

Accelerating HQC for Post-Quantum TLS 1.3 on x86 IoT Gateways

Jihoon Jang, Hyunju Park, Jebin Kim, Seokhie Hong and Suhri Kim

Abstract—Post-quantum TLS at an IoT gateway must protect many device connections without large increases in handshake delay, CPU cost, or network traffic. HQC provides code-based diversity beyond ML-KEM, but its computation and ciphertext sizes can increase these costs. We optimize HQC for x86 processors with AVX2, AVX-512, and the Galois Field New Instructions (GFNI), and integrate the resulting implementations into TLS 1.3. We extend branch-free Toom-Cook/Karatsuba multiplication to AVX-512, accelerate Reed–Solomon decoding with GFNI, improve Reed–Muller decoding, accelerate SHA3-512 and fixed-weight sampling, and port Frobenius additive FFT (FAFFT) multiplication to AVX-512 + GFNI for HQC-5. On an Intel Core i5-1135G7 processor, our AVX2 implementation reduces decapsulation by 11.1–21.5% over the fastest prior AVX2 results. Our AVX-512 implementation reduces key generation by 24.9–29.5%, encapsulation by 10.4–11.1%, and decapsulation by 20.6–26.1% relative to Cabral et al. across the three HQC parameter sets. We evaluate the effects of these implementations on TLS 1.3 handshake latency and server and client CPU costs, as well as the effects of round-trip time (RTT) and bandwidth. In local loopback TLS measurements, our AVX-512 HQC-5 implementation reduces handshake latency from 3.22 ms to 2.92 ms compared with Cabral et al. On a constrained path with 1 Mbit/s bandwidth and an added RTT of 50 ms, the HQC-5 handshake takes 243 ms, compared with 87 ms for ML-KEM-1024. This indicates that HQC public-key and ciphertext sizes, rather than implementation speed, determine the remaining handshake cost on constrained links.

Index Terms—Post-Quantum Cryptography, HQC, SIMD Implementation, AVX-512, GFNI, TLS 1.3, Post-Quantum TLS

I. INTRODUCTION

LARGE-SCALE quantum computers would break the classical public-key cryptography used by current secure communication protocols through Shor’s algorithm [1]. This threat is particularly important in IoT environments, where resource-constrained devices connect to external services through gateways. Deploying post-quantum cryptography in such environments requires efficient implementations that minimize the additional computational cost and communication latency introduced by the transition.

TLS is a representative deployment target for post-quantum key establishment. In TLS 1.3, a KEM determines the key-share group. Its public key and ciphertext are carried in the

handshake flights. Thus, post-quantum TLS affects gateway computation and the latency of device connections. NIST standardized ML-KEM as FIPS 203 in 2024 [2] and selected HQC [3], a code-based KEM, for standardization in March 2025 to provide diversity beyond lattice-based ML-KEM. The main HQC components are polynomial multiplication over $\mathbb{F}_2[x]/(x^N - 1)$, Reed-Muller and Reed-Solomon decoding, and Keccak-based functions. Efficient implementations of these components are therefore important for practical post-quantum TLS at an IoT gateway.

AVX-512 has been applied to a wide range of cryptographic schemes [4]–[11]. It extends x86 SIMD processing to 512-bit registers and provides a richer vector instruction set. VPCLMULQDQ supports vectorized carry-less multiplication, while GFNI supports SIMD arithmetic over $\mathbb{F}_{2^8}$. Recent processors such as Tiger Lake support AVX-512, VPCLMULQDQ, and GFNI.

Cabral et al. [12] presented an AVX-512 implementation of HQC with optimizations for polynomial multiplication, Keccak-based functions, and both Reed-Muller and Reed-Solomon decoding. However, their Reed-Solomon decoder does not use GFNI. It relies on PCLMULQDQ-based field multiplication, with SIMD parallelism applied to the syndrome and error-value computations while the remaining arithmetic is largely scalar. In addition, the SHA3-512 oracles `hash_g` and `hash_i` remain scalar, the AVX-512 SHAKE-256 expansion is not applied to key generation, and single polynomial multiplication and support expansion are still based on AVX2 implementations.

HQC optimization and PQ-TLS evaluation have largely been studied separately. To our knowledge, existing PQ-TLS studies compare HQC with other KEMs using the implementation provided by the TLS stack [13], [14]. They do not vary the HQC implementation to determine how its optimization changes protocol cost. We optimize HQC for AVX2 and AVX-512 + GFNI, integrate the resulting implementations into one fixed TLS 1.3 stack, and measure their effects on handshake latency and server and client CPU costs. The network experiments then show when communication costs imposed by round-trip time (RTT) and bandwidth dominate these computation gains.

A. Contributions

We optimize polynomial multiplication, decoding, Keccak-based functions, and fixed-weight sampling using AVX-512 and GFNI. We also apply our decoding methods to AVX2. The main contributions of this paper are summarized as follows.

Jihoon Jang, Hyunju Park, and Jebin Kim are with the School of Cyber Security, Korea University, Seoul, Republic of Korea.

Suhri Kim is with the School of Mathematics, Statistics and Data Science, Sungshin Women’s University, Seoul, Republic of Korea (e-mail: suhrikim@sungshin.ac.kr).

Seokhie Hong is with SmartM2M, Republic of Korea.

Corresponding author: Suhri Kim (e-mail: suhrikim@sungshin.ac.kr).

This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.

- **Polynomial multiplication on AVX-512.** We extend the branch-free Toom-Cook/Karatsuba design of Jang et al. [15] to AVX-512 and select an efficient decomposition tree for each HQC parameter set. The resulting implementation has cycle counts within about 1% of Robert and Véron [16] at HQC-1 and HQC-3, and reduces the cost by 14.1% at HQC-5. We also port the FAFFT multiplication of Chen et al. [17] to AVX-512 + GFNI and reduce its transform cost with precomputed twiddle tables and reduced basis conversions. At HQC-5, the optimized FAFFT implementation reduces the cycle count by 17.7% over Robert and Véron.
- **Reed-Solomon and Reed-Muller decoding.** For Reed-Solomon decoding, we implement GFNI-based arithmetic for both AVX2 and AVX-512 and apply it to the critical arithmetic loops in all five $\mathbb{F}_{2^8}^{\text{RS}}$ phases. Our AVX2 implementation reduces the decoder cycle count by 83.2%–87.1% over Chen et al. [17], while our AVX-512 implementation reduces it by 88.9%–91.0% over Cabral et al. [12]. For Reed-Muller decoding, we replace the binary search for the maximum in `find_peaks` with a horizontal vector reduction in both AVX2 and AVX-512. This reduces the full decoder cost by 18.2%–20.4% over Chen et al. on AVX2 and by 3.7%–9.9% over Cabral et al. on AVX-512.
- **Keccak-based functions and fixed-weight sampling.** We extend AVX-512 processing to operations that remain scalar or AVX2-based in the existing AVX-512 implementation. We implement the SHA3-512 oracles `hash_g` and `hash_i` with AVX-512, reducing their cycle counts by 37.4%–46.4% and 46.2%–46.7%, respectively, over Cabral et al. [12]. We also port `vect_write_support_to_vector`, which expands the sampled support into a dense vector, from AVX2 to AVX-512, reducing the support-expansion cost by 34.5%–36.8%.
- **TLS 1.3 integration and assessment.** We integrate optimized HQC implementations into TLS 1.3 through `liboqs` and `oqs-provider`. Using identical protocol stacks, we measure their effects on handshake latency and server and client CPU costs, while distinguishing implementation-level improvements from the impact of network conditions such as RTT and bandwidth. The measurements identify the HQC-5 ciphertext as the cause of an additional round trip at the default TCP initial congestion window.

