based hash and extendable-output functions," Tech. Rep. FIPS PUB 202, U.S. Department of Commerce, August 2015.
28. E. Fujisaki and T. Okamoto, "Secure integration of asymmetric and symmetric encryption schemes," in *Advances in Cryptology — CRYPTO'99*, vol. 1666 of *Lecture Notes in Computer Science*, Springer, 1999.
29. J. M. Pollard, "The fast fourier transform in a finite field," *Mathematics of Computation*, vol. 25, no. 114, pp. 365–374, 1971.