Optimizing HAETAE and SMAUG-T on Cortex-M4 for Resource-Constrained IoT Devices

JunHyeok Choi, DongHyun Shin, and Seog Chung Seo

***Abstract*—The practicality of post-quantum cryptography (PQC) on IoT devices depends on both operation cycle counts and communication costs from public values (public keys, ciphertexts, and signatures). The DSA HAETAE and the KEM SMAUG-T, both selected in the KpqC competition, provide smaller public values than ML-DSA and ML-KEM; however, the lack of platform-specific optimization leaves their operation cycle counts high, which can offset this advantage. In this paper, we optimize HAETAE and SMAUG-T on the Cortex-M4, a representative 32-bit embedded MCU, and conduct a multi-faceted evaluation. We fix two bugs in the existing implementation of the HAETAE $\mathcal{N}(\mathbf{s})$ evaluation and propose Autocorrelation-based Spectral-Norm Evaluation (ASNE), which replaces the per-polynomial FFTs with a single autocorrelation FFT. Together with further optimizations, we achieve improvements of up to 309.1%, 4.8%, and 6.0% in KeyGen, Sign, and Verify, respectively, over the state-of-the-art implementation. For SMAUG-T, we analyze the coefficient ranges of the polynomial multiplication operands to select an auxiliary NTT-friendly modulus q', and count the number of operations per layer to apply the optimal NTT. As a result, we achieve improvements of up to 266.7%, 348.9%, and 341.0% in KeyGen, Encaps, and Decaps, respectively, over the reference C implementation. Furthermore, we apply static analysis and dudect measurements to the optimized kernels and observe no evidence of timing leakage. Using models anchored to prior IoT protocol measurements, we estimate the transmission-unit count, end-to-end latency, and energy, showing that HAETAE and SMAUG-T can be competitive PQC alternatives in constrained wireless environments.**

***Index Terms*—Post-Quantum Cryptography, HAETAE, SMAUG-T, Cortex-M4, lattice-based cryptography**

I. Introduction

Advances in quantum computing threaten the security of existing public-key cryptographic algorithms. Accordingly, a migration to post-quantum cryptography (PQC), which is secure even against quantum computers, is required across all communication environments. Applying PQC to resource-constrained Internet of Things (IoT) devices requires considering not only computational performance, memory capacity, and energy consumption but also the limited wireless bandwidth and packet size. Therefore, the practicality of PQC for IoT devices must be evaluated comprehensively, encompassing not only the cycle count of cryptographic operations but also stack usage and code size, parameters such as the public key, ciphertext, and signature, and the resulting communication cost.

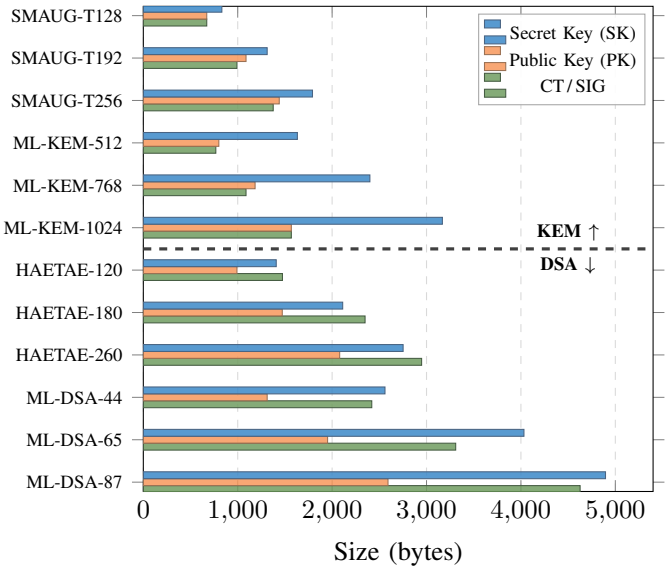


Fig. 1. Parameter sizes in bytes for the KpqC schemes and NIST PQC standards.

The Cortex-M4 is a representative 32-bit embedded microcontroller with limited SRAM and flash memory, and it is used in a wide range of IoT devices such as drones, smart-home devices, and wearables. In NIST's PQC standardization process, the Cortex-M4 was also used as the environment for evaluating the performance and RAM requirements of candidate algorithms on embedded devices, and pqm4 and subsequent studies have likewise used this platform as a representative optimization and benchmarking environment for embedded PQC implementations [1]–[3]. Therefore, reducing the computational cost and stack usage of PQC algorithms on the Cortex-M4 and evaluating their communication cost are key considerations in assessing real-world IoT deployability.

Research on applying PQC to achieve quantum security in IoT environments is also being actively conducted [4]–[7]. Many of these studies emphasize not only the computation time of the cryptography itself but also the impact of the sizes of parameters such as the public key, signature, and ciphertext. Schöffel et al. show that the main cause of TLS handshake latency is not the cryptographic computation but the increased bandwidth demand introduced by PQC [5]. Liu et al. show that using Kyber-512 in a BLE environment increases the pairing time by roughly a factor of nine compared with ECDH, owing to the larger packet size [7]. These results suggest that the

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (No. RS-2025-25411243). *(Corresponding author: Seog Chung Seo.)*

JunHyeok Choi, DongHyun Shin, and Seog Chung Seo are with the Department of Cybersecurity, Kookmin University, Seoul 02707, South Korea (e-mail: {c981126, dkldkl13, scseo}@kookmin.ac.kr).

TABLE I
PERFORMANCE IMPROVEMENTS (%) OF KEYGEN, ENCAPS/SIGN, AND DECAPS/VERIFY FOR HAETAE AND SMAUG-T ON THE CORTEX-M4 NUCLEO-L4R5ZI BOARD

Algorithm	KeyGen	Encaps/Sign	Decaps/Verify
HAETAE-120	+227.0	+4.6	+5.5
HAETAE-180	+230.6	+4.6	+5.4
HAETAE-260	+309.1	+4.8	+6.0
SMAUG-T128	+221.8	+337.5	+317.9
SMAUG-T192	+243.2	+335.3	+327.7
SMAUG-T256	+266.7	+348.9	+341.0

PQC parameter size is a key factor that directly affects overall network performance, including the communication latency, packet fragmentation, and transmission failure rate of IoT systems.

NIST conducted a PQC competition and standardized the selected algorithms, publishing the standard documents FIPS 203 and FIPS 204 for the Key Encapsulation Mechanism (KEM) ML-KEM and the Digital Signature Algorithm (DSA) ML-DSA in 2024 [8], [9]. NIST recommends these two algorithms as the primary algorithms for use in general-purpose environments [1]. Through its own KpqC competition for selecting domestic PQC algorithms, Korea selected two KEMs and two DSAs in 2025 [10]–[13]. Among them, the KEM SMAUG-T and the DSA HAETAE are lattice-based PQC schemes, like ML-KEM and ML-DSA, and they have an advantage over them in terms of parameters. Fig. 1 compares the parameter sizes of the NIST PQC schemes ML-KEM and ML-DSA with the KpqC schemes SMAUG-T and HAETAE. SMAUG-T and HAETAE provide smaller public keys, ciphertexts, and signatures than ML-KEM and ML-DSA. In IoT environments with severe bandwidth constraints or latency sensitivity, SMAUG-T and HAETAE may have an advantage over ML-KEM and ML-DSA.

However, the advantage in parameter size does not directly translate into an overall advantage when applying PQC to IoT devices. Even before standardization, ML-KEM and ML-DSA had their core operations, such as NTT-based polynomial multiplication, modular arithmetic, and Keccak, optimized across a variety of environments including Cortex-M4 and AVX2 [14]–[17]. In contrast, SMAUG-T and HAETAE, which were selected relatively recently through the KpqC competition, have received comparatively little such platform-specific optimization. Although studies on the stack usage of HAETAE and SMAUG-T have recently been published, performance optimization studies are still lacking, and a performance gap with ML-KEM and ML-DSA remains [18]–[20]. Therefore, translating the communication-side advantage of the small parameter sizes of SMAUG-T and HAETAE into real-world deployability requires optimization to narrow this performance gap. In this paper, by optimizing the core operations of SMAUG-T and HAETAE on the Cortex-M4 to narrow the performance gap with ML-KEM and ML-DSA and combining this with the communication advantage provided by their small parameter sizes, we show that the SMAUG-T and HAETAE algorithms can be competitive PQC alternatives on IoT devices.

Contributions. We analyze the performance bottlenecks of HAETAE and SMAUG-T on the Cortex-M4 and present optimized implementations tailored to the structural characteristics of each algorithm and to the target architecture. We also evaluate the proposed implementations comprehensively from an embedded perspective and provide a measurement-anchored analysis of their protocol-level implications in constrained IoT environments. Table I shows the performance improvements of HAETAE and SMAUG-T on the Cortex-M4 over the existing implementations. The main contributions of this paper can be summarized as follows.

- **Correctness-aware reformulation of HAETAE $\mathcal{N}(\mathbf{s})$ evaluation.** We identify two existing bugs and fix them in the $\mathcal{N}(\mathbf{s})$ evaluation of the existing HAETAE implementation that, for certain inputs, produce a value different from the specification's definition. We also propose Autocorrelation-based Spectral-Norm Evaluation (ASNE), which reconstructs the d per-secret-key-polynomial FFTs into d NTTs and a single FFT. ASNE performs the FFT, which requires many fixed-point complex multiplications and roundings, only once and replaces the rest with NTTs based on integer modular arithmetic, thereby reducing the cost of the $\mathcal{N}(\mathbf{s})$ evaluation.

- **Optimized HAETAE on Cortex-M4.** We profile the existing Cortex-M4 implementation in detail to analyze the remaining performance bottlenecks, and we propose optimization techniques tailored to the algorithmic characteristics of HAETAE and the Cortex-M4 architecture. In `KeyGen`, we implement ASNE as a 256-point FFT and optimize the $\mathcal{N}(\mathbf{s})$ evaluation by applying a fused butterfly, 3+3+2 layer merging, and NTT-result reuse. In `Sign`, we exploit the small coefficient ranges of the challenge c and the secret polynomial vectors $\mathbf{s}_1, \mathbf{s}_2$ to apply a small NTT based on FNT-257 and NTT-769, and we reduce pipeline stalls by improving the instruction scheduling and register usage of the inner functions of hyperball sampling. In addition, we apply early rejection, static-inline conversion of frequently called functions, and assembly porting. As a result, on the Cortex-M4 we improve the performance of `KeyGen`, `Sign`, and `Verify` by up to 309.1%, 4.8%, and 6.0%, respectively, over the existing implementation.

- **Optimized SMAUG-T on Cortex-M4.** Based on our profiling results, polynomial multiplication spends the majority of the cycles in SMAUG-T. We focus our optimization efforts on this operation. SMAUG-T is an NTT-unfriendly algorithm due to its use of a power-of-two modulus q. The current reference C implementation performs polynomial multiplication using the Toom-Cook method. We analyze state-of-the-art NTT techniques and explore the most efficient methods considering both the parameters of SMAUG-T and architectural characteristics. We demonstrate that NTT-based multiplication using homomorphism is

more efficient on Cortex-M4. Specifically, we show that the NTT method using a large single modulus q' is optimal for Cortex-M4. As a result, compared with the reference C implementation, our optimized implementation achieves performance improvements of up to 266.7%, 348.9%, and 341.0% for `KeyGen`, `Encaps`, and `Decaps`, respectively, on Cortex-M4. In particular, `Encaps` at security level 1 requires fewer cycles than ML-KEM.

- **Comprehensive evaluation of the optimized implementations and measurement-anchored analysis of IoT protocol implications.** We evaluate the cycle count, stack usage, and code size of the core kernels and the full scheme operations of HAETAE and SMAUG-T in the same Cortex-M4 environment, and we compare them with major PQC implementations. We also statically analyze the proposed optimized kernels. Using dudect, a statistics-based timing-leakage detection tool, we observe no timing-leakage evidence on the target board within the measured range. Finally, through a counterfactual model anchored to the measured results of prior PQC-IoT studies, we quantify the constrained IoT link conditions under which the small public values (public keys, ciphertexts, and signatures) and computational optimizations of HAETAE and SMAUG-T reduce the transmission cost and also provide an advantage in end-to-end cost.

II. Preliminaries

A. HAETAE

HAETAE is a lattice-based signature scheme selected as one of the two final DSAs in the KpqC competition [12]. HAETAE is designed based on the Fiat-Shamir with aborts paradigm, focusing on reducing the public key and signature sizes. The lattice is constructed over the polynomial ring $R_q = \mathbb{Z}_q[x]/(x^n+1)$, where $q = 64\,513$ and $n = 256$.

Fig. 2 shows simplified pseudocode for HAETAE `KeyGen`, `Sign`, and `Verify`. The coefficients of the coordinate polynomials of $\mathbf{s}(\in S_\eta^{k+\ell})$ all have absolute value at most $\eta = 1$, and the coordinate polynomials other than the leading 1 are denoted $s_1, \ldots, s_d$ $(d = k+\ell-1)$. In HAETAE-120 and HAETAE-180, the absolute-value bound on $\mathbf{s}_2$ increases to $\eta_{\max} = 2$ during the public-key compression of `KeyGen`. In addition, `KeyGen` restarts at line 5 whenever the $\mathcal{N}(\mathbf{s})$ evaluation of the secret key $\mathbf{s}$ exceeds the reject threshold $\gamma^2 n$, so that only secret keys admitting sufficiently small signatures are accepted. The $\mathcal{N}(\mathbf{s})$ evaluation is defined as follows:

$$\mathcal{N}(\mathbf{s}) := \tau \sum_{i=1}^{m} \max^{\text{i-th}}_{0\le j<2n} \|\mathbf{s}(\omega_j)\|_2^2 + r \cdot \max^{(m+1)\text{-th}}_{0\le j<2n} \|\mathbf{s}(\omega_j)\|_2^2, \tag{1}$$

where ω_j is a primitive $2n$-th root of unity, a root of X^n+1, and $m = \lfloor n/\tau \rfloor$, $r = n \bmod \tau$. There are n such evaluation points, and j denotes the index range $0 \le j < 2n$ over which these n evaluation points lie. Each component $\|\mathbf{s}(\omega_j)\|_2$ is a singular value of $\mathbf{s}$, and the largest of these is the spectral norm of the matrix. Collecting the squared singular value $\|\mathbf{s}(\omega_j)\|_2^2$ at each frequency point ω_j over all ω_j yields the spectrum. $\mathcal{N}(\mathbf{s})$ is the weighted sum of the top $(m+1)$ values of this spectrum with the challenge weight τ and the remainder r, and it is used for the secret-key reject decision. The challenge polynomial c used in `Sign` is sparse, with Hamming weight $\tau \in \{58, 80, 128\}$ for HAETAE-120/180/260, respectively. Therefore, multiplying by c amounts to summing τ signed cyclic shifts of each secret polynomial, so every coefficient of $c\mathbf{s}$ is bounded in absolute value by $\tau\eta$. The blue-highlighted parts in Fig. 2 indicate the target operations we optimize for HAETAE.

`KeyGen`(1^λ) :
1 : $(\mathbf{A_{gen}}) \leftarrow \mathcal{R}_q^{k\times(\ell-1)}$ and $(\mathbf{s_1}, \mathbf{s_2}) \leftarrow S_\eta^{\ell-1} \times S_\eta^k$
2 : $\mathbf{b} = \mathbf{A_{gen}} \cdot \mathbf{s_1} + \mathbf{s_2} \in \mathcal{R}_q^k$
3 : $\mathbf{A} = (-2\mathbf{b} + q\mathbf{j} \,|\, 2\mathbf{A_{gen}} \,|\, 2\mathbf{Id_k}) \bmod 2q$
4 : $\mathbf{s} = (1, \mathbf{s_1}, \mathbf{s_2})$
5 : **if** $\mathcal{N}(\mathbf{s}) > \gamma^2 n$ **then** restart
6 : **return** $\mathtt{sk} = (\mathbf{A}, \mathbf{s})$, $\mathtt{pk} = \mathbf{A}$

`Sign`$(\mathtt{sk}, M)$:
1 : $\mathbf{y} \leftarrow U(\mathcal{B}_{(1/N)\mathcal{R},(k+\ell)}(B))$
2 : $\mathbf{w} \leftarrow \mathbf{A}[\mathbf{y}]$
3 : $c = H(\mathbf{w}, M) \in \mathcal{R}_2$
4 : $\mathbf{z} = (\mathbf{z_1}, \mathbf{z_2}) = \mathbf{y} + (-1)^b c\mathbf{s}$ for $b \leftarrow U(\{0,1\})$
5 : **if** $\|\mathbf{z}\|_2 \ge B'$ **then** restart
6 : **else if** $\|\mathbf{2z} - \mathbf{y}\|_2 < B$ **then** restart with probability 1/2
7 : **return** $\sigma = (\lfloor \mathbf{z} \rceil, c)$

`Verify`$(\mathtt{pk}, M, \sigma = (\mathbf{z}, c))$:
1.$\tilde{\mathbf{w}} = \mathbf{Az} - qc\mathbf{j} \bmod 2q$
2.**return** $(c = H(\tilde{\mathbf{w}}, M)) \wedge \left(\|\mathbf{z}\|_2 < B + \frac{\sqrt{n(k+\ell)}}{2}\right)$

Fig. 2. Description of HAETAE [12].

B. SMAUG-T

SMAUG-T is a lattice-based cryptosystem that has been selected as a final algorithm in the PKE and KEM categories of the KpqC competition [10]. This algorithm is based on Module Learning With Errors (MLWE) for the public key and Module Learning With Rounding (MLWR) for the ciphertext, and operates over the polynomial ring $R_q := \mathbb{Z}_q[x]/(x^n+1)$, where the coefficients lie in the integer ring $\mathbb{Z}_q$. The core operations of SMAUG-T are matrix-vector products and vector inner products. $\mathbf{A}$ is a square matrix of size $k \times k$, and $\mathbf{c}_1$, $\mathbf{s}$, and $\mathbf{r}$ are vectors of size $k \times 1$ (cf. Fig. 3). SMAUG-T uses a power-of-two modulus q, enabling efficient modular reduction through bitwise operations. However, a power-of-two modulus does not admit the primitive $2n$-th root of unity that the NTT requires, so the NTT cannot be applied directly over q. The blue-highlighted parts in Fig. 3 indicate the target operations we optimize for SMAUG-T. Fig. 3 shows the pseudocode for SMAUG-T.PKE, whose functions are invoked by the actual KEM. For example, the `Decaps` of SMAUG-T calls both the `Dec` and `Enc` functions to perform the Fujisaki–Okamoto (FO) transform.