At the KEM level, our AVX2 implementation reduces decapsulation by 11.1–21.5% over the fastest prior AVX2 results of Jang et al. [15] and Chen et al. [17]. Relative to Cabral et al. [12], our AVX-512 implementation reduces key generation by 24.9–29.5%, encapsulation by 10.4–11.1%, and decapsulation by 20.6–26.1%. At HQC-5, the FAFFT configuration increases these reductions to 33.1%, 16.3%, and 27.4%, respectively.

The source code and implementation details of the proposed method are available at <https://github.com/jihoonjang98/hqc-x86-simd>.

B. Related Works

The HQC NIST submission [3] provides an optimized AVX2 implementation. It uses PCLMULQDQ for carry-less polynomial multiplication and for $\mathbb{F}_{2^8}^{\text{RS}}$ multiplication in the Reed-Solomon decoder. In the latter, `gf_mul_vect` evaluates 16 products in parallel for syndrome computation, whereas several other arithmetic functions call `gf_mul`, which computes one field multiplication using PCLMULQDQ. We use this implementation as the AVX2 baseline.

Robert and Véron [16] studied AVX-512 polynomial multiplication over $\text{GF}(2)[x]$ using VPCLMULQDQ. Their implementation selects Toom-Cook/Karatsuba combinations according to the target degree range and applies them to the HQC parameter sets. Their work focuses on polynomial multiplication.

Jang et al. [15] proposed GFmulOpt, a systematic search method for branch-free Toom-Cook/Karatsuba decomposition trees for $\text{GF}(2)[x]$ polynomial multiplication. They evaluate GFmulOpt on AVX2 with PCLMULQDQ and VPCLMULQDQ, and on Arm Neon with PMULL. For HQC-1 and HQC-3 on AVX2, it reduces the polynomial multiplication cost by 17–32% over the HQC baseline, and it shows that an additive-FFT approach may be preferable for HQC-5. Their work focuses on polynomial multiplication and does not address AVX-512.

Chen et al. [17] applied Frobenius additive FFT (FAFFT) to HQC polynomial multiplication on x86 AVX2 and GFNI platforms. They also study a CRT-based additive-FFT decomposition for the non-power-of-two polynomial lengths in HQC, which is used in their implementation for platforms without long binary-polynomial multiplication instructions. For x86, they reuse transforms of common operands and cache transformed key polynomials to reduce repeated transform costs. They also optimize Reed-Muller and Reed-Solomon encoding and decoding on AVX2. The Reed-Solomon part further uses GFNI-based $\mathbb{F}_{2^8}^{\text{RS}}$ arithmetic. The cached variant improves repeated encapsulation and decapsulation at the cost of increased local key storage.

Cabral et al. [12] proposed an AVX-512 implementation of HQC. Their implementation accelerates polynomial multiplication, Reed-Muller decoding, Reed-Solomon decoding, and Keccak-based functions. For polynomial multiplication, the single-product path retains the Toom-Cook/Karatsuba implementation of HQC at AVX2 width. For the two products sharing r_2 , they pack h and s into ZMM registers and process both products with AVX-512. However, the Reed-Solomon decoder does not use GFNI, and several $\mathbb{F}_{2^8}^{\text{RS}}$ arithmetic functions remain scalar. Moreover, the SHA3-512 oracles and support expansion are not covered by their AVX-512 implementation.

Chen et al. [18] subsequently proposed the Strided Frobenius Additive FFT. SFAFFT introduces a stride to reduce the number of butterfly stages and performs the leaf products over smaller quotient rings. Their implementations target AVX2, GFNI, Apple M1, and Arm processors, but do not include AVX-512. The authors state that the source code will be released. Since it is currently unavailable, we leave a comparison under the same experimental conditions for future work.

Several works integrate post-quantum KEMs into TLS and measure handshake performance. Paquin et al. [19] emulate network conditions and show that communication sizes determine handshake time on constrained links. Sosnowski et al. [13] measure 23 key-agreement variants, including HQC, on identical server hardware. They report additional round trips when server messages exceed the initial TCP congestion window. Tasopoulos et al. [20] evaluate post-quantum TLS on embedded platforms. Souvatzidaki and Limnietis [14] evaluate HQC, Kyber, and BIKE in TLS 1.3 with the OQS provider under packet loss. These studies compare KEM choices using the implementations available in their respective TLS stacks. Zheng et al. [8] optimize ML-KEM with AVX-512 and integrate it into TLS 1.3. Their ML-KEM operations are 1.64 times faster than AVX2, but their TLS handshake throughput improves by 3%–5%. In Section VII, we compare optimized HQC implementations in one fixed TLS stack to measure their effect on protocol performance.

C. Organization

The remainder of this paper is organized as follows. Section II reviews the HQC KEM, the AVX-512 + GFNI instruction set, and TLS 1.3 key exchange. Sections III–V describe the optimization targets in HQC and our corresponding methods. Section VI presents the experimental results. Section VII describes the TLS 1.3 integration and assesses protocol-level performance, and Section VIII concludes the paper.

II. PRELIMINARIES

This section introduces the notation used in the paper, reviews the HQC KEM, and summarizes the AVX-512 and GFNI instructions used throughout this work.

A. Notation

Throughout the paper, $\mathbb{F}_2$ denotes the binary field. The Reed-Solomon code in HQC is defined over $\mathbb{F}_{2^8}^{\text{RS}} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x^2 + 1)$, whereas the GFNI multiplication instruction operates over $\mathbb{F}_{2^8}^{\text{AES}} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1)$. Let $R = \mathbb{F}_2[x]/(x^N - 1)$ and let $R_w = \{\mathbf{v} \in R \mid |\mathbf{v}| = w\}$. For a vector $\mathbf{v}$ and an integer $n' \leq N$, $\text{Trun}(\mathbf{v}, n')$ keeps the low n' bits of $\mathbf{v}$.

Bold lowercase letters, such as $\mathbf{h}$ and $\mathbf{r}_2$, denote elements of R or code vectors. The parameters N , n_1 , n_2 , ω , ω_r , ω_e , and δ follow Table I. The oracles H and J are instantiated with SHA3-256, while G and I are instantiated with SHA3-512. We denote the extendable-output function by XOF.

B. HQC

HQC [3] is a code-based KEM based on the hardness of syndrome decoding for quasi-cyclic codes. It is obtained by applying an implicit-rejection transform to an IND-CPA-secure public key encryption scheme, denoted by HQC-PKE. The three parameter sets, HQC-1, HQC-3, and HQC-5, target NIST security levels 1, 3, and 5, respectively.

HQC uses a concatenated code $\mathcal{C}$ of length $n_1 n_2$. The outer code is a Reed-Solomon code over $\mathbb{F}_{2^8}^{\text{RS}}$, and the inner code is

TABLE I
HQC PARAMETER SETS [3].