$\underline{\texttt{KeyGen}}(1^\lambda)$:
$1: \mathbf{A} \leftarrow \mathcal{R}_q^{k\times k}, \mathbf{s} \leftarrow \texttt{HWT}_{h_s} \in S_\eta^k$
$2: \mathbf{e} \leftarrow \widetilde{D}_\sigma \in \mathcal{R}^k$
$3: \mathbf{b} \leftarrow -\mathbf{A}^\top \cdot \mathbf{s} + \mathbf{e} \in \mathcal{R}_q^k$
$4:$ **return** $\texttt{pk} = (\mathbf{A}, \mathbf{b}),\ \texttt{sk} = \mathbf{s}$

$\underline{\texttt{Enc}}(\texttt{pk}, \mu)$: $\quad \triangleright\ \texttt{pk} = (\mathbf{A}, \mathbf{b}),\ \mu \in \mathcal{R}_t$
$1: \mathbf{r} \leftarrow \texttt{spCBD}_r \in S_\eta^k$
$2: \mathbf{c}_1 = \lfloor p/q \cdot \mathbf{A} \cdot \mathbf{r} \rceil \in \mathcal{R}_p^k$
$3: c_2 = \lfloor p'/q \cdot \langle \mathbf{b}, \mathbf{r} \rangle + p'/t \cdot \mu \rceil \in \mathcal{R}_{p'}$
$4:$ **return** $\texttt{ct} = (\mathbf{c}_1, c_2)$

$\underline{\texttt{Dec}}(\texttt{sk}, \mathbf{c})$: $\quad \triangleright\ \texttt{sk} = \mathbf{s},\ \mathbf{c} = (\mathbf{c}_1, c_2)$
$1: \mu' = \lfloor t/p \cdot \langle \mathbf{c}_1, \mathbf{s} \rangle + t/p' \cdot c_2 \rceil \in \mathcal{R}_t$
$2:$ **return** μ'

Fig. 3. Description of SMAUG-T.PKE [10].

C. Cortex-M4

Cortex-M4 is a 32-bit microcontroller unit (MCU) based on the ARMv7-M architecture [21]. It has sixteen core registers: thirteen general-purpose registers `r0-r12`, the stack pointer `r13`, the link register `r14`, and the program counter `r15`, as well as thirty-two single-precision FP registers `s0-s31`. Although FP registers can be used only for floating-point operations, they provide temporary storage that software can access more cheaply than memory, subject to the same save-and-restore rules discussed below.

Registers in Cortex-M4 are categorized into caller-saved and callee-saved registers depending on who is responsible for preserving their values during a function call. Caller-saved registers must be preserved by the caller, while callee-saved registers must be preserved within the callee function. Among the general-purpose registers, `r0-r3` and `r12` are caller-saved, and `r4-r11` are callee-saved. Among the FP registers, `s0-s15` are caller-saved, while `s16-s31` are callee-saved. When callee-saved registers are used in a function, their original values must be saved and restored before the function returns. To do so, the typical approach is to push the values onto the stack at the beginning of the function and pop them back before returning. The link register can also be used for integer operations after undergoing the same save-and-restore process.

Cortex-M4 provides DSP extension, including DSP and SIMD instructions, to enhance performance. It also features a hardware unit called the barrel shifter, which enables a shift to be applied to an operand directly within an instruction without requiring an additional separate instruction. The following are the key instructions we use:

- `pkh{b,t}{b,t} r0, r1, r2` : Pack the {lower(b),upper(t)} 16-bit of `r1` and the {lower,upper} 16-bit of `r2` into `r0`.
- `smull r0, r1, r2, r3` : Multiply `r2` by `r3` as signed 32-bit integers to produce a 64-bit result, store the lower 32-bit into `r0` and the upper 32-bit into `r1`.
- `smlal r0, r1, r2, r3` : Multiply `r2` by `r3`, add the product to the 64-bit value formed by `r1 << 32 | r0`, and store the result back into `r0` and `r1`.
- `smmulr r0, r1, r2` : Multiply `r1` by `r2`, round the product, and store the upper 32 bits into `r0`.
- `add r0, r1, r2, lsl #4` : Perform a logical shift left by 4-bit on `r2`, add the result to `r1`, and store it in `r0`. In short, `r0 = r1 + (r2 << 4)`.

D. Number-Theoretic Transform (NTT)

In this paper, we denote the Number-Theoretic Transform as NTT, and refer to its three operations as NTT (forward NTT), basemul ($\circ$), and iNTT (inverse NTT), respectively. The multiplication of two polynomials a and b over R_q is performed in the NTT domain as $ab = \text{iNTT}(\text{NTT}(a) \circ \text{NTT}(b))$. When R_q is defined over the cyclotomic polynomial $x^n + 1$, and if q is prime and a primitive $2n$-th root of unity exists in $\mathbb{Z}_q$, then $\psi_{2n}^n \equiv -1 \pmod q$ holds. Using this property, $x^n + 1$ can be rewritten as $x^n - \psi_{2n}^n$. By applying the Chinese Remainder Theorem (CRT), it is possible to construct the following homomorphism [22].

$$\frac{\mathbb{Z}_q[x]}{x^n+1} \cong \prod_{i=0}^{n-1} \frac{\mathbb{Z}_q[x]}{\left(x - \psi_{2n}^{2\text{brv}_n(i)+1}\right)}$$

Two main algorithms for computing radix-2 NTTs are the Cooley–Tukey (CT) and Gentleman–Sande (GS) butterfly [23], [24]. The CT butterfly is used in NTT, and maps a pair (a_0, a_1) and a constant ψ_{2n} to $(a_0 + a_1\psi_{2n}, a_0 - a_1\psi_{2n})$. In contrast, the GS butterfly is used in iNTT, and maps (a_0, a_1) and ψ_{2n}^{-1} to $(a_0 + a_1, (a_0 - a_1)\psi_{2n}^{-1})$.

III. Algorithm Analysis

This section introduces the optimization techniques applied in the prior work on HAETAE and SMAUG-T, and identifies the performance bottlenecks through profiling on Cortex-M4 [10], [12]. For profiling, we use a NUCLEO-L4R5ZI board and measure the cycle count using the SysTick cycle counter of the HAL. We report the key bottleneck operations separately and group the remaining operations under Other.

A. Existing HAETAE Implementation and Profiling Results

Cheon et al. present the state-of-the-art implementation of HAETAE on Cortex-M4 [12]. HAETAE uses a 16-bit modulus $q = 64\,513$ and performs a 32-bit NTT for $n = 256$, similar to Dilithium, so it can utilize the optimized NTT code from [14]. Twiddle factors and related constants are adjusted according to the parameters of HAETAE. For hyperball sampling in `Sign`, the Cumulative Distribution Table (CDT) sampler and exponential function are optimized using SIMD instructions. For SHAKE, the implementation uses the highest-performing version on Cortex-M4 introduced in [25].

Fig. 4 shows the profiling result for the state-of-the-art HAETAE-120 implementation on Cortex-M4 [12]. $\mathcal{N}(\mathbf{s})$ evaluation is used only in `KeyGen` and accounts for 65.3% of the total cycles. Except for `KeyGen`, SHA-3 and Sampling account for the most cycles. Lastly, NTT-based polynomial multiplication accounts for 7.1% to 10% of the performance

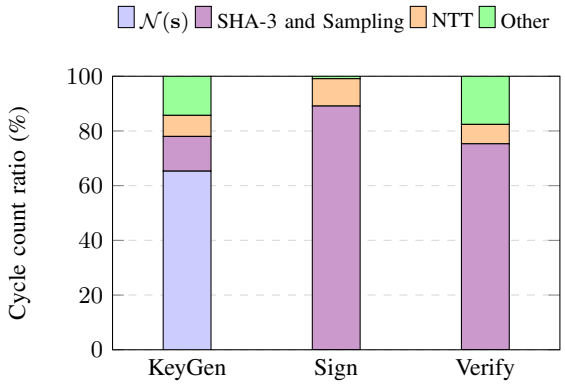


Fig. 4. Cycle-count profiling of HAETAE-120 on Cortex-M4.

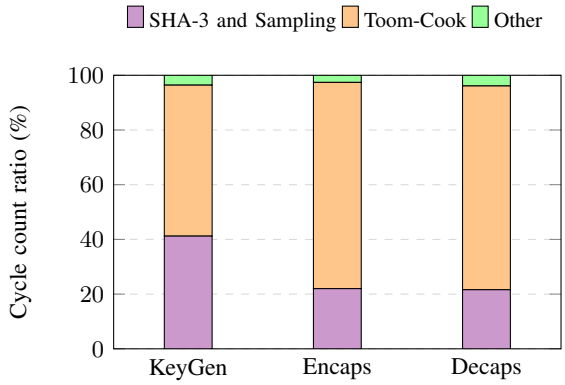


Fig. 5. Cycle-count profiling of SMAUG-T128 on Cortex-M4.

cost in each function. In the $\mathcal{N}(\mathbf{s})$ evaluation, the FFT accounts for most of the computational cost. Since the FFT is implemented in plain C in the existing implementation, we select it as an optimization target. For `Sign`, we target the two assembly functions called inside hyperball sampling, and for both `Sign` and `Verify` we select the NTT-based polynomial multiplication as an optimization target. Section V presents the detailed optimizations of HAETAE on Cortex-M4.

B. Existing SMAUG-T Implementation and Profiling Results

Narlı et al. present an optimization study of SMAUG-T on Cortex-M4, but that work targets an earlier version of SMAUG-T and therefore differs in its operations and parameters [20]. We thus port the latest version of the SMAUG-T reference C implementation to Cortex-M4 and perform profiling to identify optimization targets. Fig. 5 shows the profiling result of SMAUG-T128 on Cortex-M4. Toom-Cook-based polynomial multiplication accounts for the largest number of cycles, taking up 55.2%, 75.4%, and 74.5% of the total cycles in `KeyGen`, `Encaps`, and `Decaps`, respectively. SHA-3 and Sampling account for the second largest portion, taking 41.2%, 22%, and 21.6%, over 90% of which is due to SHA-3. Based on these profiling results, we propose optimization strategies focusing on polynomial multiplication and SHA-3. Section VI presents the detailed optimizations of SMAUG-T on Cortex-M4.

IV. Revisiting HAETAE $\mathcal{N}(\mathbf{s})$ Evaluation: Implementation Deviations and ASNE

This section analyzes two implementation bugs in the $\mathcal{N}(\mathbf{s})$ evaluation that cause existing implementations to accept secret keys that the specification's $\mathcal{N}(\mathbf{s})$ criterion should reject, and it presents Autocorrelation-based Spectral-Norm Evaluation (ASNE), which reformulates the $\mathcal{N}(\mathbf{s})$ evaluation as a single autocorrelation FFT. We then show that ASNE is mathematically equivalent to the specification's $\mathcal{N}(\mathbf{s})$ evaluation under real arithmetic and, moreover, that it conforms to the specification's definition more closely than the latest reference implementation.

A. The $\mathcal{N}(\mathbf{s})$ Evaluation in Existing Implementations and Its Deviation from the Specification

The evaluation of $\mathcal{N}(\mathbf{s})$ (Eq. (1)), which is used for the reject decision in `KeyGen`, proceeds in the following three steps.

(1) spectrum step. Apply an FFT to each secret-key coordinate polynomial $(s_1, \ldots, s_d)$ and accumulate the pointwise squared magnitudes to obtain $\mathrm{sum}[j] = \sum_i |s_i(\omega_j)|^2$.

(2) selection step. Select the $m{+}1$ largest of the $\mathrm{sum}[j]$ values obtained in (1).

(3) weighting step. Multiply the top m values by τ and the $(m{+}1)$-th value by $r = n \bmod \tau$, then sum them all to obtain $\mathcal{N}(\mathbf{s})$.

If the resulting $\mathcal{N}(\mathbf{s})$ exceeds $\gamma^2 n$, the key is rejected and the key pair is regenerated. Analyzing the previous Cortex-M4 implementation of HAETAE [12] and the latest reference implementation of HAETAE [26], we found two implementation bugs that, for certain inputs, produce a value different from the specification's definition of $\mathcal{N}(\mathbf{s})$.

The first bug occurs in the (2) selection step of the previous Cortex-M4 implementation [12]. That implementation uses a `minmaxmask` function that relies on a mask in the selection step. It places the first $m{+}1$ values into the `bestm` array and, iterating from the $(m{+}2)$-th sum to the n-th sum, swaps the current sum with an entry of `bestm` whenever it is larger. The mask is then changed from -1 to 0, so that all further swaps against `bestm` beyond the swapped array index are skipped. Fig. 6 shows part of the actual source code. However, this function does not correctly compute the top $m{+}1$ values. For example, with $m{+}1 = 3$, when the candidate 40 enters the held set $[30, 50, 10]$, the correct selection that compares all elements yields $[40, 50, 30]$, whereas `minmaxmask` stops right after the first swap and outputs $[40, 50, 10]$. As a result, `bestm` fails to hold the true top-$(m{+}1)$ values and instead includes a smaller value, making $\mathcal{N}(\mathbf{s})$ smaller. This bug can be resolved by performing the comparison and swap over all values without a mask. Since the reference implementation follows the `minmax` approach that applies this fix, this bug is limited to the previous Cortex-M4 implementation.

The second bug occurs in the (3) weighting step and applies to both the previous Cortex-M4 implementation and the latest reference implementation. By the specification's definition, the minimum among the `bestm` values obtained in the (2) selection step should be multiplied by r and the

```
static inline void minmaxmask(int32_t *x, int32_t *y
    , int32_t *mask) {
  int32_t a = *x, b = *y;
  int32_t ab = (b ^ a) & *mask;
  int32_t c = b - a; c ^= ab & (c ^ b); c >>= 31;
  *mask &= ~c;          // first swap clears mask:
      rest of the pass is skipped
  c &= ab; *x = a ^ c; *y = b ^ c;
}
...
for (size_t i = N/TAU+1; i < N; i++) {
  int32_t mask = -1;
  for (size_t j = 0; j < N/TAU+1; j++)
    minmaxmask(&sum[i], &bestm[j], &mask);    //
        comparisons stop after first swap
}
```

Fig. 6. Top-$(m{+}1)$ selection in the previous Cortex-M4 implementation [12].

```
for (size_t i = 0; i < N/TAU+1; i++) {
  // mask flags EVERY entry equal to min -> all ties
       get r (spec: only one)
  int32_t fac = ((min - bestm[i]) >> 31);
  fac = (fac & TAU) ^ ((~fac) & (N % TAU));
  bestm[i] += 0x10200; bestm[i] >>= 10;
  bestm[i] *= fac; res += bestm[i];
}
```

Fig. 7. The weighting step in the previous Cortex-M4 implementation and the latest reference implementation [12], [26].

rest by τ. However, both implementations also multiply other terms equal to the minimum by r. Consequently, a value that should be multiplied by τ is multiplied by r instead, ultimately making $\mathcal{N}(\mathbf{s})$ smaller. Fig. 7 shows the actual source code that performs this operation. For example, for an even $m{+}1$ where `bestm` is $[100, 100, 60, 60]$ ($\min = 60$), the specification yields $\tau(100{+}100{+}60) + r \cdot 60$, whereas the implementation yields $\tau(100{+}100) + r(60{+}60)$, which is smaller by $60(\tau{-}r)$. Such boundary ties arise structurally and frequently in HAETAE-180, where $m{+}1$ is even. This bug can be resolved by adding a mechanism that clearly disambiguates the boundary.

Both are implementation bugs that evaluate $\mathcal{N}(\mathbf{s})$ as smaller than its value under the specification's definition, so an incorrect value passes the reject condition $[\mathcal{N}(\mathbf{s}) > \gamma^2 n]$. That is, a secret key that should have been rejected because $\mathcal{N}(\mathbf{s})$ exceeds the threshold passes instead. This is a specification-level correctness issue, not merely a performance issue. The $\mathcal{N}(\mathbf{s})$ of an incorrectly generated secret key can exceed the specification's upper bound and thereby affect the norm bound on which HAETAE's signature-size guarantee relies.

B. Autocorrelation-based Spectral-Norm Evaluation (ASNE): Unified $\mathcal{N}(\mathbf{s})$ Evaluation via a Single Autocorrelation FFT

To evaluate $\mathcal{N}(\mathbf{s})$, existing HAETAE implementations perform a total of d 256-point [26] (512-point [12]) Q16.16 FFTs. Q16.16 denotes a 32-bit fixed-point format with a 16-bit integer part and a 16-bit fractional part. By applying ASNE, which reformulates the $\mathcal{N}(\mathbf{s})$ evaluation as a single autocorrelation FFT, we replace the d FFTs with d 256-point NTTs and a single 256-point FFT. An NTT butterfly consists of one integer modular multiplication over $\mathbb{Z}_q$ together with one addition and one subtraction, whereas an FFT butterfly requires four Q16.16 multiplications with rounding and a larger number of additions and subtractions. Thus, even for the same 256-point transform, the key idea of ASNE is to reduce the expensive FFT from d times to once and replace the rest with lower-cost NTTs.

Algorithm 1 Five-step $\mathcal{N}(\mathbf{s})$ evaluation in ASNE.

Input: secret-key polynomial vector $(s_1, \ldots, s_d) \in R_q^d$ where $R_q = \mathbb{Z}_q[X]/(X^n + 1)$
Output: $\mathcal{N}(\mathbf{s}) \in \mathbb{Z}$

$T_{\mathrm{ntt}} \leftarrow \mathbf{0} \in \mathbb{Z}_q^n$
for $i \leftarrow 1, \ldots, d$ **do**
  $S_i \leftarrow \mathrm{NTT}(s_i)$
  for $j \leftarrow 0, \ldots, n-1$ **do**
    $T_{\mathrm{ntt}}[j]$ += montgomery_reduce$(S_i[j] \cdot S_i[n-1-j])$ ▷ Step 1: autocorrelation NTT accumulation
  end for
end for
$T_{\mathrm{ntt}} \leftarrow \mathrm{reduce32_2q}(T_{\mathrm{ntt}})$ ▷ Step 2: centered-range normalization
$T \leftarrow \mathrm{iNTT}(T_{\mathrm{ntt}})$ ▷ Step 3: integer coefficient recovery
$V \leftarrow \mathrm{FFT}_{\mathrm{Q16.16}}(T)$ ▷ Step 4: single FFT evaluation, $V[j] = T(\omega_j)$
Extract $v[j] = \mathrm{Re}\, V[j]$ for $j \in [0, n)$ ▷ Step 5: full-spectrum 256-read
return spec-rank weighted $\sum$ over top-$(m+1)$ of all n bins $(1 + v)$ ▷ leading-1 recovery

As seen in the (1) spectrum step of Section IV-A, the value needed at each ω_j is the sum of the pointwise squared magnitudes of the d coordinate polynomials, $\mathrm{sum}[j] = \sum_{i=1}^{d} |s_i(\omega_j)|^2$. To obtain this value, existing implementations perform a separate FFT for each s_i to compute $s_i(\omega_j)$. However, since ω_j is a primitive $2n$-th root of unity on the unit circle, we have $\omega_j^{-1} = \overline{\omega_j}$, and since the coefficients of s_i are real, $s_i(\overline{\omega_j}) = \overline{s_i(\omega_j)}$ holds. Hence each term's squared magnitude can be rewritten as the product of the value and its conjugate, $|s_i(\omega_j)|^2 = s_i(\omega_j)\,\overline{s_i(\omega_j)} = s_i(\omega_j)\, s_i(\omega_j^{-1})$, which equals the polynomial $s_i(X)\, s_i(X^{-1})$ evaluated at ω_j. Summing this over i, the spectrum value at every ω_j coincides with $T(\omega_j)$, the evaluation of the single autocorrelation polynomial $T(X) := \sum_{i=1}^{d} s_i(X)\, s_i(X^{-1})$. That is, instead of applying an FFT to each of the d polynomials, forming the autocorrelation polynomial T once and evaluating it with a single FFT yields all spectrum values simultaneously.

Algorithm 1 describes the $\mathcal{N}(\mathbf{s})$ evaluation of ASNE in five steps. First, to compute the autocorrelation polynomial $T(X)$ above, an NTT is performed on each s_i. The NTT of the conjugate $s_i(X^{-1})$ is obtained without extra computation, using the fact that it equals the conjugate of the index-$(n-1-j)$ value of $\mathrm{NTT}(s_i)$. Then T_{ntt} is obtained through pointwise multiplication and accumulation. Applying a subtraction and an iNTT to T_{ntt} recovers the integer polynomial $T(X)$. Since the maximum value of a secret-key coordinate is $\eta_{\max} = 2$, and $d = k + \ell - 1 \le 10$, $n = 256$, we have

$|T_k| \le d \cdot n \cdot \eta_{\max}^2 \le 10{,}240 \ll \lfloor q/2 \rfloor = 32{,}256$. Therefore, the values of $T(X)$ can be recovered without overflow in the signed 32-bit integer range. The integer polynomial $T(X)$ is then evaluated with a 256-point FFT to obtain the bin value $V[j] = T(\omega_j) \in \mathbb{C}$ at each ω_j. By the self-adjoint property of the autocorrelation polynomial, $T(\omega_j)$ is real, so the real part $v[j] := \mathrm{Re}\, V[j]$ of each bin is exactly $T(\omega_j)$. Finally, the selection step and the weighting step are applied to $\{v[j]\}_{j=0}^{n-1}$ to obtain $\mathcal{N}(\mathbf{s})$.

C. Equivalence of ASNE to the Specification $\mathcal{N}(\mathbf{s})$ Evaluation

Using the three steps into which Section IV-A divided the specification's $\mathcal{N}(\mathbf{s})$ computation, spectrum, selection, and weighting, this subsection proves the equivalence under real arithmetic by showing that each stage of ASNE corresponds to these three steps in turn. We then construct an oracle that performs the $\mathcal{N}(\mathbf{s})$ evaluation in Q64.64 integer fixed-point and show that ASNE conforms to the specification's $\mathcal{N}(\mathbf{s})$ evaluation definition more closely than existing implementations.

spectrum step. The specification's secret key is $\mathbf{s} = (1 \mid s_1 \mid \cdots \mid s_d)$, which includes the constant 1 as its leading component, so at each ω_j we have $\|\mathbf{s}(\omega_j)\|_2^2 = 1 + \sum_{i=1}^{d} |s_i(\omega_j)|^2 = 1 + T(\omega_j)$. By the identity in Section IV-B, ASNE obtains $T(\omega_j)$ as the bin value $V[j]$ of a single autocorrelation FFT. Here $T(X)$ satisfies the self-adjoint property $\overline{T}(X) = T(X^{-1}) = T(X)$. By definition, each term $s_i(X)\, s_i(X^{-1})$ is unchanged under the substitution $X \leftrightarrow X^{-1}$, which merely swaps its two factors, so $T(X^{-1}) = T(X)$; and since the coefficients of s_i are real, coefficient conjugation coincides with the substitution $X \to X^{-1}$, so $\overline{T}(X) = T(X^{-1})$. Therefore $T(\omega_j)$ is real at every ω_j, the imaginary part of the Step 4 FFT evaluation is theoretically 0, and the real part $v[j] = \mathrm{Re}\, V[j]$ is exactly $T(\omega_j)$. Consequently, the $1 + v[j]$ that ASNE recovers at each bin equals the squared singular value $\|\mathbf{s}(\omega_j)\|_2^2$ of the spectrum step.

selection step. The definition selects the top $(m{+}1)$ squared singular values. Since the spectrum step showed that each bin value equals a squared singular value, ASNE's sorting of the bins by magnitude to pick the top $(m{+}1)$ is the same operation as the selection step of the definition. The self-adjoint property makes the spectrum symmetric under $j \leftrightarrow n - j$, so the same top set could be obtained from just the 128 bins of the first half; however, when bin values are tied, the rank weighting can be misassigned, so ASNE performs a full-spectrum selection that reads all 256 bins to pick the top $(m{+}1)$.

weighting step. ASNE multiplies the selected top m values by τ and the $(m{+}1)$-th by r before summing, which matches the weighting step of the definition. Since all three steps correspond, ASNE, under real arithmetic, produces the same value as the specification's $\mathcal{N}(\mathbf{s})$ for the entire $\mathbf{s}$ including the leading-1.

TABLE II
FALSE-ACCEPT COUNTS OF THE FOUR IMPLEMENTATIONS AGAINST THE Q64.64 ORACLE OVER 10^7 SAMPLES

Implementation	false-accept (vs. Q64.64 high-precision oracle)		
	HAETAE-120	HAETAE-180	HAETAE-260
Previous Cortex-M4 [12]	4,542,911	1,982,008	660,628
reference [26]	419	17,562	676
reference_fix	414	514	673
ASNE (this work)	**113**	**206**	**211**

The equivalence so far assumes real arithmetic; since the actual implementation uses a Q16.16 FFT, a rounding error arises in Step 4. Because of this error, the evaluation values of ASNE and the reference implementation differ from $\mathcal{N}(\mathbf{s})$ by at most about 22 in the same integer units as $\mathcal{N}(\mathbf{s})$. This value is obtained by multiplying the rounding error of about $5648/2^{16}$ accumulated over the 8 FFT stages per bin by the selection weight sum $n = 256$. Since this is 4–5 orders of magnitude smaller than the reject threshold $\gamma^2 n \in [6.1{\times}10^5, 8.5{\times}10^5]$, the accept/reject decisions of the two implementations can diverge only in the rare near-threshold cases where the evaluation value is very close to the threshold.

To verify whether ASNE conforms to the specification's $\mathcal{N}(\mathbf{s})$ definition more closely than existing implementations, we implement an oracle that performs the $\mathcal{N}(\mathbf{s})$ evaluation in Q64.64, a higher precision than the existing Q16.16. The rounding error accumulated over the 8 FFT stages in this oracle is about 10^{-13}, far smaller than the integer unit of the $\mathcal{N}(\mathbf{s})$ evaluation, and since it uses no floating point, it yields the same result regardless of platform and compiler. The oracle is compared against four implementations in total: the ASNE presented in this work, the latest reference implementation [26], reference_fix that fixes the bugs of the reference implementation, and the previous HAETAE Cortex-M4 implementation [12]. For each implementation, we ran `KeyGen` 10^7 times in an x86 environment and compared the oracle's accept/reject decision for the resulting $\mathbf{s}$. Table II reports, for the four implementations, the number of key-acceptance decision mismatches (false-accepts), i.e., cases where a secret key that should be discarded under the specification is adopted by the implementation as a valid key, meaning the oracle rejects it but the implementation accepts it. This is a decision at the KeyGen stage and is unrelated to accepting a forged signature during signature verification; hereafter, this paper refers to it as false-accept.

ASNE's false-accept counts are $113/206/211$, smaller than the reference's $419/17{,}562/676$ at every security level. The reference exhibits larger false-accepts because of the boundary-value bug described in Section IV-A. The false-accepts of reference_fix, which fixes the reference's bugs, are $414/514/673$, fewer than the reference. In particular, in HAETAE-180, where the boundary bug occurs most frequently, they drop sharply from 17,562 to 514. The previous Cortex-M4 implementation, which has both bugs, shows the largest false-accepts at $4{,}542{,}911/1{,}982{,}008/660{,}628$. ASNE shows lower false-accepts than all implementations, including reference_fix. This is because ASNE evaluates $\mathcal{N}(\mathbf{s})$ with a

single FFT on one autocorrelation polynomial, so its FFT rounding error is smaller than that of the other implementations. However, since the Q16.16 error still exists, its decisions are not perfectly identical to the oracle's.

Examining all 530 ASNE false-accept keys from the $N = 10^7$ measurement, we find that even the false-accepted keys exceed the oracle's reject threshold by at most 2.83, which lies within the maximum possible Q16.16 error of about 22. This is a phenomenon that also occurs identically in the reference implementation, which uses the same Q16.16 FFT. The opposite direction, false-reject, i.e., cases where the oracle accepts but the implementation rejects, only incurs the performance loss of discarding a specification-valid key and regenerating it, with no effect on security, so it is not reported separately in the table. The related security proof for HAETAE is beyond the scope of this paper and is left as future work.

V. Optimized Implementation of HAETAE on Cortex-M4

This section describes our optimization techniques for HAETAE on Cortex-M4. For `KeyGen`, we reduce the number of NTT operations required by the $\mathcal{N}(\mathbf{s})$ evaluation reformulated in Section IV and improve the implementation of its core operation, the FFT. For `Sign`, we exploit the characteristics of the $c\mathbf{s_1}, c\mathbf{s_2}$ products to apply a small NTT and improve the pipeline of the hyperball sampling core functions. Finally, as additional optimizations, we apply early rejection, which reorders the rejection conditions by their frequency of occurrence. We also convert repeatedly called functions to static inline functions and port other frequently called functions to assembly.

A. Implementation Optimization of the $\mathcal{N}(\mathbf{s})$ Evaluation in HAETAE `KeyGen`

This subsection presents two implementation optimizations that reduce the Cortex-M4 cycle count of ASNE, which is newly applied to the $\mathcal{N}(\mathbf{s})$ evaluation. The first applies a 256-point fused butterfly and layer merging to the FFT itself, which ASNE has reduced to a single call, thereby lowering the cost of one FFT. In the autocorrelation NTT step of ASNE, the second reuses the NTT of $\mathbf{s_1}$ that `KeyGen` has already computed, thereby reducing the number of NTT operations.

1) FFT Optimization: Fused Butterfly and 3+3+2 Layer Merging: The previous state-of-the-art Cortex-M4 implementation [12] expands each 256-point integer polynomial s_i into a 512-point Q16.16 complex polynomial and performs a 9-layer FFT. In contrast, the latest HAETAE reference implementation [26] applies a pre-twist that multiplies s_i by specific twiddle factors to reduce the FFT length from 512 to 256 points, thereby reducing the number of layers the FFT requires. We adopt this 256-point FFT approach in our implementation.

The CT butterfly used in the FFT outputs $a' = a + b\zeta$ and $b' = a - b\zeta$ for complex inputs $a(= a_r + a_i i), b(= b_r + b_i i), \zeta(= \zeta_r + \zeta_i i)$. The real part and the imaginary part of the complex product $b\zeta = (b_r\zeta_r - b_i\zeta_i) + (b_r\zeta_i + b_i\zeta_r)i$ each consist of two Q16.16 multiplications. The existing implementation computes these four multiplications independently, rounding a 64-bit product four times per complex multiplication. We apply a fused butterfly that accumulates, with `smull`/`smlal`, the two multiplications for each of the real and imaginary parts, and rounds only once per part. As a result, the number of `smull` and rounding operations per complex multiplication drops from four to two, and the rounding-error variance is also halved from $1/6$ to $1/12$. The resulting $\mathcal{N}(\mathbf{s})$ evaluation is therefore more accurate. Algorithm 2 gives the pseudocode of the fused butterfly.

Algorithm 2 Fused butterfly for complex numbers on Cortex-M4.

Inputs: complex numbers $a(= a_r + a_i i),\ b(= b_r + b_i i),\ \zeta(= \zeta_r + \zeta_i i)$ with 32-bit fixed-point components, temporary registers t_1, t_2, t_3.
Output: $a' = a + b\zeta = a'_r + a'_i i,\ b' = a - b\zeta = b'_r + b'_i i$

```
1:  rsb t2, ζ_i, #0                          ▷ t2 = −ζ_i
2:  smull t1, t3, b_r, ζ_r                   ▷ t3‖t1 = b_r ζ_r
3:  smlal t1, t3, b_i, t2                    ▷ t3‖t1 += b_i(−ζ_i)
4:  adds t1, t1, #0x8000                     ▷ round
5:  adc t3, t3, #0
6:  lsl t3, t3, #16                          ▷ pack
7:  pkhtb t1, t3, t1, asr #16
8:  smull b_r, t3, b_r, ζ_i                  ▷ t3‖b_r = b_r ζ_i
9:  smlal b_r, t3, b_i, ζ_r                  ▷ t3‖b_r += b_i ζ_r
10: adds b_r, b_r, #0x8000
11: adc t3, t3, #0
12: lsl t3, t3, #16
13: pkhtb b_r, t3, b_r, asr #16
14: sub.w b_i, a_i, b_r                      ▷ b'_i = a_i − Im(bζ)
15: add.w a_i, a_i, b_r                      ▷ a'_i = a_i + Im(bζ)
16: sub.w b_r, a_r, t1                       ▷ b'_r = a_r − Re(bζ)
17: add.w a_r, a_r, t1                       ▷ a'_r = a_r + Re(bζ)
```

Although the fused butterfly reduces the cost of one multiplication, the memory accesses repeated at every layer of the 8-layer FFT remain a performance bottleneck. The existing implementation performs a load/store for all coefficients at every layer. Exploiting the regularity of the coefficients and twiddle factors that participate in the per-layer butterfly operations, we apply layer merging that carries out the operations of several layers with a single load/store. For example, layer 1 performs the butterfly of (a_i, a_{i+1}), layer 2 that of (a_i, a_{i+2}), and layer 3 that of (a_i, a_{i+4}), so loading $a_i, \ldots, a_{i+7}$ at once allows the memory accesses of the intermediate layers to be skipped. The number of merged layers is determined by considering the number of registers on the Cortex-M4. In general, merging n layers requires holding 2^n coefficients and $2^n - 1$ twiddle factors simultaneously; since all of these values are Q16.16 complex, a total of $2^{n+1} + 2^{n+1} - 2$ registers are needed. Table III summarizes, for each number of merged layers, the required number of coefficients and twiddle factors and the additional FP registers needed. Merging four or more layers is infeasible with the Cortex-M4 registers, and by comparing the performance against 2-layer merging we adopt the faster

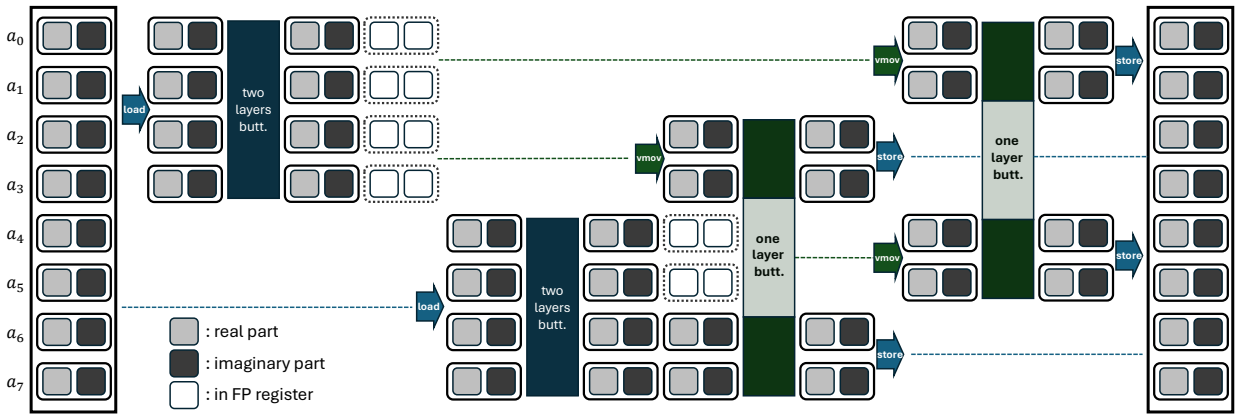


Fig. 8. Operation flow for 8 complex coefficients when merging three FFT layers (butt. = fused butterfly).

TABLE III
REQUIRED DATA AND FP REGISTER ALLOCATION FOR EACH LAYER MERGING CONFIGURATION

Layer merging	Required data		FP register	
	coeff.	twid.	inter.	twid.
1+$\cdots$+1	2	1	N/R	N/R
2+2+2+2	4	3	N/R	s3-s8
3+3+2	8	7	s3-s14	s15-s28
4+4	16	15	N/A	N/A

coeff. : coefficient, twid. : twiddle factor, inter. : intermediate value
N/R : Not Required, N/A : Insufficient Registers

3-layer merging 3+3+2.

Fig. 8 shows the operation flow of 3-layer merging. First, the coefficients a_0, a_1, a_2, a_3 are loaded into general-purpose registers to perform the butterflies of two layers, and the results are stored in FP registers. Next, a_4, a_5, a_6, a_7 are loaded to perform two layers, but to reduce the FP-register pressure only a_4, a_5 are stored in FP registers while a_6, a_7 are kept in general-purpose registers. In the last layer, the paired coefficients (such as a_2, a_3) are reloaded from the FP registers to complete the butterfly. We perform the above 3-layer merging twice for the first six layers of the 8-layer FFT and process all operations of the last two layers in general-purpose registers without storing intermediate values. The twiddle factors are loaded into the FP registers with a single `vldm` at the start of each loop iteration, and they are stored in memory in a pre-streamlined form so that the correct factor for the operation order is used.