Parameter set	Security	N	n_1	n_2	δ	ω	ω_r, ω_e
HQC-1	NIST 1	17,669	46	384	15	66	75
HQC-3	NIST 3	35,851	56	640	24	100	114
HQC-5	NIST 5	57,637	90	640	29	131	149

a repeated Reed-Muller code based on $\text{RM}(1, 7)$. The Reed-Solomon component has error-correction capacity δ , while the Reed-Muller component repeats each encoded symbol $n_2/128$ times. Algorithms 1 and 2 summarize the components of HQC.

Algorithm 1 Main components in HQC-PKE [3].

```

1: Global parameters ( $N, n_1, n_2, \omega, \omega_r, \omega_e, \mathcal{C}$ )
2: HQC-PKE.KeyGen( $\text{seed}_{\text{PKE}}$ )
3:   ( $\text{seed}_{\text{dk}}, \text{seed}_{\text{ek}}$ )  $\leftarrow I(\text{seed}_{\text{PKE}})$ 
4:   ( $\mathbf{y}, \mathbf{x}$ )  $\xleftarrow{\$} R_\omega \times R_\omega$  from  $\text{seed}_{\text{dk}}$ 
5:    $\mathbf{h} \xleftarrow{\$} R$  from  $\text{seed}_{\text{ek}}$ 
6:    $\mathbf{s} \leftarrow \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$ 
7:   return  $\text{ek}_{\text{PKE}} \leftarrow (\text{seed}_{\text{ek}}, \mathbf{s})$ ,  $\text{dk}_{\text{PKE}} \leftarrow \text{seed}_{\text{dk}}$ 
8: HQC-PKE.Encrypt( $\text{ek}_{\text{PKE}}, \mathbf{m}$ ,  $\theta$ )
9:    $\mathbf{h} \xleftarrow{\$} R$  from  $\text{seed}_{\text{ek}}$ 
10:  ( $\mathbf{r}_2, \mathbf{e}, \mathbf{r}_1$ )  $\xleftarrow{\$} R_{\omega_r} \times R_{\omega_e} \times R_{\omega_r}$  from  $\theta$ 
11:   $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$ 
12:   $\mathbf{v} \leftarrow \mathcal{C}.\text{Encode}(\mathbf{m}) + \text{Trun}(\mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}, n_1 n_2)$ 
13:  return  $\text{c}_{\text{PKE}} \leftarrow (\mathbf{u}, \mathbf{v})$ 
14: HQC-PKE.Decrypt( $\text{dk}_{\text{PKE}}, \text{c}_{\text{PKE}} = (\mathbf{u}, \mathbf{v})$ )
15:   $\mathbf{y} \xleftarrow{\$} R_\omega$  from  $\text{seed}_{\text{dk}}$ 
16:  return  $\mathbf{m} \leftarrow \mathcal{C}.\text{Decode}(\mathbf{v} - \text{Trun}(\mathbf{u} \cdot \mathbf{y}, n_1 n_2))$ 

```

Algorithm 2 Main components in HQC-KEM [3].

```

1: HQC-KEM.KeyGen()
2:   $\text{seed}_{\text{KEM}} \xleftarrow{\$} \{0, 1\}^{|\text{seed}|}$ 
3:  ( $\text{seed}_{\text{PKE}}, \sigma$ )  $\leftarrow \text{XOF}(\text{seed}_{\text{KEM}})$ 
4:  ( $\text{ek}_{\text{PKE}}, \text{dk}_{\text{PKE}}$ )  $\leftarrow \text{HQC-PKE.KeyGen}(\text{seed}_{\text{PKE}})$ 
5:   $\text{ek}_{\text{KEM}} \leftarrow \text{ek}_{\text{PKE}}$ 
6:   $\text{dk}_{\text{KEM}} \leftarrow (\text{ek}_{\text{KEM}}, \text{dk}_{\text{PKE}}, \sigma, \text{seed}_{\text{KEM}})$ 
7:  return  $\text{ek}_{\text{KEM}}, \text{dk}_{\text{KEM}}$ 
8: HQC-KEM.Encaps( $\text{ek}_{\text{KEM}}$ )
9:   $\mathbf{m} \xleftarrow{\$} \{0, 1\}^k$ ,  $\text{salt} \xleftarrow{\$} \{0, 1\}^{|\text{salt}|}$ 
10:  ( $K, \theta$ )  $\leftarrow G(H(\text{ek}_{\text{KEM}}) \parallel \mathbf{m} \parallel \text{salt})$ 
11:   $\text{c}_{\text{PKE}} \leftarrow \text{HQC-PKE.Encrypt}(\text{ek}_{\text{KEM}}, \mathbf{m}, \theta)$ 
12:   $\text{c}_{\text{KEM}} \leftarrow (\text{c}_{\text{PKE}}, \text{salt})$ 
13:  return  $K, \text{c}_{\text{KEM}}$ 
14: HQC-KEM.Decaps( $\text{dk}_{\text{KEM}}, \text{c}_{\text{KEM}}$ )
15:  Parse  $\text{dk}_{\text{KEM}} = (\text{ek}_{\text{KEM}}, \text{dk}_{\text{PKE}}, \sigma, \text{seed}_{\text{KEM}})$ 
16:  Parse  $\text{c}_{\text{KEM}} = (\text{c}_{\text{PKE}}, \text{salt})$ 
17:   $\mathbf{m}' \leftarrow \text{HQC-PKE.Decrypt}(\text{dk}_{\text{PKE}}, \text{c}_{\text{PKE}})$ 
18:  ( $K', \theta'$ )  $\leftarrow G(H(\text{ek}_{\text{KEM}}) \parallel \mathbf{m}' \parallel \text{salt})$ 
19:   $\text{c}'_{\text{PKE}} \leftarrow \text{HQC-PKE.Encrypt}(\text{ek}_{\text{KEM}}, \mathbf{m}', \theta')$ 
20:   $\bar{K} \leftarrow J(H(\text{ek}_{\text{KEM}}) \parallel \sigma \parallel \text{c}_{\text{KEM}})$ 
21:  if  $\mathbf{m}' = \perp$  or  $\text{c}'_{\text{PKE}} \neq \text{c}_{\text{PKE}}$  then  $K' \leftarrow \bar{K}$ 
22:  return  $K'$ 

```

Polynomial multiplication is used to compute $\mathbf{h} \cdot \mathbf{y}$, $\mathbf{h} \cdot \mathbf{r}_2$, $\mathbf{s} \cdot \mathbf{r}_2$, and $\mathbf{u} \cdot \mathbf{y}$. The decoder $\mathcal{C}.\text{Decode}$ invokes Reed-Muller and Reed-Solomon decoding. Figure 1 shows the contribution