2) Reusing $\mathbf{s_1}$ in the ASNE Autocorrelation NTT Step: ASNE requires the NTT of each s_i to build the autocorrelation polynomial $T(X) = \sum_{i=1}^{d} s_i(X)\, s_i(X^{-1})$. Here, in HAETAE-120 and HAETAE-180 where $D = 1$, the NTTs of the polynomials corresponding to $\mathbf{s_1}$ are already computed during the public-key computation of `KeyGen`, so they are reused and only the NTT for $\mathbf{s_2}$ is performed in ASNE. In contrast, in HAETAE-260 where $D = 0$, the NTTs of $\mathbf{s_1}, \mathbf{s_2}$ are performed after the $\mathcal{N}(\mathbf{s})$ evaluation, so the full NTTs of $\mathbf{s_1}, \mathbf{s_2}$ are performed in ASNE. HAETAE-260 could also store the NTT results of $\mathbf{s_1}, \mathbf{s_2}$ computed in ASNE and reuse them in the operations after the $\mathcal{N}(\mathbf{s})$ evaluation. However, unlike HAETAE-120 and HAETAE-180, where the number of $\mathbf{s_1}$ NTTs decreases in proportion to the number of rejection-loop iterations, HAETAE-260 saves only a single $\mathbf{s_1}, \mathbf{s_2}$ NTT computation while incurring additional stack usage to store it. Therefore, in HAETAE-260 the NTTs of $\mathbf{s_1}, \mathbf{s_2}$ are not reused; instead the corresponding operations are recomputed after the $\mathcal{N}(\mathbf{s})$ evaluation. This reuse only reduces the number of NTT operations and does not change the evaluation result.

B. Small NTT for `Sign`

The $c\mathbf{s_1}, c\mathbf{s_2}$ products in `Sign` consist of polynomials with small integer coefficients. The coefficients of $\mathbf{s_1}, \mathbf{s_2}$ are at most $\eta = 1$, and the challenge c has $\tau \in \{58, 80, 128\}$ coefficients equal to 1 per security level, so the absolute value of the product does not exceed a few hundred, and a 16-bit NTT alone suffices. With a 16-bit NTT the product of two coefficients fits in a single 32-bit register, avoiding the 64-bit multiplication and the accompanying heavy modular reduction required by the existing 32-bit NTT over $q = 64\,513$. Nevertheless, the existing implementation also applies the NTT over the scheme's standard modulus $q = 64\,513$ to the $c\mathbf{s_1}, c\mathbf{s_2}$ products; although q itself is 16-bit, the product of two coefficients may require up to 32 bits and thus needs 32-bit operations, incurring extra overhead. We therefore redefine the NTT with a smaller prime q' matched to the actual range of the $c\mathbf{s_1}, c\mathbf{s_2}$ products at each security level. Since the coefficients of $c\mathbf{s_1}$ and $c\mathbf{s_2}$ are bounded in absolute value by $\tau \cdot \max|\mathbf{s_2}|$, the computation can be regarded as being in $\mathbb{Z}_{q'}$ for any $q' > 2\tau \cdot \max|\mathbf{s_2}|$ [14]. This upper bound varies with the security level because, as mentioned in Section II, the public-key compression of `KeyGen` decomposes b as $b = 2^D b_1 + b_0$ ($b_0 \in \{-1, 0, 1\}$) and updates $\mathbf{s_2}$ to $\mathbf{s_2} = \mathbf{s_2} - b_0$ so that it matches the high bits b_1. In HAETAE-120 and HAETAE-180 this update raises $\max|\mathbf{s_2}|$ from η to $2\eta = 2$, whereas in HAETAE-260 the rounding is omitted, so $\mathbf{s_2}$ stays within $[-1, 1]$.

Table IV presents the per-security-level parameters of $c, \mathbf{s_2}$ and the maximum $2\tau \cdot \max|\mathbf{s_2}|$ obtained by substituting them. The maximum is 232 for HAETAE-120, 320 for HAETAE-180, and 256 for HAETAE-260. Thus HAETAE-120 and HAETAE-260 can use the Fermat Number Transform (FNT),

TABLE IV
PER-SECURITY-LEVEL $c, \mathbf{s_2}$ PARAMETERS AND THE MAXIMUM PRODUCT MAGNITUDE

Level	η	τ	$\mathbf{s_2}$ range	$2\tau \cdot \max \lvert\mathbf{s_2}\rvert$
HAETAE-120	1	58	$[-2, 2]$	232
HAETAE-180	1	80	$[-2, 2]$	**320**
HAETAE-260	1	128	$[-1, 1]$	256

an NTT over the Fermat prime $q' = 257$, whereas HAETAE-180 requires a larger prime because its maximum exceeds $q' = 257$. We apply $q' = 769$ for HAETAE-180, so the final per-security-level prime assignment is $q' = \{257, 769, 257\}$.

The reason $q' = 257$ is advantageous for HAETAE-120 and HAETAE-260 is the Fermat-prime relation $-1 \equiv 2^8 \pmod{257}$. For a Fermat prime, the following isomorphism holds.

$$\mathbb{Z}_{257}[X]/(X^{256}+1) \cong \mathbb{Z}_{257}[X]/(X^{128}+2^4) \times \mathbb{Z}_{257}[X]/(X^{128}-2^4),$$

Applying this to the CT butterfly $(a + 2^4 b,\ a - 2^4 b)$ replaces the modular multiplication with a shift. Since the Cortex-M4 allows an inline barrel shift on an `add` operand, this multiplication is handled in a single cycle. This isomorphism is applicable up to layer 3. The multiplications of the remaining layers are handled with Barrett reduction.

For HAETAE-180, which uses $q' = 769$, we apply 16-bit Plantard multiplication, defined for integers a and b over the prime q' as follows, where we follow the notation of [15].

$$\left[\left(\left[\left[a\, b\, q_{\text{inv}}\right]_{2l}\right]^l + 2^\alpha\right) q'\right]^l$$

Here, l is the word size, α and q_{inv} are precomputed values with $q' q_{\text{inv}} \equiv 1 \pmod{2^{2l}}$. In the NTT, a is a coefficient and b is a twiddle factor, so b is a fixed value. Therefore, precomputing $b\, q_{\text{inv}}$ in a table reduces the multiplications from three to two. Since the precomputed $b\, q_{\text{inv}}$ is 32-bit, computing $a\, b\, q_{\text{inv}}$ requires a 16-bit (a) $\times$ 32-bit ($b\, q_{\text{inv}}$) multiplication. The `smulw{t,b}` instruction handles this in a single cycle. In addition, two 16-bit coefficients can be packed into a 32-bit register and processed in parallel with SIMD instructions.

Since the maximum does not exceed 769 at any security level, a Plantard-multiplication-based 16-bit NTT with $q' = 769$ can be applied uniformly across all security levels. In environments where code size is critical, applying $q' = 769$ identically across all security levels allows the small NTT of `Sign` to be unified into a single assembly routine. This reduces the code size at the cost of conceding a small number of cycles.

C. Pipelining Improvements

The latest Cortex-M4 HAETAE implementation provides four kernels in assembly, of which the general-purpose primitives SHAKE and NTT already use highly optimized implementations. We improve the register usage and instruction scheduling of `approx_exp` and the `CDT sampler` used in the hyperball sampling of `Sign` to reduce pipeline stalls. The two functions incur the `push`/`pop` cost of callee-saved registers as well as stalls from instruction alignment and adjacent dependencies, and they run repeatedly inside the rejection loop of `Sign`, so their cycles accumulate in proportion to the number of iterations.

The `approx_exp` assembly scatters the ten constant loads needed by the computation across eight places. On the Cortex-M4 a single `ldr` costs 2 cycles, whereas n consecutive `ldr`s cost $n + 1$ cycles. Taking into account the constant-use positions and register dependencies of the existing implementation, we improve it to perform the ten constant loads in two contiguous blocks. This reduces the cycles spent on constant loads from 18 cycles to 12 cycles. In addition, by considering the live-range of the registers we reduce the number of registers used by one. This reduces the cycles spent on `push`/`pop` from 20 cycles to 18 cycles. In the `CDT sampler`, we change the instruction layout so that a 32-bit instruction can be read in its entirety with a single fetch, thereby reducing pipeline stalls.

The two functions are called inside `sample_gauss_sigma76`, which generates a single coefficient, and hyperball sampling repeatedly generates the coefficients of $\ell + k \in \{6, 9, 11\}$ degree-256 polynomials inside the rejection loop. Based on the average `Sign` iteration counts of $\{6, 5, 6\}$, coefficient generation occurs at least $M \cdot n(\ell + k) = \{9{,}216,\ 11{,}520,\ 16{,}896\}$ times, and the actual number of calls is larger because of the inner rejection of `sample_gauss_sigma76` and the outer hyperball rejection. As a result, the per-call cycle reduction of the two assembly functions accumulates across the whole of `Sign`.

D. Additional Optimizations

Beyond the major bottlenecks discussed above, the cost of the small functions that are called repeatedly inside HAETAE also accumulates from hundreds to thousands of times across `KeyGen`, `Sign`, and `Verify`. In `KeyGen` and `Sign`, which have a rejection loop, additional cost accumulates in proportion to the number of iterations. This subsection describes three additional optimizations that reduce this repeated cost.

1) Early Rejection: HAETAE repeats a structure that rejects a signature and regenerates it depending on whether the values produced during `Sign` satisfy certain conditions. HAETAE has three rejection conditions. rejection1 and rejection2 check whether values produced during signing satisfy certain conditions, and rejection3 is performed at the final signature encoding step. The existing implementation computes all the values needed for rejection1 and rejection2 and then checks them at once. We performed `Sign` 100,000 times to measure the occurrence count of each condition, and confirmed that rejection1 occurs about 4.4 times more often. We therefore perform the rejection1 check first, and when rejection1 occurs we omit the computation for rejection2. Both conditions are judgments on the same value, and the values used in each step are generated together during the initial hyperball sampling, so early rejection does not affect the random sequence. Therefore, the set of finally accepted signatures is preserved regardless of the evaluation order.

2) Static-Inline Conversion of Reduction and Decomposition Functions: Reduction functions such as `freeze`,

`freeze2q`, and `reduce32_2q` and decomposition functions such as `decompose_hint` and `decompose_vk` are small functions called inside loops that iterate over the $n = 256$ coefficients of a polynomial. For example, `Verify` invokes them $k\ell n$ times in total (`freeze`), $2kn$ times (`freeze2q`, `reduce32_2q`), and kn times (`decompose_hint`). The existing implementation places these functions in a compilation unit separate from the calling code, so they cannot be inlined without link-time optimization, and the per-call branch, register save/restore, and calling-convention spills accumulate thousands of times. We move them into a header and convert them to static inline; as a result, the compiler hoists the Barrett constant loads out of the loop and eliminates the spills forced by the calling convention, reducing by 14–16 cycles per coefficient on the `Verify` path. We apply the same conversion to the identical paths of `KeyGen` and `Sign` that run repeatedly inside the rejection loop and to the fixed-point accumulation function of hyperball sampling.

3) Assembly Porting of Frequently Called Functions: The two operations with high call frequency are ported to assembly because the C implementation itself is the source of their cost. The first is the fixed-point multiplication `fixpoint_mul` of hyperball sampling, which generates coordinate coefficients with 96-bit arithmetic having a 76-bit fractional part; it is called per coefficient and thus runs at least $n(\ell + k)$ times per sampling. Using the `umull`/`umlal` instructions and FP registers for this operation, we reduce the per-call cost from 255 cycles to 198 cycles, a 22.4% reduction. The second is the basemul `poly_pointwise_montgomery` repeated in the matrix product $\mathbf{A_0 s_1}$ of `KeyGen`; we replace the C loop, which made a per-coefficient call to a Montgomery reduction that the compiler could not inline, with assembly that loads two coefficients at once with `ldrd` and places the reduction inline between the multiplications, reducing the cost from 5,194 cycles to 3,281 cycles per basemul, a 36.8% reduction. The matrix product of `Verify` already uses a C implementation with inline reduction, so it yields no additional gain; the benefit of this porting is realized on the paths where the compiler could not inline the reduction.

VI. Optimized Implementation of SMAUG-T on Cortex-M4

This section presents strategies for optimizing SMAUG-T on Cortex-M4. We address polynomial multiplication, SHA-3, and sampling, which are the dominant operations across `KeyGen`, `Encaps`, and `Decaps`. We first derive the range of the auxiliary modulus required to apply NTT-based polynomial multiplication to SMAUG-T, and by comparing the Toom-Cook/Karatsuba approach of the reference implementation with NTT-based multiplication, we conclude that using the NTT is more efficient. We then propose an optimal NTT implementation methodology for SMAUG-T on Cortex-M4.

A. Modulus Selection for NTT-Based SMAUG-T on Cortex-M4

SMAUG-T operates over the NTT-unfriendly ring $R_q = \mathbb{Z}_q[x]/(x^n + 1)$ with a power-of-two modulus q. According to prior work, even for an algorithm that uses an NTT-unfriendly ring, NTT-based polynomial multiplication can be applied by introducing an auxiliary NTT-friendly modulus q' and carrying out the computation over $R_{q'} = \mathbb{Z}_{q'}[x]/\{x^n+1\}$, so that the polynomial multiplication result in the original ring can be recovered exactly [27], [28]. The key condition is that every coefficient of the matrix-vector multiplication, computed over the integers, must lie in the centered interval $(-q'/2,\, q'/2)$, so that the original product is recovered exactly from the result modulo q'. Assuming that both operands are full-width polynomials with coefficients in $[0, q)$, the lower-bound requirement on q' is as follows, where k denotes the matrix dimension, n the polynomial degree, and q the original modulus.

$$q' > 2 \times k \times n \times \frac{q}{2} \times \frac{q}{2}. \tag{2}$$

However, in SMAUG-T the two operands of a polynomial multiplication do not have symmetric coefficient ranges. While $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{c}_1$ have full-width coefficients in R_q, the secret vector $\mathbf{s}$ and the ephemeral vector $\mathbf{r}$ have small coefficients satisfying $|s_i|, |r_i| \leq \eta$. Exploiting this asymmetry, the required range can be tightened further as follows.

$$q' > 2 \times k \times n \times \frac{q}{2} \times \eta. \tag{3}$$

SMAUG-T uses $n = 256$ at all security levels. SMAUG-T128 at security level 1 ($q = 1{,}024$, $k = 2$, $\eta = 1$) requires $q' > 524{,}288$, so q' cannot be a 16-bit modulus.

Since the Hamming weight of the secret vector $\mathbf{s}$ is fixed at h_s, a tighter upper bound can be computed for multiplications involving $\mathbf{s}$ as follows. For SMAUG-T128 ($q = 1{,}024$, $h_s = 140$, $\eta = 1$):

$$q' > 2 \times h_s \times \frac{q}{2} \times \eta. \tag{4}$$

Even with this tighter upper bound, $q' > 143{,}360$ is required, so q' still cannot be a 16-bit modulus. If a 16-bit q' could be chosen, it would be advantageous on Cortex-M4. The SIMD instructions of Cortex-M4 can process packed 16-bit data, so using a 16-bit modulus allows two coefficients to be processed in parallel, as in Kyber [14]. However, under this straightforward analysis, no valid 16-bit q' exists for any SMAUG-T parameter set. Therefore, because q' cannot be a 16-bit modulus and must instead be represented in a 32-bit register at all security levels, choosing a common auxiliary modulus q' based on the highest security level allows the same code to be reused. This simplifies the software implementation and reduces the code size. Accordingly, we select the auxiliary modulus based on SMAUG-T256. This choice is also advantageous for hardware design, since a single NTT module can be reused across all security levels.

Two approaches are known for implementing the NTT in an NTT-unfriendly algorithm. The first is the single-large-modulus approach, which performs the entire computation over one NTT-friendly modulus q'. The second is the multi-moduli approach, which uses q'_1 and q'_2 satisfying $q'_1 q'_2 > q'$, performs NTT/iNTT/basemul operations for each modulus, and then reconstructs the result via the CRT. The *multi-moduli* approach can leverage SIMD instructions because it

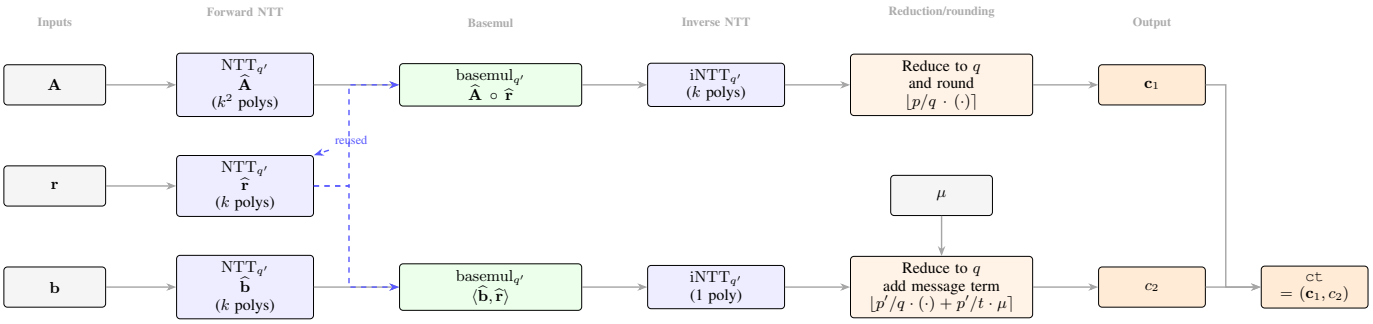


Fig. 9. NTT-based computation flow of Encaps. The dashed arrows indicate that $\hat{\mathbf{r}}$ is computed once and then reused in both the matrix-vector product $\mathbf{A} \cdot \mathbf{r}$ and the inner product $\langle \mathbf{b}, \mathbf{r} \rangle$.

uses smaller moduli, but the overhead of CRT reconstruction and two NTT/iNTT pairs offsets this benefit. Prior studies of Cortex-M4 NTT implementations have likewise reported that, under the same conditions, two 16-bit NTTs are slower than a single 32-bit NTT [14], [15], [27]. Accordingly, we adopt a 32-bit NTT using the single-large-modulus approach.

B. NTT Application Scenario

To evaluate the efficiency of NTT-based multiplication in SMAUG-T, we must consider not the cost of a single polynomial multiplication but the cumulative number of NTT, iNTT, and basemul operations required across the entire KEM computation. In NTT-friendly schemes such as ML-DSA [9], the public matrix $\mathbf{A}$ is sampled directly in the NTT domain, so its forward transform can be omitted. In contrast, because SMAUG-T is NTT-unfriendly by design, all operands must be transformed into the NTT domain to perform NTT-based multiplication.

TABLE V
NUMBER OF NTT PRIMITIVES FOR THE TWO CORE OPERATIONS

Operation	NTT	iNTT	basemul
MatrixVectorProd	$k^2 + k$	k	k^2
InnerProd	$2k$	1	k

KeyGen computes $\mathbf{b} = -\mathbf{A}^\top \mathbf{s} + \mathbf{e}$. Adding the error vector $\mathbf{e}$ in the integer domain after the inverse transform avoids a forward transform of $\mathbf{e}$. Thus, the NTT cost of KeyGen is that of a matrix-vector multiplication: $k^2 + k$ forward NTTs for $\mathbf{A}$ and $\mathbf{s}$, k^2 basemul operations, and k iNTTs for the output vector. Encaps computes $\mathbf{c}_1 = \mathbf{A} \cdot \mathbf{r}$ and $c_2 = \langle \mathbf{b}, \mathbf{r} \rangle$. Hence, the NTT cost is one matrix-vector product and one inner product. Since the same ephemeral vector $\mathbf{r}$ is used in both the matrix-vector product and the inner product, computing $\hat{\mathbf{r}} = \mathrm{NTT}(\mathbf{r})$ only once and reusing it in both branches saves k forward transforms. Decaps computes $\langle \mathbf{c}_1, \mathbf{s} \rangle$ and thus performs one inner product. This requires $2k$ forward NTTs for $\mathbf{c}_1$ and $\mathbf{s}$, k basemul operations, and one iNTT for the output polynomial. Fig. 9 shows the NTT-based computation flow of Encaps. Each operand involved in the matrix-vector product or inner product ($\mathbf{A}$, $\mathbf{r}$, and $\mathbf{b}$) is transformed into the NTT domain, after which the basemul operations are performed. Because $\mathbf{r}$ is common to both operations, its NTT-domain representation is reused. The matrix-vector product and inner product are obtained by applying the iNTT after the basemul and then reducing the result modulo the original q. Since q is a power of two, this modular reduction can be implemented with a simple shift. NTT-based polynomial multiplication can be applied similarly to KeyGen and Decaps.

C. Toom-Cook vs. NTT-Based Polynomial Multiplication

The polynomial multiplication of the SMAUG-T reference C implementation adopts a hybrid Toom-Cook/Karatsuba approach [10]. For a degree-256 polynomial, each input polynomial is split into four blocks of 64 coefficients, and a 4-way Toom-Cook evaluation is performed. The resulting seven partial products, one per evaluation point, are computed via Karatsuba, and this function in turn combines schoolbook multiplications over units of 16 coefficients in a Karatsuba manner. The intermediate result of 512 coefficients obtained by interpolation is then reduced modulo $x^{256}+1$ by subtracting the upper half from the lower half, and the results are accumulated. Earlier versions of SMAUG-T used sparse multiplication for polynomial multiplication. Because this multiplication used the positions of secret coefficients as memory indices, it was reported to be vulnerable to information leakage through a timing channel and has been removed from the latest SMAUG-T reference C implementation.

We conducted experiments to evaluate whether NTT-based multiplication is a competitive alternative for SMAUG-T on Cortex-M4. Specifically, we compared the SMAUG-T reference C implementation, which uses Toom-Cook-based multiplication, with a version that uses NTT-based multiplication. This comparison is intended to isolate the algorithmic advantage of the NTT-based approach under the same C-level conditions. Starting from the reference C implementation, we replaced Toom-Cook with the NTT only for matrix-vector products and inner products. We used the Dilithium prime 8 380 417, which satisfies the q' condition at all security levels, and adopted the corresponding NTT implementation from [29]. We benchmarked both implementations using the pqm4 [2] framework. Specifically, we integrated them into the crypto_kem benchmark harness and ran each parameter set 100 times on a NUCLEO-L4R5ZI (STM32L4R5ZI, Cortex-M4 @20 MHz) board. We compiled the implementations with GCC 10.3.1 (arm-none-eabi) using the -O3 optimization flag

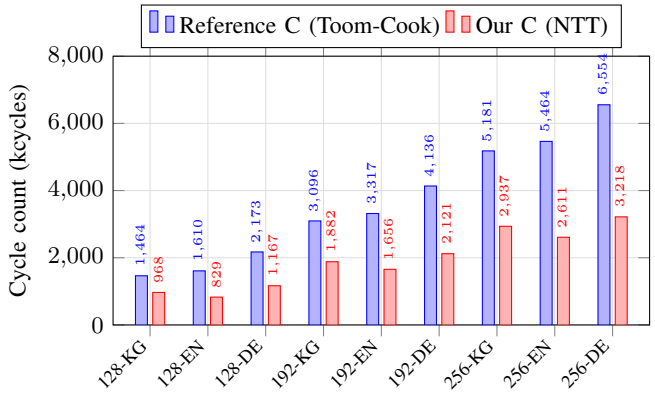


Fig. 10. Comparison of average cycle counts per SMAUG-T KEM operation (100 iterations). KG, EN, DE denote `KeyGen`, `Encaps`, and `Decaps`, respectively.

from the pqm4 speed build configuration. All 100 runs for every combination of parameter set and implementation passed the `OK KEYS` check.

Fig. 10 compares the benchmark results for Toom-Cook/Karatsuba multiplication and NTT-based multiplication. The NTT-based implementation outperforms the reference Toom-Cook/Karatsuba implementation at all security levels. For SMAUG-T128, the `KeyGen`, `Encaps`, and `Decaps` cycle counts decrease from 1,464k to 968k, from 1,610k to 829k, and from 2,173k to 1,167k, respectively. For SMAUG-T192, they decrease from 3,096k to 1,882k, from 3,317k to 1,656k, and from 4,136k to 2,121k, respectively, and for SMAUG-T256, from 5,181k to 2,937k, from 5,464k to 2,611k, and from 6,554k to 3,218k, respectively. This corresponds to a performance improvement of up to 109.3%. Although Toom-Cook reduces the number of coefficient multiplications, its evaluation and interpolation stages incur substantial addition/-subtraction and irregular memory-access overhead. In contrast, NTT multiplication has a regular butterfly structure that is well suited to in-place computation and effectively distributes load/store costs across the in-order execution pipeline of Cortex-M4. Therefore, we conclude that NTT-based polynomial multiplication is advantageous for SMAUG-T on Cortex-M4.

D. NTT Layer Selection

The choice of the number of NTT layers entails a trade-off between butterfly cost and basemul cost. A complete NTT minimizes the degree of the base polynomials in the basemul stage, but it requires more butterflies in the NTT and iNTT, thereby increasing their cost. Conversely, an incomplete NTT reduces butterfly overhead but increases the complexity of the basemul. Considering this trade-off, we search for the optimal number of NTT layers for NTT-based polynomial multiplication in SMAUG-T on Cortex-M4.

In this paper, addition, subtraction, multiplication, multiply-accumulate, and reduction are denoted by A, S, M, MA, and R, respectively. Load and store are denoted by Ld and St, respectively. For the NTT transform, one layer consists of $n/2$ butterflies. Since the cost of one butterfly is $1A+1S+1M+1R$, the cost of an NTT performing L layers is as follows.

$$C_{\mathrm{NTT}}(L) = L \cdot \frac{256}{2}\,(1A + 1S + 1M + 1R)\,. \tag{5}$$

For the basemul, let the size of each base block be m. An $m \times m$ schoolbook multiplication is performed for each of the 2^L base blocks over the following ring [14].

$$\mathbb{Z}_{q'}[X]/(X^m - \psi). \tag{6}$$

In the product of two polynomials $a(X) = \sum_{j=0}^{m-1} a_j X^j$ and $b(X) = \sum_{j=0}^{m-1} b_j X^j$, the output coefficient c_i splits into a low-degree term and a high-degree term reduced by $X^m = \psi$.

$$c_i = \sum_{j=0}^{i} a_j b_{i-j} + \psi \sum_{j=i+1}^{m-1} a_j b_{m+i-j}, \quad 0 \le i < m. \tag{7}$$

In [28], an improved accumulation scheme for schoolbook multiplication in the 4×4 case was introduced. This method does not perform a reduction at every multiplication; instead, it first accumulates the high part, performs a Montgomery reduction, multiplies the result by ψ, then accumulates all of the low part and performs a reduction. This approach reduces the number of reductions and efficiently computes low-degree polynomial multiplication on Cortex-M4, which provides a multiply-accumulate capability.

The cost of an $m \times m$ schoolbook multiplication, accounting for the fact that the last coefficient c_{m-1} has no high part, can be generalized as follows.

$$\begin{aligned} C_{\mathrm{SB}}(m) = {} & 1M + (m-1)MA + 1R \\ & + \sum_{i=0}^{m-2} \Bigg[\underbrace{(1M + (m-2-i)MA + 1R)}_{\text{high part}} \\ & \qquad + \underbrace{(1M + (i+1)MA + 1R)}_{\text{low part}} \Bigg]. \end{aligned} \tag{8}$$

After L layers, each base block has size $m = n/2^L$, and there are 2^L blocks in total. An L-layer basemul therefore performs one such schoolbook multiplication per block, so its cost is as follows.

$$C_{\mathrm{basemul}}(L) = 2^L\, C_{\mathrm{SB}}\big(n/2^L\big)\,. \tag{9}$$

To select the optimal L, we compare the total costs of the matrix-vector product and inner product as functions of L. Assuming that the NTT and iNTT have equal costs, the respective costs of these two operations with L layers are as follows.

$$\begin{aligned} C_{\mathrm{MatVec}}(L,k) &= (k^2+2k)C_{\mathrm{NTT}}(L) + k^2 C_{\mathrm{basemul}}(L), \\ C_{\mathrm{Inner}}(L,k) &= (2k+1)C_{\mathrm{NTT}}(L) + kC_{\mathrm{basemul}}(L). \end{aligned} \tag{10}$$

Table VI expands these costs for 5–8 layers. As shown in Table VI, reducing L decreases the NTT and iNTT costs but increases the basemul cost. Conversely, increasing L decreases the basemul cost but increases the NTT and iNTT costs.

TABLE VI
MATRIX-VECTOR PRODUCT AND INNER PRODUCT COST ACCORDING TO THE NUMBER OF LAYERS AT $n = 256$

Layer L	Matrix-vector product cost	Inner product cost
5	$(640k^2 + 1280k)(A + S)$ $+ (1120k^2 + 1280k)(M + R) + 1792k^2MA$	$(1280k + 640)(A + S)$ $+ (1760k + 640)(M + R) + 1792kMA$
6	$(768k^2 + 1536k)(A + S)$ $+ (1216k^2 + 1536k)(M + R) + 768k^2MA$	$(1536k + 768)(A + S)$ $+ (1984k + 768)(M + R) + 768kMA$
7	$(896k^2 + 1792k)(A + S)$ $+ (1280k^2 + 1792k)(M + R) + 256k^2MA$	$(1792k + 896)(A + S)$ $+ (2176k + 896)(M + R) + 256kMA$
8	$(1024k^2 + 2048k)(A + S)$ $+ (1280k^2 + 2048k)(M + R)$	$(2048k + 1024)(A + S)$ $+ (2304k + 1024)(M + R)$

On Cortex-M4, A, S, M, and MA each execute in a single cycle. For R, signed Montgomery reduction, a state-of-the-art 32-bit implementation, can be performed in two cycles using `mul` and `smlal` [14], [30]. With these weights, for matrix-vector multiplication the 5-layer choice costs about 1.6% less than the 6-layer choice when $k = 2$, whereas the 6-layer choice is about 0.6% and 2.0% more efficient when $k = 3$ and $k = 4$, respectively. For the inner product, the 5-layer choice costs about 4.1%, 3.4%, and 3.0% less than the 6-layer choice when $k = 2$, $k = 3$, and $k = 4$, respectively. In every case, the basemul savings of the 7-layer and 8-layer choices are insufficient to offset their increased NTT costs. Moreover, because at most three layers can be merged, the 7-layer and 8-layer choices also incur higher memory-access costs. The remaining candidates are therefore 5 and 6 layers, whose costs differ by less than 5% in every case.

However, the 5-layer choice incurs additional basemul overhead because of its higher register pressure. Its basemul uses 8×8 schoolbook multiplication, for which it is difficult to keep all 16 coefficients of the two input blocks in the Cortex-M4 integer registers throughout the computation. For example, suppose that $c_0, \ldots, c_7$ are computed sequentially while $a_0, \ldots, a_7$ remain in registers and $b_0, \ldots, b_7$ are moved from memory or FPU (Floating Point Unit) registers to general-purpose registers as needed. Each 8×8 schoolbook multiplication then incurs 64 `vmov` or `ldr` instructions. Consequently, every 5-layer basemul incurs an additional cost of $64 \times (256/8) = 2{,}048$ such instructions, making the 6-layer NTT faster overall. Therefore, we use $q' = 25\,570\,049$ and a 6-layer NTT.

E. Optimization techniques

Modular Arithmetic. Montgomery multiplication has been widely known as the most efficient method for modular arithmetic, and Alkim et al. proposed a 3-cycle signed Montgomery multiplication and 5-cycle 32-bit CT butterfly on Cortex-M4 using the accumulated multiplication instruction `smlal` [30]. Recently, Huang et al. introduced Plantard multiplication, which can be computed in 2 cycles on Cortex-M4 [15]. Although the proposed implementation uses 16-bit arithmetic, it can be extended to 32-bit arithmetic by increasing the word size. However, 32-bit Plantard arithmetic requires 64-bit words, which are inefficient on the 32-bit Cortex-M4 architecture. We therefore adopt the signed Montgomery multiplication-based 32-bit CT butterfly of Alkim et al. [14], [30], which executes in 5 cycles on Cortex-M4.

NTT Reuse. During encapsulation, $\mathbf{c}_1$ is computed as $\mathbf{A} \cdot \mathbf{r}$, while c_2 is computed as $\langle \mathbf{b}, \mathbf{r} \rangle$. When NTT-based multiplication is used, a forward transformation must be applied to both $\mathbf{A}$ and $\mathbf{r}$ to obtain $\hat{\mathbf{A}}$ and $\hat{\mathbf{r}}$, and likewise to $\mathbf{b}$ and $\mathbf{r}$ to obtain $\hat{\mathbf{b}}$ and $\hat{\mathbf{r}}$. The value $\hat{\mathbf{r}}$ does not need to be recomputed; the $\hat{\mathbf{r}}$ obtained during the computation of $\mathbf{c}_1$ can be directly reused for the computation of c_2. Therefore, k forward transformations can be saved.

Simplified CT butterfly. In SMAUG-T, the secret vectors $\mathbf{s}$ and $\mathbf{r}$ are composed of ternary polynomials, where each coefficient is either -1, 0, or 1. As a result, the first layer CT butterfly in the NTT for $\mathbf{s}$ and $\mathbf{r}$ does not require modular multiplication. This eliminates 128 modular multiplications in the first layer and reduces a total of $128k$ modular multiplications.

SHA-3 and SHAKE. The XKCP GitHub repository provides optimized implementations of SHA-3 and SHAKE for various platforms including Cortex-M4 [31]. Furthermore, Huang et al. proposed faster SHA-3 and SHAKE implementations on Cortex-M4 by improving instruction scheduling, and these are now integrated into pqm4 [25]. We use these implementations.

VII. PERFORMANCE ANALYSIS

A. Benchmarking Setup

To ensure a fair comparison with NIST PQC, we measure performance on Cortex-M4 using the pqm4 framework, which counts cycles with the SysTick cycle counter of the HAL. We integrate our optimized implementation into the pqm4 framework and utilize its benchmarking functionality for performance evaluation. While the original target board of pqm4 was the `STM32F4DISCOVERY` running at 24 MHz [2], it has recently been updated to use the `NUCLEO-L4R5ZI` board running at 20 MHz [3]. We follow this update and conduct our measurements using the `NUCLEO-L4R5ZI` board at 20 MHz. Since the latest pqm4 paper using this board reports performance only for additional round DSAs [3], we directly measured the performance of some algorithms ourselves (HAETAE, SMAUG-T, AIMer, and HQC), while citing the rest (ML-DSA, Falcon, MAYO, PERK, and ML-KEM) from the pqm4 project [32].

TABLE VII
PER-CALL CYCLE COUNTS (CC) FOR THE $\mathcal{N}(\mathbf{s})$ EVALUATION, THE $c\mathbf{s_1}, c\mathbf{s_2}$ MULTIPLICATION, AND THE SAMPLER OF HAETAE ON CORTEX-M4

Algorithm	Implementation	$\mathcal{N}(\mathbf{s})$ cc	$c\mathbf{s_1}, c\mathbf{s_2}$ cc	sampler cc
HAETAE-120	Cheon et al. [12]	914 139	149 473	
HAETAE-180		1 448 555	230 020	206[a]/207[b]
HAETAE-260		1 804 492	283 689	
HAETAE-120	This Work	114 409	101 268	
HAETAE-180		134 915	150 437	198[a]/199[b]
HAETAE-260		199 617	187 913	

[a]: `CDT sampler`, [b]: `approx_exp`

The `KeyGen` and `Sign` of HAETAE contain a rejection loop, so the cycle count of a function is proportional to the number of rejections that occur in that execution. Since this rejection count is determined by chance for each seed, directly comparing the medians of two implementations measured with different random seeds may distort the comparison merely because of an incidental difference in the rejection-count samples. As Table VII shows, even though our implementation always outperforms the existing one in the per-call cycles of the core kernels inside `KeyGen` and `Sign`, we experimentally confirmed cases where this advantage is reversed under a naive (unconditioned) median comparison. Therefore, for each measurement we also record the rejection count at that time, collect the cycles of the executions in which the number of rejections equals the expected number prescribed by the specification, and report their median as the performance. Specifically, `KeyGen` uses executions with 10 attempts corresponding to a key acceptance rate of 0.1, i.e., 9 rejections [12], and `Sign` uses executions with $M-1$ rejections corresponding to the expected repetition $M = 6.0/5.0/6.0$ for security levels 1/3/5, respectively. For `Verify`, which has no rejection, we report the median of all measurements. To secure a sufficient sample size, HAETAE measurements are performed with 1,000 repetitions per security level.