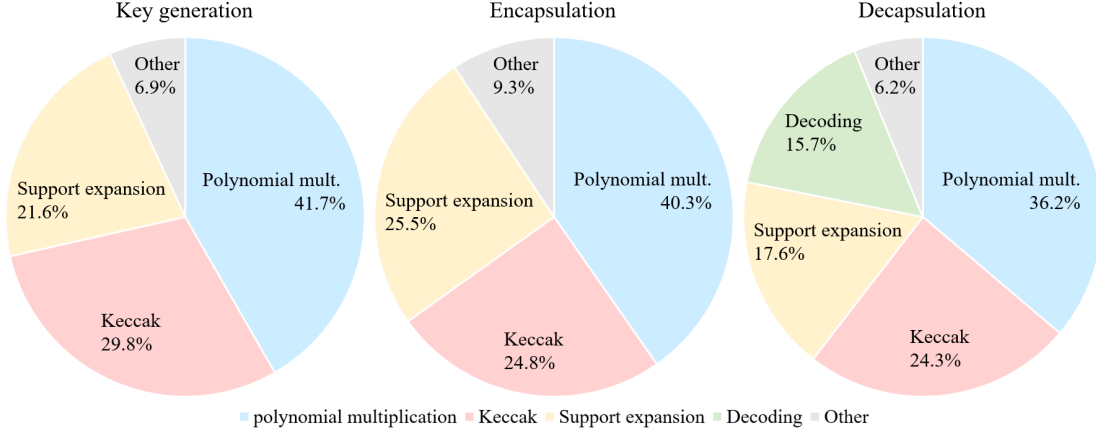


Fig. 1. Cycle distribution of the Cabral et al. AVX-512 implementation for HQC-5 on an Intel Core i5-1135G7 processor.

of these components to the KEM cycle counts in the AVX-512 implementation of Cabral et al. [12] at HQC-5. In the figure, the Keccak category includes the SHA3-256 oracles H and J , the SHA3-512 oracles G and I , and the SHAKE-256 XOF. Support expansion, the final step of fixed-weight sampling, converts the sampled support positions into a binary vector of length N . These results motivate our optimization of polynomial multiplication, Keccak-based functions, decoding, and support expansion.

C. AVX-512 and GFNI

AVX-512 extends the x86 SIMD instruction set to 32 512-bit ZMM registers and eight opmask registers. A ZMM register extends the corresponding 256-bit YMM register, which forms its lower half. In this work, we use AVX-512 instructions together with VPCLMULQDQ for binary polynomial multiplication and GFNI for byte-wise arithmetic over $\mathbb{F}_{2^8}^{\text{RS}}$.

The VPCLMULQDQ instruction computes carry-less products of 64-bit operands. The 256-bit form uses YMM registers and computes two 64-bit carry-less products per instruction. The 512-bit form uses ZMM registers and computes four such products.

GFNI provides byte-wise finite-field and affine instructions through `vgf2p8mulb`, `vgf2p8affineqzb`, and `vgf2p8affineinvqb` [21]. The multiplication and inversion instructions are defined over $\mathbb{F}_{2^8}^{\text{AES}}$, whereas the Reed-Solomon code in HQC is defined over $\mathbb{F}_{2^8}^{\text{RS}}$. We use the GFNI affine instructions to map elements from $\mathbb{F}_{2^8}^{\text{RS}}$ to $\mathbb{F}_{2^8}^{\text{AES}}$ before multiplication or inversion and map the results back to $\mathbb{F}_{2^8}^{\text{RS}}$ afterward. Let $T : \mathbb{F}_{2^8}^{\text{RS}} \rightarrow \mathbb{F}_{2^8}^{\text{AES}}$ be the linear isomorphism between the two fields. For $a, b \in \mathbb{F}_{2^8}^{\text{RS}}$, the RS-field multiplication is computed as

$$a \cdot_{\text{RS}} b = T^{-1}(T(a) \cdot_{\text{AES}} T(b)).$$

The same conversion is used for inversion as $a^{-1} = T^{-1}(T(a)^{-1})$.

Our AVX2 implementation targets processors supporting AVX2, PCLMULQDQ, and GFNI, and our AVX-512 implementation additionally requires AVX-512 and VPCLMULQDQ. The experiments are performed on an Intel

TABLE II
INSTRUCTION SUPPORT ON REPRESENTATIVE MICROARCHITECTURES.

Microarchitecture (year)	AVX2	AVX-512	VPCLMULQDQ	GFNI
Ice Lake (2019)	✓	✓	✓	✓
Tiger Lake (2020)	✓	✓	✓	✓
Rocket Lake (2021)	✓	✓	✓	✓
Alder Lake (2021)	✓	–	✓	✓
Raptor Lake (2022)	✓	–	✓	✓
Meteor Lake (2023)	✓	–	✓	✓
Arrow Lake (2024)	✓	–	✓	✓
Lunar Lake (2024)	✓	–	✓	✓
Skylake-X (2017)	✓	✓	–	–
Ice Lake Xeon (2021)	✓	✓	✓	✓
Sapphire Rapids (2023)	✓	✓	✓	✓
Emerald Rapids Xeon (2023)	✓	✓	✓	✓
Granite Rapids Xeon (2024)	✓	✓	✓	✓
Zen 3 (2020)	✓	–	–	–
Zen 4 (2022)	✓	✓	✓	✓
Zen 5 (2024)	✓	✓	✓	✓

Tiger Lake processor. Table II summarizes instruction support on representative x86 microarchitectures.

D. TLS 1.3 and Post-Quantum TLS

TLS 1.3 establishes a shared secret through a key-share exchange in the first round trip. When the key exchange is instantiated with a KEM, the client first generates a KEM key pair and sends the public key in its ClientHello message. The server then encapsulates a shared secret to this public key and returns the resulting ciphertext in its handshake flight. Finally, the client decapsulates the ciphertext with its secret key to recover the same shared secret. The client therefore performs key generation and decapsulation, while the server performs one encapsulation per handshake. The public key and the ciphertext are carried in the handshake flights, so the communication sizes of the KEM add directly to the handshake traffic. Post-quantum KEMs are commonly integrated into TLS through liboqs and oqs-provider [22], which expose them as key-share groups to OpenSSL. We use this path in Section VII.

III. POLYNOMIAL MULTIPLICATION

This section describes how polynomial multiplication is implemented in our work. The binary polynomial products in the HQC PKE component are computed in $R = \mathbb{F}_2[x]/(x^N - 1)$. As shown in Section II-B, HQC uses four such products, namely $\mathbf{h} \cdot \mathbf{y}$ in key generation, $\mathbf{h} \cdot \mathbf{r}_2$ and $\mathbf{s} \cdot \mathbf{r}_2$ in encryption, and $\mathbf{u} \cdot \mathbf{y}$ in decryption. Since N ranges from 17,669 to 57,637, these products account for a large part of the HQC cycle count.

A. Toom-Cook / Karatsuba for AVX-512

Toom-Cook and Karatsuba multiplication decompose a large binary-polynomial product into smaller subproducts and combine them by additions over $\mathbb{F}_2$. In HQC implementations, these decompositions are applied recursively until the subproducts reach a size handled by carry-less multiplication instructions.

For single polynomial multiplication, Robert and Véron [16] provide an AVX-512 base multiplication that computes four carry-less products of 64-bit operands per instruction. Their implementation places this base case below fixed Toom-Cook/Karatsuba trees. GFmulOpt of Jang et al. [15] systematically searches decomposition trees and generates branch-free implementations for AVX2 and Arm Neon. Cabral et al. [12] retain the Toom-Cook/Karatsuba decomposition of the HQC AVX2 implementation. Their lower Karatsuba layer uses both the 256-bit and 512-bit forms of VPCLMULQDQ, but the upper decomposition tree remains at AVX2 width.

We extend GFmulOpt to search and generate Toom-Cook/Karatsuba decomposition trees for AVX-512. We adopt the AVX-512 multiplication of Robert and Véron as the base case of the extended framework. With this base multiplication fixed, GFmulOpt selects the sequence of Toom-Cook and Karatsuba decompositions for each HQC parameter set. We use the resulting implementations for $\mathbf{y} \cdot \mathbf{h}$ in key generation and $\mathbf{y} \cdot \mathbf{u}$ in decryption.

In encryption, the two products $\mathbf{r}_2 \cdot \mathbf{h}$ and $\mathbf{r}_2 \cdot \mathbf{s}$ have no data dependency and can be processed in parallel. For this case, we use the `vect_mul2x` implementation of Cabral et al. [12] unchanged. It packs $\mathbf{h}$ and $\mathbf{s}$ into the two halves of a ZMM register and duplicates $\mathbf{r}_2$ across both halves, allowing the same Toom-Cook/Karatsuba operations to process both products in parallel.

B. Additive-FFT for AVX-512

FFT-based multiplication has lower asymptotic complexity than Toom-Cook/Karatsuba multiplication. At the polynomial sizes used in HQC, the transform overhead can offset this advantage. FFAFFT reduces the transform cost by using the Frobenius relation. The input polynomials are first converted from the monomial basis to the novel polynomial basis and evaluated over a Frobenius partition Σ using addFFT. Since the input polynomials have coefficients in $\mathbb{F}_2$, we have $f(\alpha^2) = f(\alpha)^2$. Hence, the value at α^2 can be derived from the value at α by repeated squaring. FFAFFT therefore reduces the number of explicit evaluation points by a factor of m compared with naive additive-FFT multiplication over $\mathbb{F}_{2^m}$.

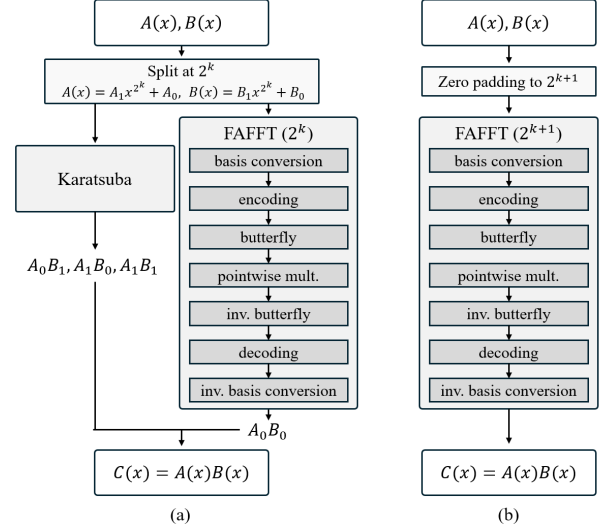


Fig. 2. FFAFFT-based polynomial multiplication of Chen et al. [17] for $N = 2^k + r$. (a) Combination of FFAFFT and Karatsuba multiplication for HQC-1 and HQC-3. (b) Straightforward FFAFFT multiplication for HQC-5. The values in parentheses denote the FFAFFT input sizes.

Chen et al. [17] implement FFAFFT with AVX2 and GFNI and show that it outperforms Toom-Cook/Karatsuba multiplication at HQC-5. Their implementation uses two multiplication structures according to the HQC parameter set. For HQC-1 and HQC-3, the polynomial length has the form $2^k + r$, where $r < 2^k$. FFAFFT computes the product of the first 2^k coefficients, while the products involving the remaining r coefficients are computed separately with Karatsuba multiplication. For HQC-5, the input polynomials are padded to 2^{k+1} coefficients and the entire product is computed with FFAFFT. Each FFAFFT computation follows the same sequence of basis conversion, encoding, forward butterfly, pointwise multiplication, inverse butterfly, decoding, and inverse basis conversion. Figure 2 illustrates the two multiplication structures and the FFAFFT stages.

We port this FFAFFT implementation to AVX-512 and use AVX-512 GFNI for encoding, decoding, and the forward and inverse butterfly operations, thereby increasing the degree of parallelism in the underlying field arithmetic. Specifically, the number of elements processed per iteration increases from 128 to 256 for encoding and decoding, and from 64 to 128 for the butterfly operations.

To further reduce the cost of the butterfly operations, we precompute the twiddle constants used in the fused low stages. Since these constants depend only on the public parameter set and the iteration index, we precompute the constants required for each iteration over 128 elements of $\mathbb{F}_{2^{32}}$ and store them in tables. This replaces the broadcasts and XORs used to construct the constants with table loads. The optimization introduces a trade-off between table storage and instruction count. The tables occupy 11,776, 23,552, and 47,104 bytes for HQC-1, HQC-3, and HQC-5, respectively. We also evaluated this approach for AVX2. However, the AVX2 butterfly processes only one 64-element block per iteration and does not require the two-block packing used in the AVX-512 implementation;

replacing constant construction with table loads therefore does not consistently reduce the cycle count. We consequently use the precomputed tables only in the AVX-512 implementation.

We further reduce the transform overhead by eliminating unnecessary basis-conversion operations on the padded zero region. For example, at HQC-5, the polynomial occupies only 7,205 bytes of the 8,192-byte FFT core, while the original basis conversion and its inverse process the entire core, including the padded zero region. We derive the corresponding loop ranges from the public parameter set and skip these unnecessary operations in both the AVX2 and AVX-512 implementations. Since the loop ranges depend only on the public parameter set, the resulting control flow and memory-access pattern remain input-independent.

For the Karatsuba component in Figure 2(a), we use the AVX-512 form of VPCLMULQDQ, which computes four 64-bit carry-less products in parallel.

C. Performance

Table III reports cycle counts of single polynomial multiplication on a Tiger Lake i5-1135G7. Turbo Boost and Hyper-Threading are disabled, and the benchmark process is pinned to a single core. Each value is the median of the measured samples after warmup, as described in Section VI-A. In the table, TC/Karat denotes Toom-Cook/Karatsuba multiplication.

TABLE III
CYCLE COUNTS OF POLYNOMIAL MULTIPLICATION ON AVX-512.

Implementation	Method	HQC-1	HQC-3	HQC-5
Robert-Véron [16]	TC/Karat	15,394	46,604	110,066
Cabral et al. [12]	TC/Karat	17,840	51,217	124,453
Ours	TC/Karat	15,584	46,178	94,539
Ours	FAFFT	27,650	67,037	90,627

The “TC/Karat” row of this work is obtained from the GF-mulOpt search extended to AVX-512, while the “FAFFT” row is the AVX-512 port of the implementation of Chen et al. [17] with the twiddle-table implementation and the reduced basis conversions of Section III-B. The TC/Karat implementation gives the lowest cycle count at HQC-1 and HQC-3, while the FAFFT implementation gives the lowest cycle count at HQC-5. The TC/Karat implementation is close to Robert-Véron at HQC-1 and HQC-3, with about 1% difference, and reduces the cost by 14.1% at HQC-5. Compared with Cabral et al. [12], the reductions are 12.6%, 9.8%, and 24.0%.