The pqm4 framework does not support benchmarking at the level of individual functions. Therefore, we measure the performance of internal functions through a separate measurement harness that uses the same board, the same clock, and the same cycle-measurement tool. We provide identical inputs to the implementations under comparison and adopt the modal (most frequent) value of 10 repeated measurements after one warmup. Since a single-call measurement is inaccurate for internal sampler functions that take on the order of a few hundred cycles per call, we measure them as the average per-call cycles obtained by subtracting the cycles of an empty loop from those of a 4,096-iteration loop and dividing by the iteration count.

In this paper, the performance improvement of the optimized implementation is reported as a percentage. The improvement is computed as $(C_{\text{ref}} - C_{\text{opt}})/C_{\text{opt}} \times 100$ based on the median cycles of the existing implementation and the optimized implementation, where C_{ref} and C_{opt} are the median cycles before and after optimization, respectively. In contrast, the performance gap relative to other algorithms is expressed as a factor ($\times$).

B. Performance of HAETAE on Cortex-M4

For performance comparison, we use the HAETAE implementation in pqm4 [32], which includes the Cortex-M4 implementation of HAETAE with the best-known performance introduced in [12] and the state-of-the-art SHAKE implementation [25]. The optimization techniques applied in this implementation are described in Section III-A. Table VII compares the cycles of the $\mathcal{N}(\mathbf{s})$ evaluation, the $c\mathbf{s_1}, c\mathbf{s_2}$ multiplication of `Sign`, and the two internal sampler functions. In the existing implementation, the $\mathcal{N}(\mathbf{s})$ evaluation is performed with a 512-point FFT repeated for each secret-key polynomial, whereas our implementation applies ASNE (Section IV-B) to perform only an autocorrelation NTT and a single 256-point FFT, achieving a per-call performance improvement of up to +973.7%. HAETAE-120/180 reuse the NTT($\mathbf{s_1}$) values computed during `KeyGen`, while HAETAE-260 performs NTT($\mathbf{s_1}$). The $c\mathbf{s_1}, c\mathbf{s_2}$ multiplication is compared as the sum of single calls of the three functions involved in the $\mathbf{z}$ computation: the NTT transforms of $\mathbf{s_1}$, $\mathbf{s_2}$, and the challenge c. Our small-NTT path shows an improvement of up to +52.9% over the existing full-NTT path. These sampler figures reflect the performance after pipelining improvements to the `CDT sampler` and `approx_exp`, each of which improves by +4.0%.

Table VIII presents the performance and stack memory usage of PQC DSAs on Cortex-M4. The comparison includes the NIST standardized DSAs, the NIST Additional Round 2 DSAs, and the KpqC DSAs provided in pqm4. The performance results of ML-DSA, Falcon, MAYO, and PERK are taken from the `benchmark.md` results of commit `90bfb63` in pqm4 [32]. For AIMer, we directly measured the performance of the state-of-the-art implementation [33]. For HAETAE, we directly measured the performance of both the existing implementation and our optimized implementation. As described in Section VII-A, the performance of HAETAE is the median of executions with 9 rejections for `KeyGen` and with $M-1$ rejections for `Sign`, while `Verify` is the median of all measurements. Our optimized implementation on Cortex-M4 achieves performance improvements of 227.0% (230.6%, 309.1%) for `KeyGen`, 4.6% (4.6%, 4.8%) for `Sign`, and 5.5% (5.4%, 6.0%) for `Verify` over the existing implementation [12] for security levels 1 (and 3, 5), respectively. The improvement in `Verify` results from the static-inline conversion of the reduction and decomposition functions described in Section V-D2. Moreover, updating the FFT to the latest implementation reduces the overall memory usage of `KeyGen` while simultaneously achieving a substantial performance improvement. However, for HAETAE-180 the stack overhead required for NTT reuse in ASNE is larger, so its stack usage increases slightly by 612 bytes. Regarding code size, our optimized implementation uses 52 954 to 53 674 bytes of `.text`. This is 9.8 to 10.8% (4 718 to 5 242 bytes) larger than the existing Cortex-M4 implementation [12], as the additional assembly kernels increase the code size in exchange for reducing cycles. The resulting code size amounts to only 2.6% of the 2 MB flash on the target `NUCLEO-L4R5ZI` board.

When comparing HAETAE with ML-DSA, the performance

TABLE VIII
CYCLE COUNTS (kcycles) AND STACK USAGE (bytes) FOR KEYGEN, SIGN, AND VERIFY OF ALL SECURITY LEVELS OF POST-QUANTUM SIGNATURE SCHEMES ON CORTEX-M4. NUMBERS IN PARENTHESES DENOTE BENCHMARK REPETITIONS. HAETAE CYCLE COUNTS ARE MEDIANS, WITH KEYGEN AND SIGN CONDITIONED ON THE EXPECTED NUMBER OF REJECTIONS; ALL OTHERS ARE UNCONDITIONED AVERAGES

Algorithm	Parameter	Imp.	Performance			Stack Usage		
			KeyGen kcycles	Sign kcycles	Verify kcycles	KeyGen bytes	Sign bytes	Verify bytes
ML-DSA [32] (1 000)	ml-dsa-44	m4f	1 426	3 943	1 422	38 296	44 816	8 912
		m4stack	1 799	12 134	3 242	4 408	5 080	2 712
	ml-dsa-65	m4f	2 516	6 193	2 416	60 824	68 872	9 888
		m4stack	3 413	24 422	5 732	4 408	6 616	2 712
	ml-dsa-87	m4f	4 276	7 947	4 193	97 688	107 892	12 060
		m4stack	5 821	33 358	9 912	4 408	8 144	2 720
Falcon [32] (10)	fndsa-provisional-512	m4f	67 693	22 470	397	14 348	41 952	2 976
	fndsa-provisional-1024	m4f	308 609	48 321	794	27 772	81 992	5 024
MAYO [32] (10)	mayo1	m4f	9 573	16 860	11 032	179 276	251 204	253 484
	mayo2	m4f	9 575	11 007	6 028	169 548	168 700	159 500
	mayo3	m4f	27 170	45 518	28 447	499 508	592 012	583 212
AIMer [33] (10)	aimer128f	m4speed	490	28 590	26 751	8 720	123 376	15 448
		m4stack	490	46 471	26 744	8 720	13 864	15 440
	aimer128s	m4speed	490	277 354	225 137	8 720	166 204	34 624
		m4stack	490	372 459	225 134	8 720	27 324	34 624
	aimer192f	m4speed	1 193	80 107	75 084	19 184	273 104	31 608
		m4stack	1 193	116 330	75 138	19 184	29 976	31 608
	aimer192s	m4speed	1 193	763 113	617 799	19 184	349 788	60 384
		m4stack	1 193	904 631	617 835	19 184	43 220	60 384
	aimer256f	m4speed	2 900	160 891	166 072	50 032	573 176	78 608
		m4stack	2 900	228 360	166 042	50 032	77 056	78 568
	aimer256s	m4speed	2 900	1 391 296	1 321 985	50 032	615 856	116 936
		m4stack	2 900	1 712 314	1 322 034	50 032	76 032	116 936
PERK [32] (10)	perk-128-fast-3	m4	644	246 017	81 025	7 696	24 020	20 684
	perk-128-fast-5	m4	817	237 885	78 109	9 032	25 160	21 752
	perk-128-short-3	m4	642	1 346 726	460 140	7 696	27 780	25 228
	perk-128-short-5	m4	817	1 249 408	428 017	9 032	28 616	26 056
	perk-192-fast-3	m4	1 491	578 040	194 444	14 952	47 720	41 352
	perk-192-fast-5	m4	1 804	560 780	186 978	16 864	48 776	42 416
	perk-192-short-3	m4	1 491	3 287 248	1 117 626	14 952	51 344	46 656
	perk-192-short-5	m4	1 804	3 055 641	1 098 103	16 864	51 888	47 256
	perk-256-fast-3	m4	2 579	1 180 043	418 171	25 504	80 304	69 912
	perk-256-fast-5	m4	3 067	1 132 866	398 350	28 064	80 896	70 640
	perk-256-short-3	m4	2 585	6 776 857	2 264 610	25 504	82 264	74 784
	perk-256-short-5	m4	3 080	6 207 736	2 452 602	28 064	82 056	74 808
HAETAE (1 000)	HAETAE-120	m4f	11 661	24 403	977	19 788	55 628	23 296
		m4f (this work)	**3 566**	**23 327**	**926**	**18 816**	**55 724**	**23 328**
	HAETAE-180	m4f	19 041	31 176	1 868	29 484	83 436	31 776
		m4f (this work)	**5 759**	**29 810**	**1 772**	**30 096**	**79 348**	**31 916**
	HAETAE-260	m4f	21 264	45 729	2 482	34 084	103 988	37 292
		m4f (this work)	**5 197**	**43 636**	**2 341**	**29 016**	**103 988**	**37 224**

gap in KeyGen, which was from 4.97× to 8.18× in the existing implementation, has been reduced to from 1.22× to 2.50× with our optimization. In Sign, HAETAE still requires up to 5.92× as many cycles as ML-DSA due to the use of hyperball sampling, but HAETAE offers advantages in parameter size. Compared to another lattice-based DSA, Falcon, HAETAE is up to 59.38× faster in KeyGen, achieves similar performance in Sign, and Falcon is up to 2.95× faster in verification. When compared with the multivariate-based MAYO, HAETAE is up to 4.72× faster in KeyGen and up to 16.05× faster in verification, while MAYO is up to 2.12× faster in Sign. For MPC-based DSAs, AIMer and PERK outperform HAETAE in KeyGen by up to 7.28×, but HAETAE is up to 155.27× faster in Sign and up to 1,047.67× faster in verification.

C. Performance of SMAUG-T on Cortex-M4

Our implementation of SMAUG-T is based on [10]. All proposed optimizations are implemented in assembly and integrated into the existing C code. Table IX compares the performance of matrix-vector product, inner product, and the SHA-3 implementation including Keccak-f1600 on Cortex-M4. For the matrix-vector product and inner product, the reference C implementation uses Toom-Cook-based multiplication. We apply NTT-based multiplication using a suitable auxiliary modulus q' for SMAUG-T on Cortex-M4, achieving up to 1333.8% improvement. In Encaps, the inner product $\langle \mathbf{b}, \mathbf{r} \rangle$ reuses $\mathbf{r}$ in the NTT domain from $\mathbf{A} \cdot \mathbf{r}$, allowing k forward transformations to be skipped, which explains why Encaps is faster than Decaps. Even compared with the existing work that uses sparse multiplication [20], our implementation

TABLE IX
CYCLE COUNTS (cc) FOR MATRIX-VECTOR PRODUCT, INNER PRODUCT, AND SHA-3 IMPLEMENTATION (GENAX) OF SMAUG-T ON CORTEX-M4

Algorithm	Implementation	$-\mathbf{A}^T \cdot \mathbf{s}$ cc	$\mathbf{A} \cdot \mathbf{r}$ cc	$\langle \mathbf{b}, \mathbf{r} \rangle$ cc	$\langle \mathbf{c}_1, \mathbf{s} \rangle$ cc	GenAx cc
SMAUG-T128	Cheon et al. [10] (Ref C)	767 778	767 791	383 890	383 893	207 958
SMAUG-T192		1 727 496	1 727 505	575 828	575 826	640 313
SMAUG-T256		3 071 036	3 071 024	767 754	767 755	1 138 034
SMAUG-T128	Narlı et al. [20]	580 446	549 855	274 931	290 227	108 995
SMAUG-T192		1 230 550	944 040	314 678	410 182	257 334
SMAUG-T256		1 867 621	1 709 880	341 973	373 543	715 513
SMAUG-T128	This work	71 047	71 038	29 560	41 500	111 089
SMAUG-T192		142 563	142 579	41 559	59 455	347 693
SMAUG-T256		238 051	238 061	53 547	77 389	617 800

achieves an improvement of up to 830.1% in polynomial multiplication. For SHA-3, we use the state-of-the-art optimized implementation on Cortex-M4 from [25]. The most significant performance cost in SHA-3 arises not from hash function calls but from extendable-output function (XOF) calls used in the sampling algorithm. Our optimization achieves up to 87.2% improvement. The existing work uses the same optimized implementation as ours, but a performance difference arises because its computational structure and parameter sizes differ.

Table X summarizes the performance and stack memory usage of PQC KEM implementations on Cortex-M4. The performance results of ML-KEM are taken from the `benchmark.md` file of pqm4 commit `90bfb63` [32]. For HQC, we directly measured the performance of the latest state-of-the-art implementation [34]. Since SMAUG-T is not included in the pqm4 project, we independently benchmarked both the reference and our optimized implementation using the pqm4 framework. For SMAUG-T, our optimized implementation outperforms the reference C implementation by 221.8% (243.2%, 266.7%) in `KeyGen`, 337.5% (335.3%, 348.9%) in `Encaps`, and 317.9% (327.7%, 341.0%) in `Decaps` at security levels 1, 3, and 5, respectively. In terms of performance, our implementation is faster than the existing implementation [20] at all security levels, with a gap of up to 201.6%. Moreover, the existing implementation uses sparse multiplication as is and is therefore a non-constant-time implementation, whereas our implementation is based on v1.2, in which this vulnerability is removed. The fact that the existing implementation exhibits lower stack usage at security levels 1 and 3 stems not from a separate optimization technique but from differences in the parameters and the secret-key encoding scheme between the versions. Regarding code size, our optimized SMAUG-T implementation uses 24 320 to 25 484 bytes of `.text`. This is 13.4 to 13.9% (2 880 to 3 076 bytes) larger than the same-specification reference C implementation, as the additional assembly kernels increase the code size in exchange for reducing cycles. The resulting code size amounts to only 1.2% of the 2 MB flash on the target `NUCLEO-L4R5ZI` board.

These improvements are mainly due to our optimizations of polynomial multiplication and SHA-3 operations, which dominate the performance cost of SMAUG-T. While the reference C implementation lags significantly behind ML-KEM, our optimized version narrows this gap and, at security level 1, `Encaps` is faster than ML-KEM-512 (m4fspeed). This is because SMAUG-T does not require error vector generation and uses an LWR system during `Encaps`, reducing the number of SHAKE-based XOF calls. In contrast, when the security level increases, the forward NTT of $\mathbf{A}$ incurs significantly more overhead than the XOF. At security level 5, ML-KEM (m4fspeed) achieves up to 1.39× faster performance in `KeyGen`, and up to 1.18× and 1.36× faster performance in `Encaps` and `Decaps`, respectively. Regarding memory usage, the ML-KEM (m4fstack) version exhibits lower stack usage than our implementation, as it applies stack-usage optimizations based on $\mathbf{A}$ on-the-fly generation. Since SMAUG-T is also designed in a module-based structure, the same technique could be applied. However, our work focuses on performance optimization without using additional stack space, and we leave this as future work. Compared to HQC, our implementation achieves superior performance across all metrics. At security level 3, our `Decaps` achieves a 32.24× speedup over HQC-192. This significant gap stems from the fundamental difference in underlying mathematical structure: HQC relies on quasi-cyclic code-based operations, including large sparse polynomial multiplications and error-correction decoding, whereas SMAUG-T's lattice-based design admits efficient NTT-based polynomial multiplication on Cortex-M4.

These results demonstrate that NTT-based polynomial multiplication, combined with optimized assembly, is superior to the reference Toom-Cook approach in both speed and stack usage on Cortex-M4.

D. Correctness and Reproducibility

We verify that our proposed optimized implementation is functionally equivalent to the existing implementation by two criteria. Since the NTT-based multiplication of SMAUG-T computes the same polynomial product as the reference Toom-Cook multiplication, all three operations `KeyGen`, `Encaps`, and `Decaps` match the reference byte-for-byte on the Known Answer Test (KAT) vectors. The `Verify` and `Sign` of HAETAE also consist only of value-preserving transformations. The small NTT recovers the coefficients exactly, and early rejection preserves the set of accepted signatures and the order of random-number consumption (Section V-D1), so for the same key and message input it produces signatures and verification results that match the reference byte-for-byte.

TABLE X
CYCLE COUNTS (kcycles) AND STACK USAGE (bytes) FOR KEYGEN, ENCAPS, AND DECAPS OF ALL SECURITY LEVELS OF POST-QUANTUM KEY ENCAPSULATION SCHEMES ON CORTEX-M4. NUMBERS IN PARENTHESES DENOTE BENCHMARK REPETITIONS

Algorithm	Parameter	Imp.	Performance			Stack Usage		
			KeyGen kcycles	Encaps kcycles	Decaps kcycles	KeyGen bytes	Encaps bytes	Decaps bytes
ML-KEM [32] (10)	ml-kem-512	m4fspeed	392	391	428	4 372	5 436	5 412
		m4fstack	392	393	430	2 300	2 348	2 332
	ml-kem-768	m4fspeed	642	659	708	5 396	6 468	6 452
		m4fstack	644	665	714	2 820	2 860	2 844
	ml-kem-1024	m4fspeed	1 019	1 032	1 094	6 436	7 500	7 484
		m4fstack	1 020	1 038	1 101	3 332	3 372	3 356
HQC [34] (10)	hqc-128	m4f	4 202	8 657	14 075	-	-	-
	hqc-192	m4f	9 771	19 846	31 170			
	hqc-256	m4f	12 758	25 794	41 868			
SMAUG-T (10)	SMAUG-T128	ref‡	1 464	1 610	2 173	10 940	12 588	13 356
		m4f† [20]	902	1 110	1 486	5 672	7 300	8 068
		m4f (this work)‡	**455**	**368**	**520**	**8 956**	**10 612**	**11 380**
	SMAUG-T192	ref‡	3 096	3 317	4 136	15 028	17 212	18 300
		m4f† [20]	1 807	1 789	2 326	10 924	13 076	14 196
		m4f (this work)‡	**902**	**762**	**967**	**13 564**	**15 748**	**16 836**
	SMAUG-T256	ref‡	5 181	5 464	6 554	20 148	22 940	24 316
		m4f† [20]	3 012	3 118	3 662	26 308	29 484	31 052
		m4f (this work)‡	**1 413**	**1 217**	**1 486**	**19 204**	**21 892**	**23 364**

†: SMAUG-T v1.1 parameters, ‡: SMAUG-T v1.2 parameters

In contrast, the `KeyGen` of HAETAE does not take byte-for-byte agreement as its correctness criterion. Since ASNE reconstructs the key-admission decision to conform to the specification's definition of $\mathcal{N}(\mathbf{s})$ more closely than the reference implementation (Section IV), it is an intended consequence of the design that the two implementations make different accept/reject decisions on some keys. Therefore, the correctness of `KeyGen` is verified not by byte-for-byte agreement with the reference but by comparison against a Q64.64 fixed-point oracle, and Table II shows that the false-accept of ASNE is lower than that of the reference.