The relative cost of the AVX-512 FAFFT implementation over the Toom-Cook/Karatsuba implementation decreases with the parameter size, from 77% at HQC-1 to 45% at HQC-3, and turns into a 4.1% advantage at HQC-5. The crossover is realized by the twiddle-table implementation and the reduced basis conversions of Section III-B. These changes remove the per-iteration constant construction of the low butterfly stages and the basis-conversion operations on the padded zero part. For AVX-512, TC/Karat gives the lowest cycle count at HQC-1 and HQC-3, while FAFFT gives the lowest cycle count at HQC-5. We report both configurations at the KEM level in Section VI-C.

The table reports single polynomial multiplication only. For the two encryption products that share the operand r_2 , we reuse `vect_mul2x` of [12]. Under the same measurement conditions, `vect_mul2x` takes 27,364, 77,041, and 181,096 cycles on HQC-1, HQC-3, and HQC-5, respectively. These values are below twice the corresponding single polynomial multiplication cost, showing the benefit of shared-operand packing.

IV. RM/RS DECODING

This section presents our optimization of the Reed-Muller and Reed-Solomon decoding procedures in HQC. HQC uses a concatenated code that combines an outer Reed-Solomon code over $\mathbb{F}_{2^8}^{\text{RS}}$ and an inner Reed-Muller code $\text{RM}(1, 7)$ of length 128. The decoder operates in two stages. The Reed-Muller stage performs a maximum-likelihood decoding of each 128-bit block via the Fast Hadamard Transform, producing one byte of the outer codeword per block. The Reed-Solomon stage then runs an algebraic decoder over $\mathbb{F}_{2^8}^{\text{RS}}$ to correct residual errors and recover the message. The two stages have very different arithmetic profiles. The Reed-Muller stage mainly operates on 16-bit integer values, using additions and subtractions in the Hadamard transform and signed comparisons in `find_peaks`. The Reed-Solomon stage mainly uses $\mathbb{F}_{2^8}^{\text{RS}}$ multiplications and inversions. We therefore present them separately in Sections IV-A and IV-B.

A. Reed-Muller Decoder

1) *Decoder phases*: The Reed-Muller decoder recovers one byte from `MULTIPLICITY` duplicated 128-bit codeword copies. It consists of three phases, which we refer to as `expand_and_sum`, `hadamard`, and `find_peaks`, following the naming convention of the HQC AVX2 implementation. The `expand_and_sum` phase accumulates the duplicated codeword copies into 128 signed 16-bit values. The `hadamard` phase applies a seven-stage Fast Hadamard Transform in place. After correcting the zero coefficient by $64 \cdot \text{MULTIPLICITY}$, the `find_peaks` phase locates the entry of maximum absolute value in the transformed vector. The decoded byte is obtained from the corresponding 7-bit position and sign bit.

2) *Prior implementations*: The HQC NIST submission [3] implements the three Reed-Muller phases with AVX2. The `hadamard` phase uses `_mm256_hadd_epi16` and `_mm256_hsub_epi16`, which compute horizontal additions and subtractions on 16-bit lanes. To find the maximum, the `find_peaks` phase performs a binary search with a fixed number of iterations over the possible coefficient values. It then obtains the position of the maximum with `_bit_scan_forward`, which returns the index of the least significant set bit.

Chen et al. [17] optimize `expand_and_sum` with AVX2. They expand each 16-bit input word into 16 16-bit values using vector comparisons and accumulate the duplicated codeword copies with `vpaddw`. Their `hadamard` and `find_peaks` follow the HQC AVX2 implementation.

Cabral et al. [12] reimplement the three phases with AVX-512. For `expand_and_sum`, they replace the AVX2 bit-extraction pattern with `vpmovm2w` and accumulate the duplicated codeword copies with `vpaddw`. For `hadamard`, they unroll the seven butterfly stages and use `_mm512_mask_blend_epi16` with byte shifts and permutations. For `find_peaks`, they keep the binary-search structure of the reference and replace the AVX2 extraction step with a constant-time mask-based extraction of the maximum index.

3) *Our Reed-Muller decoder:* We modify the `find_peaks` phase for both AVX2 and AVX-512. We use the `expand_and_sum` and `hadamard` implementations of Cabral et al. [12] for AVX-512 and those of Chen et al. [17] for AVX2. In both settings, we replace the previous `find_peaks` with the maximum-search method described below.

Both `find_peaks` implementations retain the compact 256-bit vector layout and replace the binary search for the maximum with a horizontal vector reduction. Figure 3 shows this reduction in our AVX2 implementation. It first computes lane-wise maxima over eight vectors and then reduces the 16 remaining values to one maximum. The AVX-512 implementation performs the same sequence on 256-bit vectors. Once the maximum is obtained, comparison masks identify its row. The mask of the selected row is collapsed, and `_bit_scan_forward` returns the column index. The AVX2 implementation selects the row and its coefficient with constant-time masks. The AVX-512 implementation uses comparison masks for the row and `_mm256_permutexvar_epi16` for the coefficient. This avoids a data-dependent scalar index into a temporary array. The sign bit of the final output is then derived from the selected coefficient without data-dependent branches.

The resulting `find_peaks` has lower cycle counts than both the HQC AVX2 implementation and the AVX-512 implementation of Cabral et al., as shown in Section IV-C.

B. Reed-Solomon Decoder

1) *Decoder phases:* The Reed-Solomon decoder corrects up to δ byte errors in the outer codeword by computing syndromes, recovering the error-locator polynomial, locating the error positions, and applying Forney’s formula. It consists of six phases. The `compute_syndromes` phase evaluates the received word at the 2δ syndrome points in $\mathbb{F}_{2^8}^{\text{RS}}$. The `compute_elp` phase runs the Berlekamp-Massey algorithm to recover the error-locator polynomial σ . The `compute_roots` phase locates the error positions from the roots of σ . The `compute_z_poly` phase forms the auxiliary polynomial z used by Forney’s formula. The `compute_error_values` phase applies Forney’s formula to compute the nonzero coefficients of the error polynomial. The `correct_errors` phase XORs the error pattern into the received word. The first five phases mainly consist of $\mathbb{F}_{2^8}^{\text{RS}}$ arithmetic, while `correct_errors` is a byte-wise XOR step.

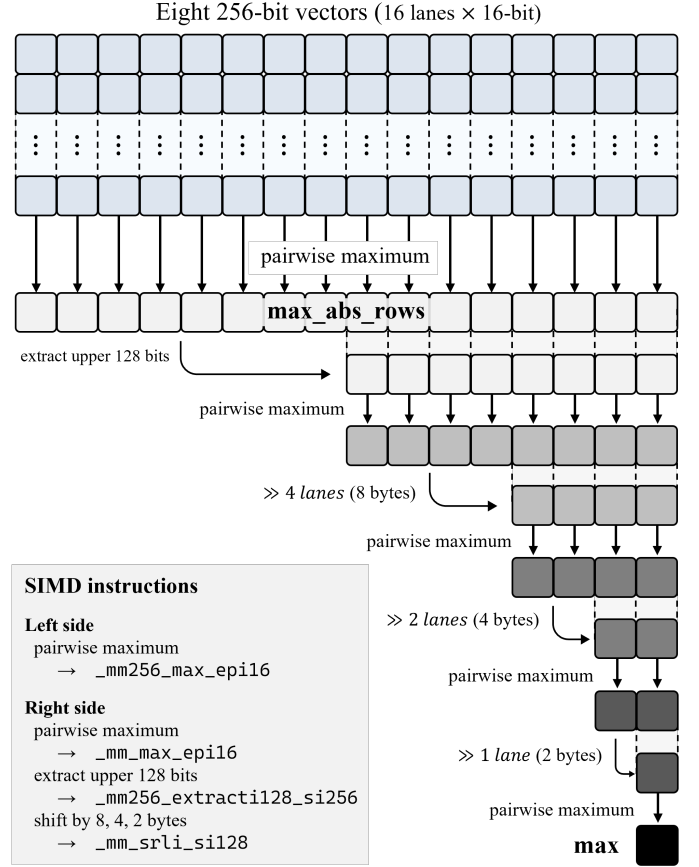


Fig. 3. Maximum reduction in the AVX2 implementation of `find_peaks`.

2) *Prior implementations:* The optimized AVX2 implementation in the HQC NIST submission [3] evaluates syndromes with the AVX2 function `gf_mul_vect`, which implements packed $\mathbb{F}_{2^8}^{\text{RS}}$ multiplication using PCLMULQDQ. Several other arithmetic paths invoke the scalar `gf_mul` function, which computes one carry-less product with PCLMULQDQ followed by polynomial reduction.

Chen et al. [17] also optimize $\mathbb{F}_{2^8}^{\text{RS}}$ multiplication on AVX2+GFNI. In the Reed-Solomon decoder, their syndrome computation converts bytes to the AES field representation with `vgf2p8affineqb`, uses `vgf2p8mulb` for packed products, and converts the result back to the Reed-Solomon representation. The remaining decoding phases still rely on scalar `gf_mul` calls.

Cabral et al. [12] mainly introduce AVX-512 in `compute_error_values` of the Reed-Solomon decoder. This phase uses Montgomery batch inversion and 512-bit packed field multiplication with the AVX-512 form of VPCLMULQDQ without GFNI. The remaining decoding phases keep the HQC AVX2 field-arithmetic structure.

3) *Our Reed-Solomon decoder:* GFNI provides `vgf2p8mulb` for packed byte multiplication and `vgf2p8affineinvqb` for packed byte inversion over $\mathbb{F}_{2^8}^{\text{AES}}$. Since the Reed-Solomon code in HQC is defined over $\mathbb{F}_{2^8}^{\text{RS}}$, we use `vgf2p8affineqb` to convert the byte representation before and after each GFNI field operation.

Using this conversion, we replace the PCLMULQDQ-based field arithmetic in the five $\mathbb{F}_{2^8}^{\text{RS}}$ -arithmetic phases with packed GFNI operations for both AVX2 and AVX-512. Since each vector lane stores one byte, a 256-bit GFNI instruction processes 32 field elements and a 512-bit GFNI instruction processes 64 field elements. Our AVX2 decoder uses 256-bit vectors in all five phases and evaluates 32 Chien search candidates per iteration. Our AVX-512 decoder uses 256-bit vectors for the syndrome, error-locator, auxiliary polynomial, and error-value computations. Its Chien search uses 512-bit vectors to evaluate 64 candidates per iteration. The phases are optimized as follows.

- *compute_syndromes*. This phase evaluates the received codeword at all 2δ syndrome points. We process the evaluations in parallel with packed Horner computation, where the $\mathbb{F}_{2^8}^{\text{RS}}$ multiplications use `vgf2p8mulb` and the additions are XORs.
- *compute_elp*. This phase runs the Berlekamp-Massey update for the error-locator polynomial. We retain the control flow of the HQC AVX2 implementation and replace the inner $\mathbb{F}_{2^8}^{\text{RS}}$ multiplications with `vgf2p8mulb`. The discrepancy inversion processes one field element at a time using `vgf2p8affineinvqb`.
- *compute_roots*. This phase locates the roots of the error-locator polynomial. Prior implementations [3], [12], [17] use an additive-FFT-based procedure for root finding. This approach reduces the number of field evaluations, but it also requires basis conversions and data movement. These operations are not well matched to GFNI, which is most effective when the same byte-field operation is applied to many lanes. Since the candidates lie in $\mathbb{F}_{2^8}^{\text{RS}}$ and the degree of σ is bounded by δ , we use Chien search and evaluate $\sigma(x)$ over the nonzero field elements in parallel. Each iteration evaluates 64 candidates by Horner evaluation with `vgf2p8mulb` and XOR, and the roots are found by comparing the results with zero. Figure 4 illustrates this packed evaluation.
- *compute_z_poly*. This phase forms the auxiliary polynomial z from σ and the truncated syndrome sequence. We compute the convolution in vector blocks, where each accumulation step is a `vgf2p8mulb` followed by an XOR.
- *compute_error_values*. This phase evaluates Forney’s formula at the error locations. We collect the error locations into byte vectors, compute the required inverses with `vgf2p8affineinvqb`, and evaluate the numerator and denominator in parallel with `vgf2p8mulb`.

The resulting Reed-Solomon decoder replaces the PCLMULQDQ-based field arithmetic in the critical loops with packed GFNI operations. Cycle counts for the individual phases and for the full `reed_solomon_decode` are shown in Section IV-C.

C. Performance

Table IV reports cycle counts for the Reed-Muller and Reed-Solomon decoders on a Tiger Lake i5-1135G7. Turbo Boost and Hyper-Threading are disabled, and the benchmark process

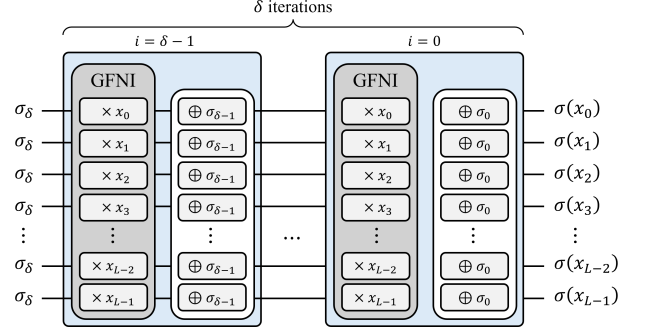


Fig. 4. Parallel Chien search based on Horner evaluation, where $L = 32$ for AVX2 and $L = 64$ for AVX-512.

is pinned to a single core. The measurements follow the same protocol as Table III. For both decoders, phase rows are measured with internal timing code placed around the named phases, and the “Total clean” row is measured around the complete decoder call in a clean build without this timing code. Hence, the phase rows indicate where the reductions occur, while “Total clean” is the decoder cost used for comparison. We group the results by AVX2 and AVX-512 implementations, and mark implementations using GFNI with *. The AVX2 setting compares the HQC AVX2 implementation [3], Chen et al. [17], and our AVX2+GFNI build. The AVX-512 setting compares the AVX-512 implementation of Cabral et al. [12] against our AVX-512+GFNI build.