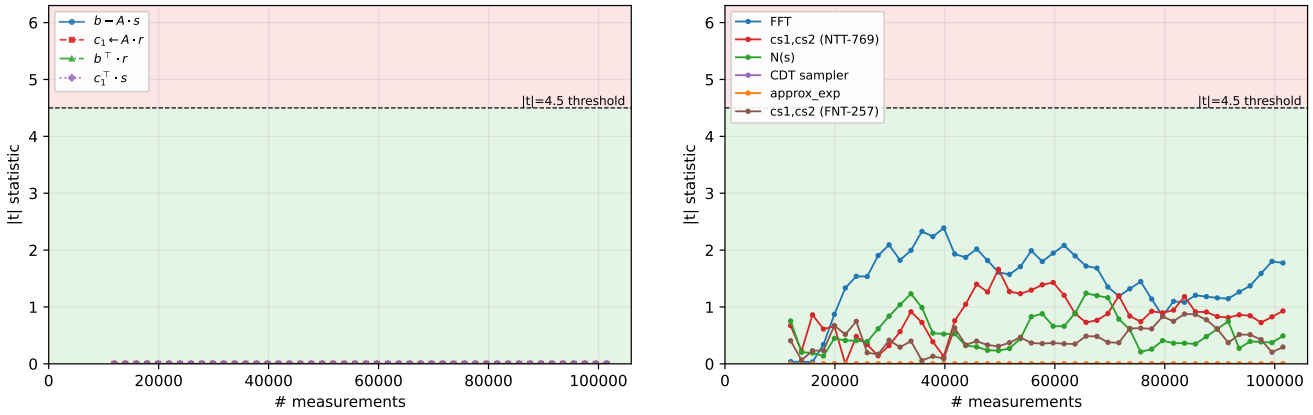
VIII. TIMING LEAKAGE ANALYSIS

This section analyzes the potential for timing leakage in the kernels of the proposed optimized implementations and evaluates whether any evidence of timing leakage is observed. A well-known technique for preventing information leakage through timing channels is a constant-time implementation. A constant-time implementation does not use secret information as a branch condition or a memory address, and it must avoid feeding secret data into instructions whose execution time depends on the operand values [35]. The proposed implementations (the NTT multiplications of SMAUG-T; the FFT, NTT-769/FNT-257, `CDT sampler`, and `approx_exp` of HAETAE) are all hand-written in assembly, and we manually reviewed their instruction streams to confirm that this policy actually holds at the instruction level.

While coding discipline and static analysis confirm structural secret-independence at the instruction level, we additionally perform dynamic analysis on the target board to independently cross-verify that this property also holds in the execution time actually observed on the Cortex-M4. This combination of static and dynamic analysis in the real execution environment has also been adopted in prior work [36], [37]. The valgrind-based dynamic analysis tools ctgrind [38] and TIMECOP [39] track secret-dependent branches and memory accesses, but because they run in a host environment, they cannot inspect the Cortex-M4-specific assembly routines. We therefore use dudect [40], which directly measures execution time on the target platform and applies a statistical hypothesis test to detect evidence of timing leakage.

dudect repeatedly measures a class 0 with a fixed secret input and a class 1 that is randomized at every measurement, interleaving them in random order, and applies Welch's t-test to the execution-time distributions of the two classes. It simultaneously runs several t-tests with different percentile-based preprocessing and reports their maximum $\max|t|$, taking $|t| > 4.5$ as strong statistical evidence of timing leakage. A result below the threshold means that there is no evidence of leakage up to the given number of measurements, but it does not constitute a formal proof of constant-time behavior. The measurements are performed using the SysTick cycle counter of the HAL.

Both HAETAE and SMAUG-T use rejection sampling; in particular, HAETAE follows the Fiat–Shamir-with-aborts paradigm, so the number of rejection-loop iterations can vary with the seed. Because this iteration count is a design element intended to make the signature distribution independent of the secret key and is not a direct function of the secret information [12], we focus constant-time verification on the operations that directly handle the secret within each iteration. Since the execution time of the whole algorithm can vary with the input seed, we perform the dudect evaluation on these operations in the proposed optimized implementations. For SMAUG-T, we measure the four NTT-based matrix–vector and vector–vector multiplications ($\mathbf{b} - A\mathbf{s}$, $A\mathbf{r}$, $\mathbf{b}^\top\mathbf{r}$, $\mathbf{c_1}^\top\mathbf{s}$). For



(a) SMAUG-T (b) HAETAE

Fig. 11. Evolution of the dudect $\max|t|$ for the internal optimized operations of HAETAE and SMAUG-T as a function of the number of measurements (fixed-vs-random, HAL SysTick cycle counter). The dashed line marks the decision threshold $|t| = 4.5$.

HAETAE, we measure the FFT on the secret key $\mathbf{s}$ and the entire $\mathcal{N}(\mathbf{s})$ evaluation in KeyGen, the $c\mathbf{s_1}, c\mathbf{s_2}$ computation in Sign covering both the NTT-769 path of HAETAE-180 and the FNT-257 path of HAETAE-{120, 260}, and the `CDT sampler` and `approx_exp` used in hyperball sampling. Each target is measured over 100,000 valid measurements under a fixed-vs-random design that takes the secret operand ($\mathbf{s}$, $\mathbf{r}$, or the sampler input) as the class.

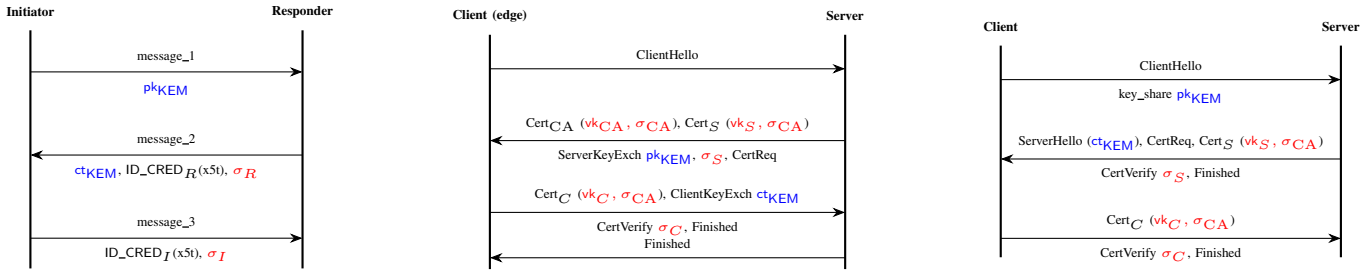
Fig. 11 shows how the per-batch $\max|t|$ of each measurement target evolves with the number of measurements. The four NTT multiplications of SMAUG-T (Fig. 11a) report $\max|t| = 0.000$ over the entire range of 100,000 valid measurements. Because the timed span runs on a fixed schedule independent of the secret, the cycle counts of the two classes coincide exactly across all measurements; in this case the variance is 0, the t-statistic itself is undefined, and the measurement harness reports it as 0.000. The `CDT sampler` and `approx_exp` of HAETAE are 0.000 throughout for the same reason, meaning that all cycle counts observed in this experiment were identical regardless of the input random values. For HAETAE (Fig. 11b), the peak $\max|t|$ values are 2.388 for the FFT, 1.662 for $c\mathbf{s_1}, c\mathbf{s_2}$ (NTT-769), 1.241 for $\mathcal{N}(\mathbf{s})$, and 0.877 for $c\mathbf{s_1}, c\mathbf{s_2}$ (FNT-257), all far below the decision threshold 4.5. These nonzero values remain below the decision threshold and are consistent with the magnitude expected under the null hypothesis rather than with leakage. Because we take the maximum over multiple correlated tests, observing a peak somewhat above 0 is normal even in the absence of leakage, and the decision criterion is whether the threshold 4.5 is exceeded. In summary, our implementations use no secret-dependent branches, no secret-dependent memory accesses, and no variable-latency instructions, and in the dudect measurements on the target board none of the targets showed any evidence of leakage up to 100,000 valid measurements. Together, these two results provide empirical evidence consistent with constant-time execution, but not a formal proof.

IX. Protocol-Level Implications of Compact Parameter Footprint

On embedded devices, the computational load of cryptographic operations is an important evaluation metric, but when a real security protocol runs on an edge node, the transmission cost of public values also governs overall performance. In some constrained wireless settings, the communication burden of enlarged PQC public values can become a major factor in protocol latency [5], [7], and in measurements of PQ-EDHOC over 6LoWPAN-over-BLE, the transmit/receive (Tx/Rx) phase accounted for 82% of the total handshake time and 74% of the total energy [41]. As shown in Fig. 1, the candidate algorithms SMAUG-T and HAETAE have smaller public values than the baseline algorithms ML-KEM and ML-DSA; this section evaluates, through an assumption-based substitution applied to prior measurements, whether these smaller public values reduce the transmission cost and whether the advantage is retained from an end-to-end perspective that accounts for the relatively large computational cost. The anchor scenarios are a communication-dominated EDHOC, a TLS 1.2 in which the DSA public values are transmitted repeatedly in the certificate chain, and a computation-dominated TLS 1.3 (Table XI). For TLS 1.3 we combine a paper that reports communication size and latency with a separate study that reports energy [41]–[44].

A. Evaluation Scenarios and Substitution Methodology

Anchors. The EDHOC (RFC 9528 [45]) method 0 anchor reports time and energy decomposed into a processing phase and Tx/Rx, and because it uses the x5t scheme that references only a thumbprint instead of the full certificate, no verification key is transmitted [41]. The TLS 1.2 [46] anchor is a mutual-authentication handshake over a two-tier PKI in which the root CA directly signs both parties' certificates, so verification keys and signatures are transmitted repeatedly, amounting to $\mathrm{bw}_{\mathrm{DSA}} = 3|\mathsf{vk}| + 5|\sigma|$, and from this fixed number of occurrences we compute the PQC public-value bytes [42]. The TLS 1.3 anchor reports bytes and latency for a wolfSSL-based handshake [43] and energy in a separate study [44]; because



(a) EDHOC method 0, mutual signature authentication with x5t thumbprint reference

(b) TLS 1.2 mutual authentication, full transmission of a two-tier PKI certificate chain

(c) The anchor study's PQ-TLS 1.3 mutual authentication flow, a single certificate on each side over a two-tier PKI

Fig. 12. Protocol communication flows of the three scenarios. Blue denotes the KEM public values, the public key $\mathsf{pk}_{\mathsf{KEM}}$ and the ciphertext $\mathsf{ct}_{\mathsf{KEM}}$; red denotes the DSA public values, the verification key vk and the signature σ, which are the objects that SMAUG-T and HAETAE replace. The diagram is schematic and shows only the messages that carry PQC public values.

TABLE XI
MEASUREMENT ENVIRONMENTS AND BASELINE ALGORITHM COMBINATIONS OF THE THREE ANCHOR SCENARIOS

	EDHOC [41]	**TLS 1.2** [42]	**TLS 1.3** [43], [44]
Device	nRF52840 (M4, 64 MHz)	nRF52840 (M4, 64 MHz)	NUCLEO-F439ZI (M4, 180 MHz)
Link	6LoWPAN/BLE	6LoWPAN/BLE	Ethernet
Tx unit	BLE LL PDU 251 B	—[b]	TCP MSS 1,460 B
Baseline KEM	ML-KEM-512	Kyber512	Kyber512
Baseline DSA	ML-DSA-44	Dilithium2	Dilithium2
Auth. model	Mutual (x5t[a])	Mutual (chain)	Mutual (1 cert. each)
Timing basis	Measured, prop.	Excluded[b]	Not reported (0)

[a] x5t: X.509 certificate thumbprint. Only a hash reference is sent instead of the full certificate, so no verification key is carried.
[b] TLS 1.2 is analyzed by bytes only; no fixed transmission unit is reported and the time totals in the anchor source are inconsistent.

it is an Ethernet link with a 0.49 ms average round-trip delay, it serves as a computation-dominated contrast case.

Substitution method. We fix the protocol structure, link conditions, and authentication model to the anchor's measured conditions, and perform an assumption-based analysis that changes only the public-value sizes of the KEM and DSA and the constrained-device-side computational cost to the SMAUG-T and HAETAE values. This assumes that the code points and usage rules for identifying and negotiating HAETAE are in place; however, this combination and these rules constitute a hypothetical configuration that is not currently registered in the IANA COSE and TLS registries [45]–[51]. The total transmission volume is computed from the number of occurrences n_x (Fig. 12) of each public value and the size difference $\Delta_x = |x_{\text{anchor}}| - |x_{\text{cand}}| \geq 0$ as

$$B_{\text{cand}} = B_{\text{anchor}} - \sum_{x \in \{\mathsf{pk},\mathsf{ct},\mathsf{vk},\sigma\}} n_x \Delta_x \qquad (11)$$

A constrained wireless link has a small link-layer transmission unit, so the protocol transcript is fragmented into several transmission units. Because the anchors report only the total transmission volume rather than per-message sizes, the transmission-unit count $\lceil B/P \rceil$ is an auxiliary metric that compares only the difference in totals under the same nominal denominator P, and it is not the actual number of BLE LL PDUs or TCP segments. Size-dependent time and energy are applied only to EDHOC, in proportion to the transmission volume, since it reports Tx/Rx separately.

The end-to-end time is computed by adding to the anchor's measured total T_{anchor} only the change induced by the value substitution, as

$$T_{\text{cf}} = T_{\text{anchor}} + \Delta T_{\text{size}} + \Delta T_{\text{comp}} \qquad (12)$$

Here ΔT_{size} is the candidate−baseline time that varies with the public-value size; it is applied to EDHOC's measured Tx/Rx in proportion to the transmission volume, and is set to 0 for TLS 1.3 because the anchor does not report a size-dependent delay separately. ΔT_{comp} is the candidate−baseline difference in cryptographic-operation time, for which we convert the Cortex-M4 cycles of Table VIII and Table X to the anchor frequency under the same rule for the baseline and the candidate. This conversion assumes that execution time is inversely proportional to the operating frequency and does not account for board-specific differences such as flash wait states. The common calibration residual is the anchor's measured total minus the baseline computation time and the size-related time; it includes non-cryptographic processing time and the peer endpoint's processing, and is held identical for the baseline and the candidate within a scenario. Therefore, the end-to-end values in this section are handshake times in which only the constrained-device-side computation is changed. Energy follows the measurement scope of each anchor study. EDHOC applies the byte ratio to the measured Tx/Rx energy and multiplies the non-transmission time, that is, the sum of the computation time and the common calibration residual, by an average processing power of 21.15 mW [41]. TLS 1.3 assumes that the board average power of 155.8 mW from the separate study [44] is constant over the entire candidate handshake time and multiplies by the total time. All end-to-end values are model estimates calibrated to the anchors.

B. Transmission Footprint and Size-Related Effects

Table XII compares the baseline and candidate bytes and the change in the transmission-unit count; for TLS 1.2, following the accounting scope of the anchor source, only the PQC public-value bytes are summed.

EDHOC. In EDHOC, where both parties authenticate each other with signatures and verification-key transmission is omitted via the x5t reference, substituting ML-KEM-512 and ML-DSA-44 with SMAUG-T and HAETAE respectively reduces the transcript from 9,780 to 7,664 bytes, a 21.6% reduction,

TABLE XII
TRANSMISSION BYTES OF THE BASELINE COMBINATIONS AND THE SMAUG-T/HAETAE CANDIDATES, AND THE TRANSMISSION-UNIT COUNT $\lceil B/P \rceil$

Scenario	P	Bytes		$\Delta(\%)$	$\lceil B/P \rceil^{\ddagger}$	
		Baseline	Candidate		Baseline	Candidate
EDHOC (ML-KEM-512/ML-DSA-44)	251 B	9,780	7,664†	−21.6	39	31
TLS 1.2 (Kyber512/Dilithium2)	—	17,604	11,690	−33.6	—	—
TLS 1.3 endpoint (Kyber512/Dilithium2)	1,460 B	14,748	11,992	−18.7	11	9
TLS 1.3 cert+CV (Kyber512/Dilithium2)	1,460 B	14,748	10,100	−31.5	11	7

‡ Comparison value under the nominal denominator P, not the actual number of PDUs or segments. † Model value with only the public values substituted; the baseline 9,780 bytes in the same row is measured.

and the transmission-unit count based on the BLE LL PDU drops from 39 to 31. This byte reduction is reflected proportionally in time and energy as well: from the measured Tx/Rx time of 2,535.07 ms and energy of 33.42 mJ, the transmission time decreases by about 548 ms and the radio energy by about 7.2 mJ (21.6%), giving candidate values of 1,986.6 ms and 26.19 mJ.

TLS 1.2. Because the entire certificate chain is transmitted, carrying three verification keys and five signatures, the HAETAE verification key, which is 320 bytes smaller than the baseline Dilithium2, accumulates three times over the chain, reducing the PQC public-value communication size by a total of 5,914 bytes (33.6%). However, since the TLS 1.2 anchor does not decompose the absolute communication time and the internal time totals in the source are mutually inconsistent, this section treats TLS 1.2 only as a byte computation of the PQC public values and excludes it from the time and energy estimates.

TLS 1.3. Because a CertificateVerify signature is verified with the public key inside the immediately preceding certificate [47], substituting this signature also requires substituting the endpoint verification key. In the endpoint-key-substitution configuration that substitutes only the two verification keys of the endpoint certificates and the two CertificateVerify signatures, the transmission size is reduced by 2,756 bytes (18.7%), and the transmission-unit count based on the MSS drops from 11 to 9. In the transmitted-certificate-reissuance configuration that also reissues the two CA signatures of those certificates, four signatures and two verification keys are substituted, reducing the size by 4,648 bytes (31.5%) and the transmission-unit count to 7. The latter requires that the root CA's HAETAE verification key be pre-provisioned as a trust anchor, and this pre-provisioned verification key is not included in the transcript bytes. Because the Ethernet anchor does not separately isolate the communication delay, the size-dependent term change is 0, and this contrast is quantified in the end-to-end analysis (§ IX-C).

C. End-to-End Latency and Energy Including Computation

Although the candidates have smaller public values, their computational cost is larger than that of ML-KEM and ML-DSA in most cases, as shown in Table VIII and Table X, so we evaluate whether the communication gain exceeds the computation loss from the end-to-end perspective of Eq. (12). The online computation of the constrained device is one KEM `KeyGen+Decaps` and one `Sign`, together with `Verify`, which is performed once in EDHOC and twice in the TLS 1.3 transmitted-certificate-reissuance configuration, where both signatures are substituted with candidates. The baseline, the existing implementation, and our implementation are all compared under the same normalization rule, and the results are presented in Table XIII.

TABLE XIII
MODEL ESTIMATES OF THE COMPUTATION AND TRANSMISSION-RELATED TIMES AND ENERGY OF THE EDHOC AND TLS 1.3 END-TO-END HANDSHAKES

Scenario	Config.	Comp. (ms)	Transfer (ms)	Total (ms)	Energy (mJ)
EDHOC	Baseline*	96.6	2,535.1	3,103.9	45.46
	Existing	453.4	1,986.6	2,912	45.8
	This work	**394.2**	**1,986.6**	**2,853**	**44.5**
TLS 1.3	Baseline*	42.3	—	96.3	15.01
	Existing	166.6	—	221	34.4
	This work	**145.3**	—	**199**	**31.1**

* Value matched to the anchor's measured total by adding a common calibration residual (EDHOC 472.2 ms, TLS 1.3 54.1 ms). For TLS 1.3 the transmission-related time is not reported separately by the anchor and is included in the residual.

In the BLE EDHOC model based on ML-KEM-512/ML-DSA-44, the transmission-related gain exceeds the computation increase, but in the TLS 1.3 model that does not include size-dependent time, the opposite holds. For the EDHOC candidate, the computation time increases by about 298 ms due to the HAETAE Sign cost, but the transmission time decreases by 548 ms, so the end-to-end time shortens from 3,103.9 ms to 2,853 ms, about 251 ms shorter. For the existing implementation, the computation-energy increase offsets the communication reduction, so the total energy of 45.8 mJ exceeds the baseline 45.46 mJ; our optimization reduces the candidate computation time by 59.2 ms, lowering it to 44.5 mJ. However, this difference of about 0.9 mJ (2.1%) is a point estimate sensitive to the processing power and to the cycle conversion, and its sign flips if the processing power exceeds 24.3 mW or the applied candidate cycles are about 11% or more larger. In contrast, for TLS 1.3 on the Ethernet link, the delay change from the byte reduction is not included in the model and only the computation increase is reflected, so the end-to-end time roughly doubles from 96.3 ms to about 199 ms.

The candidate combination gains in latency when the actual transmission rate R is lower than $R^{*} = 8\Delta B/\Delta T_{\mathrm{comp}}$, the rate at which gain and loss balance out, where R is the effective bit rate at which delay grows with the data sent, not the datasheet PHY rate, $\Delta B = B_{\mathrm{anchor}} - B_{\mathrm{cand}} > 0$, and $\Delta T_{\mathrm{comp}} > 0$. For ML-KEM-512/ML-DSA-44 EDHOC, R^{*} is about 57 kbit/s, which exceeds the anchor's measured transmission rate, so on the BLE link the small-parameter advantage of the candidate is predicted under the above model assumptions. Unlike the ephemeral KEM key, the KeyGen of the long-term DSA key is not on the online handshake path, so this paper's KeyGen improvement appears as a gain in key-reissuance scenarios, while the Sign improvement and hyperball-sampling pipelining appear as a gain in every handshake. The public-value byte reduction is computed exactly under fixed occurrence counts and encoding assumptions,

whereas the transcript and actual wire-size reduction is a model value depending on lower-layer additions. Since the latency and energy may vary with packet loss, retransmission, and CoAP block scheduling, end-to-end network measurement is left for future work.

X. Conclusion

This paper presents new performance records on the Cortex-M4 for HAETAE and SMAUG-T, the algorithms selected in the KpqC competition. For HAETAE, we restore correctness by fixing two implementation bugs in the $\mathcal{N}(\mathbf{s})$ evaluation, in which the existing implementation accepts secret keys that violate the specification's upper bound, and we accelerate KeyGen by up to 309.1% through ASNE, which restructures the per-polynomial FFTs into a single autocorrelation FFT, together with Cortex-M4-specific optimizations. For SMAUG-T, from an analysis of the coefficient ranges of the operands in its polynomial multiplication, we show that a 6-layer incomplete NTT based on a single large modulus is optimal on the Cortex-M4, accelerating Encaps by up to 348.9% over the existing Toom-Cook-based reference C implementation and making Encaps at security level 1 faster than ML-KEM. Static analysis of these optimized implementations confirms the absence of secret-dependent branches, memory accesses, and variable-latency instructions, and 100,000 dudect measurements of the optimized kernels on the target board show no evidence of timing leakage.

Furthermore, this paper quantifies the implications of the optimized computational performance and the small parameter sizes of HAETAE and SMAUG-T for real-world IoT security protocols using assumption-based analytical models anchored to prior measurements. In EDHOC, which is communication-dominated, the communication gain outweighs the increase in computation, so the handshake time is shortened by about 251 ms, and for TLS 1.2, where the verification keys and signatures are repeatedly transmitted in the certificate chain, the communication size of the PQC public values is reduced by 33.6%. In contrast, for TLS 1.3, which is computation-dominated, the communication-size effect is offset, roughly doubling both the handshake time and the energy. The slower the link, the more transmission time is saved by the smaller public values, so the benefit of HAETAE and SMAUG-T appears on constrained wireless links where the reduction in communication time outweighs the added computation time.

Directions for future work include validating the models through end-to-end protocol measurements in real network environments, and further optimizing HAETAE Sign, the main contributor to the computation-time increase. A faster Sign reduces the computation time, so the benefit appears even on faster links such as TLS 1.3, widening the range of links that benefit. This paper prioritizes speed while keeping or reducing the stack usage on most computational paths, so optimizations aimed at reducing stack usage itself are beyond the scope of this paper and are also left as future work.

References

[1] G. Alagic, J. Alperin-Sheriff, D. Apon, D. Cooper, Q. Dang, J. Kelsey, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone, "Status report on the second round of the NIST post-quantum cryptography standardization process," National Institute of Standards and Technology, Gaithersburg, MD, NIST Interagency/Internal Report (NISTIR) 8309, Jul. 2020. [Online]. Available: https://doi.org/10.6028/NIST.IR.8309

[2] M. J. Kannwischer, J. Rijneveld, P. Schwabe, and K. Stoffelen, "pqm4: Testing and benchmarking nist pqc on arm cortex-m4," *Cryptology ePrint Archive*, 2019.

[3] M. J. Kannwischer, M. Krausz, R. Petri, and S.-Y. Yang, "pqm4: Benchmarking nist additional post-quantum signature schemes on microcontrollers," *Cryptology ePrint Archive*, 2024.

[4] K. Bürstinghaus-Steinbach, C. Krauß, R. Niederhagen, and M. Schneider, "Post-quantum tls on embedded systems: Integrating and evaluating kyber and sphincs+ with mbed tls," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020, pp. 841–852.

[5] M. Schöffel, F. Lauer, C. C. Rheinländer, and N. Wehn, "On the energy costs of post-quantum kems in tls-based low-power secure iot," in *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, 2021, pp. 158–168.

[6] T. Liu, G. Ramachandran, and R. Jurdak, "Post-quantum cryptography for internet of things: a survey on performance and optimization," *arXiv preprint arXiv:2401.17538*, 2024.

[7] ——, "Towards quantum resilient iot: A backward-compatible approach to secure ble key exchange against quantum threats," in *Proceedings of the 9th ACM/IEEE Conference on Internet of Things Design and Implementation (IoTDI 2024)*. Institute of Electrical and Electronics Engineers Inc., 2024, pp. 170–180.

[8] National Institute of Standards and Technology (NIST), "FIPS 203: Module-lattice-based key-encapsulation mechanism standard," NIST, Federal Information Processing Standards Publication 203, 2024, published: 2024-08-13. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf

[9] ——, "FIPS 204: Module-lattice-based digital signature standard," NIST, Federal Information Processing Standards Publication 204, 2024, published: 2024-08-13. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf

[10] J. H. Cheon, H. Choe, J. Seo, and H. Seong, "Smaug (-t), revisited: Timing-secure, more compact, less failure," *IEEE Access*, 2024.

[11] J. Woo, J. Kim, G. H. Hong, S. Lee, M. Kim, H. Lee, and J. H. Park, "Ntru+ sign: Compact ntru-based signatures using bimodal distributions," *Cryptology ePrint Archive*, 2025.

[12] J. H. Cheon, H. Choe, J. Devevey, T. Güneysu, D. Hong, M. Krausz, G. Land, M. Möller, D. Stehlé, and M. Yi, "Haetae: Shorter lattice-based fiat-shamir signatures," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 3, pp. 25–75, 2024.

[13] J. Lee, J. Cho, J. Ha, S. Kim, J. Kwon, B. Lee, J. Lee, S. Lee, D. Moon, M. Son, H. Yoon *et al.*, "The aimer signature scheme," 2024.

[14] A. Abdulrahman, V. Hwang, M. J. Kannwischer, and A. Sprenkels, "Faster kyber and dilithium on the cortex-m4," in *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*. Springer, 2022, pp. 853–871.

[15] J. Huang, J. Zhang, H. Zhao, Z. Liu, R. C. Cheung, Ç. K. Koç, and D. Chen, "Improved plantard arithmetic for lattice-based cryptography," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 4, pp. 614–636, 2022.

[16] G. Seiler, "Faster avx2 optimized ntt multiplication for ring-lwe lattice cryptography," Cryptology ePrint Archive, Paper 2018/039, 2018, https://eprint.iacr.org/2018/039. [Online]. Available: https://eprint.iacr.org/2018/039

[17] J. Huang, H. Zhao, J. Zhang, W. Dai, L. Zhou, R. C. Cheung, Ç. K. Koç, and D. Chen, "Yet another improvement of plantard arithmetic for faster kyber on low-end 32-bit iot devices," *IEEE Transactions on Information Forensics and Security*, 2024.

[18] Y. Kim, C. van Vredendaal, G. Banegas, and S. C. Seo, "Low-stack haetae for memory-constrained microcontrollers," *Cryptology ePrint Archive*, 2026.

[19] Y. Hyoung, S. Cho, U. Kim, M. Lee, H. Seo, and M. Sim, "Memory-efficient implementation of smaug-t and haetae," *Cryptology ePrint Archive*, 2026.

[20] O. Narlı, M. K. Pehlivanoğlu, and S. Akleylek, "A new optimized implementation of smaug-t for lightweight devices," in *Nordic Conference on Secure IT Systems*. Springer, 2025, pp. 77–90.

[21] ARM, *Cortex-M4 Devices Generic User Guide*, ARM, Aug. 2011, available at https://developer.arm.com/documentation/dui0553/latest/.

[22] Z. Liang and Y. Zhao, "Number theoretic transform and its applications in lattice-based cryptosystems: A survey," *arXiv preprint arXiv:2211.13546*, 2022.
[23] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.
[24] W. M. Gentleman and G. Sande, "Fast fourier transforms: for fun and profit," in *Proceedings of the November 7-10, 1966, fall joint computer conference*, 1966, pp. 563–578.
[25] J. Huang, A. Adomnicăi, J. Zhang, W. Dai, Y. Liu, R. C. Cheung, Ç. K. Koç, and D. Chen, "Revisiting keccak and dilithium implementations on armv7-m," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 2, pp. 1–24, 2024.
[26] 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: Module-lattice based fiat-shamir signature scheme (specification and reference implementation)," specification version 260204 and reference implementation v1.1.2; GitHub https://github.com/CryptoLabInc/HAETAE commit 743c31d. [Online]. Available: https://kpqc.cryptolab.co.kr/haetae
[27] A. Abdulrahman, J.-P. Chen, Y.-J. Chen, V. Hwang, M. J. Kannwischer, and B.-Y. Yang, "Multi-moduli NTTs for Saber on Cortex-M3 and Cortex-M4," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2022, no. 1, pp. 127–151, 2022.
[28] C.-M. M. Chung, V. Hwang, M. J. Kannwischer, G. Seiler, C.-J. Shih, and B.-Y. Yang, "Ntt multiplication for ntt-unfriendly rings: New speed records for saber and ntru on cortex-m4 and avx2," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 159–188, 2021.
[29] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "Improving software quality in cryptography standardization projects," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2022, pp. 19–30.
[30] E. Alkim, D. Y.-L. Cheng, C.-M. M. Chung, H. Evkan, L. W.-L. Huang, V. Hwang, C.-L. T. Li, R. Niederhagen, C.-J. Shih, J. Wälde *et al.*, "Polynomial multiplication in ntru prime: Comparison of optimization strategies on cortex-m4," *Cryptology ePrint Archive*, 2020.
[31] XKCP, "XKCP: extended Keccak Code Package," 2023, https://github.com/XKCP/XKCP.
[32] M. J. Kannwischer, R. Petri, J. Rijneveld, P. Schwabe, and K. Stoffelen, "PQM4: Post-quantum crypto library for the ARM Cortex-M4," https://github.com/mupq/pqm4, Accessed: 2025-07-30.
[33] J. Kwon, S. Lee, B. Lee, H. Seo, and J. Cho, "Efficient implementations of aimer post-quantum signature scheme for low-end to high-end iot devices," *IEEE Internet of Things Journal*, 2025.
[34] M. Lee, J. Jang, S. Kim, and S. Hong, "Improved frobenius fft for code-based cryptography on cortex-m4," *IEEE Internet of Things Journal*, 2025.
[35] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, 2016, pp. 53–70.
[36] S. Constable, J. Van Bulck, X. Cheng, Y. Xiao, C. Xing, I. Alexandrovich, T. Kim, F. Piessens, M. Vij, and M. Silberstein, "{AEX-Notify}: Thwarting precise {Single-Stepping} attacks through interrupt awareness for intel {SGX} enclaves," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4051–4068.
[37] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "Fact: a dsl for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 174–189.
[38] A. Langley, "ctgrind: Checking that functions are constant time with Valgrind," 2010, https://github.com/agl/ctgrind.
[39] M. Neikes, "TIMECOP: Automated dynamic constant-time analysis," 2020, https://www.post-apocalyptic-crypto.org/timecop/.
[40] O. Reparaz, J. Balasch, and I. Verbauwhede, "dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2017, pp. 1697–1702.
[41] L. Pocero Fraile, G. Tasopoulos, C. Koulamas, R. K. Zhao, N. H. Sultan, F. Regazzoni, and A. P. Fournaris, "Enabling quantum-resistant edhoc: Design and performance evaluation," *IEEE Access*, vol. 13, pp. 75 861–75 884, 2025.
[42] M. Schöffel, F. Lauer, C. C. Rheinländer, and N. Wehn, "Secure IoT in the era of quantum computers—where are the bottlenecks?" *Sensors*, vol. 22, no. 7, p. 2484, 2022.
[43] G. Tasopoulos, J. Li, A. P. Fournaris, R. K. Zhao, A. Sakzad, and R. Steinfeld, "Performance evaluation of post-quantum TLS 1.3 on resource-constrained embedded systems," in *Information Security Practice and Experience (ISPEC 2022)*, ser. Lecture Notes in Computer Science, vol. 13620. Springer, 2022, pp. 432–451.
[44] G. Tasopoulos, C. Dimopoulos, A. P. Fournaris, R. K. Zhao, A. Sakzad, and R. Steinfeld, "Energy consumption evaluation of post-quantum TLS 1.3 for resource-constrained embedded devices," in *Proceedings of the 20th ACM International Conference on Computing Frontiers (CF '23)*. ACM, 2023, pp. 366–374.
[45] G. Selander, J. P. Mattsson, and M. Tiloca, "Ephemeral Diffie–Hellman over COSE (EDHOC)," RFC 9528, IETF, 2024.
[46] T. Dierks and E. Rescorla, "The transport layer security (TLS) protocol version 1.2," RFC 5246, IETF, 2008.
[47] E. Rescorla, "The transport layer security (TLS) protocol version 1.3," RFC 8446, IETF, 2018.
[48] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile," RFC 5280, IETF, 2008.
[49] Internet Assigned Numbers Authority, "Ephemeral Diffie-Hellman Over COSE (EDHOC) Registries: IANA Protocol Parameter Registry," https://www.iana.org/assignments/edhoc/edhoc.xhtml, accessed: Jul. 12, 2026.
[50] ——, "CBOR Object Signing and Encryption (COSE): IANA Protocol Parameter Registry," https://www.iana.org/assignments/cose/cose.xhtml, accessed: Jul. 12, 2026.
[51] ——, "Transport Layer Security (TLS) Parameters: IANA Protocol Parameter Registry," https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml, accessed: Jul. 12, 2026.



JunHyeok Choi received the B.S. degree in information security, cryptology, and mathematics and the M.S. degree in cybersecurity from Kookmin University, Seoul, South Korea, in 2021 and 2026, respectively. He is currently pursuing the Ph.D. degree in cybersecurity with the Department of Cybersecurity, Kookmin University, Seoul, South Korea. His research interests include efficient implementations of post-quantum cryptography on embedded and parallel computing platforms, and cryptographic module validation.



DongHyun Shin received the B.S. degree in information security, cryptology, and mathematics and the M.S. degree in cybersecurity from Kookmin University, Seoul, South Korea, in 2023 and 2025, respectively. He is currently pursuing the Ph.D. degree in cybersecurity with the Department of Cybersecurity, Kookmin University, Seoul, South Korea. His research interests include optimization of cryptographic algorithms, post-quantum cryptography implementation, and SW/HW co-design for IoT devices.



Seog Chung Seo received the B.S. degree in information and computer engineering from Ajou University, Suwon, South Korea, in 2005, the M.S. degree in information and communications from the Gwangju Institute of Science and Technology (GIST), Gwangju, South Korea, in 2007, and the Ph.D. degree from Korea University, Seoul, South Korea, in 2011. He was a Research Staff Member with the Samsung Advanced Institute of Technology (SAIT) and the Samsung DMC Research and Development Center from September 2011 to April 2014. He was a Senior Research Member with the Affiliated Institute of ETRI, South Korea, from 2014 to 2018. He is currently a Professor with the Department of Cybersecurity, Kookmin University, Seoul, South Korea. His research interests include public-key cryptography, efficient implementations of cryptographic algorithms on various IT devices, cryptographic module validation, network security, and data authentication algorithms.