For the Reed-Muller decoder, our implementation of Section IV-A3 reduces the clean per-call cost of `reed_muller_decode` by 67.9%–77.9% relative to the HQC AVX2 implementation, by 18.2%–20.4% relative to Chen et al. on AVX2, and by 3.7%–9.9% relative to Cabral et al. on AVX-512. Compared with the prior optimized implementations, the reduction mainly comes from `find_peaks`. The `find_peaks` phase is reduced by 25.7%–29.7% over Chen et al. on AVX2 and by 20.5%–25.7% over Cabral et al. on AVX-512. The AVX2 `hadamard` phase follows the same Fast Hadamard Transform structure as the HQC AVX2 implementation, based on `hadd` and `hsub`. Its profiled phase row is higher than that of the HQC AVX2 implementation because the phase counter places fence and timestamp boundaries around the transform. Therefore, this row is used only to locate where the costs are spent, while the clean full-decoder row is used for decoder-cost comparison. As a result, the improvement in `find_peaks` is reflected in the full Reed-Muller decoding cost.

For the Reed-Solomon decoder, the GFNI-based reimplementation of Section IV-B3 reduces the clean per-call cost of `reed_solomon_decode` by 84.7%–88.8% relative to the HQC AVX2 implementation, by 83.2%–87.1% relative to Chen et al. on AVX2, and by 88.9%–91.0% relative to Cabral et al. on AVX-512. The phase-level cycle counts of Chen et al. show a large reduction in `compute_syndromes`, where their implementation uses GFNI, but the other $\mathbb{F}_{2^8}^{\text{RS}}$ arithmetic phases remain close to the HQC AVX2 implementation. These results show that our gain is concentrated in the five phases that perform $\mathbb{F}_{2^8}^{\text{RS}}$ arithmetic. The final byte-wise XOR phase

`correct_errors` runs under 100 cycles in every configuration and is omitted from the table.

V. KECCAK-BASED FUNCTIONS AND FIXED-WEIGHT SAMPLING

All symmetric primitives used by HQC are built on the Keccak- $f[1600]$ permutation. In this section, we describe the AVX-512 implementation of the SHA3-512 oracles and the AVX-512 support expansion used in fixed-weight sampling.

A. Background

HQC instantiates four random oracles together with a SHAKE-based extendable-output function (XOF) used for vector expansion and rejection sampling. All five primitives use the Keccak- $f[1600]$ permutation and differ in the absorb pattern, the output length, and the domain separator. We list the call sites below.

- *Hash oracles H and J* are instantiated with SHA3-256. The oracle H takes the encapsulation key ek (between 2,241 and 7,245 bytes depending on the parameter set) as input. The oracle J takes $H(ek)$, σ , the ciphertext, and the salt as input. Both oracles produce a 32-byte output. The oracle H is called once in encapsulation and once in decapsulation, while J is called once in decapsulation to derive the implicit-rejection key.
- *Hash oracles G and I* are instantiated with SHA3-512. The oracle G derives the per-encapsulation seed (K, θ) from the message and salt, and is called once per encapsulation and once per decapsulation. The oracle I derives the key-pair seed from a single 32-byte input and is called once per key generation.
- *SHAKE-256 expansion* produces the public vector h from a 32-byte seed during both key generation and encapsulation, and additionally drives the rejection-sampling XOF used for the low-weight vector samplers in all three KEM operations.
- *Fixed-weight sampling* generates low-weight vectors in R_ω , R_{ω_r} , and R_{ω_e} from SHAKE-256 output. The sampler first obtains support positions by rejection sampling and then expands the support into a binary vector in R . This final support-expansion step is measured separately in Section V-D.

B. Prior Implementations

The HQC AVX2 implementation distributed with the NIST submission [3] uses the scalar Keccak implementation from PQClean [23] without SIMD instructions.

Cabral et al. [12] introduced a dedicated AVX-512 implementation of Keccak- $f[1600]$ that maps the 1600-bit state across five 512-bit registers, with five active 64-bit lanes per register. The 24 permutation rounds are unrolled and rely on `_mm512_ternarylogic_epi64` for the three-way XORs of the θ step, on `_mm512_rolv_epi64` for the per-lane rotations of the ρ step, and on `_mm512_permutexvar_epi64` for the cross-lane shuffles of the π step. On top of this permutation, Cabral et

al. inline absorb-and-squeeze wrappers for the two SHA3-256 oracles H and J and provide `shake256_512`, which is specialized to the expansion of the public vector h . These implementations are used only on the encapsulation and decapsulation paths. The two SHA3-512 oracles G and I and the incremental SHAKE-256 used by rejection sampling remain on the scalar PQClean wrappers in their implementation.

The HQC-3 implementation accompanying Cabral et al. [12] absorbed only 33 of the 67 input blocks of the oracle J . Since J is used only to derive the implicit-rejection key, this error is not detected by known-answer tests based on valid ciphertexts. A subsequent update corrected the absorption. The correction adds 34 Keccak- $f[1600]$ permutations to HQC-3 decapsulation. We use the corrected implementation for both the Cabral et al. baseline and our implementations.

The XKCP project [24] provides a third-party AVX-512 implementation of Keccak- $f[1600]$ with the same byte-level state layout as the PQClean scalar code, a 25-element array of `uint64_t`. We use this implementation in place of the scalar permutation called by the PQClean wrappers.

C. Our Keccak Strategy

1) *SHA3 oracles*: We use the AVX-512 implementations of H and J from Cabral et al. [12]. For the two SHA3-512 oracles G and I , which Cabral et al. leave on scalar wrappers, we provide AVX-512 implementations that follow the same template as their H and J . The state uses the same five-register layout. The absorb phase loads the input into the first nine lanes of the state with two masked loads, applies the SHA3-512 padding and domain separator at compile-determined byte offsets, and calls the Cabral et al. permutation implementation once or twice depending on the input length. The squeeze phase stores the 64-byte output from the rate region of the state with two masked stores. The number of permutation calls is fixed for each parameter set. The oracle I uses one permutation for all parameter sets, while G uses one at HQC-1 and two at HQC-3 and HQC-5.

2) *Fixed-weight sampling*: Fixed-weight sampling in HQC consists of support generation from SHAKE-256 followed by support expansion into a binary vector. For SHAKE-256, we replace the scalar Keccak- $f[1600]$ permutation with the XKCP AVX-512 implementation [24]. Rejection sampling ends with `vect_write_support_to_vector`, which expands the sampled ω support positions into a binary vector of length N . In a constant-time implementation, the sampled positions are not used as memory addresses. Therefore, support expansion scans all output words against all support positions and accumulates the matching bits with branch-free masks. Since the inner loop is repeated for each output word and support position, this step increases the cost of fixed-weight sampling. The HQC AVX2 implementation and Cabral et al. [12] both use the AVX2 implementation of this function, which compares four 64-bit output words at a time with a `vpcmpeqq-and-AND` pattern. We reduce this cost by processing eight 64-bit output words per iteration with AVX-512, using `_mm512_cmpeq_epi64_mask` and mask-controlled lane selection.

TABLE IV
CYCLE COUNTS OF REED-MULLER AND REED-SOLOMON DECODING. PHASE ROWS ARE MEASURED WITH INTERNAL TIMING CODE. TOTAL CLEAN ROWS ARE MEASURED AROUND COMPLETE DECODER CALLS WITHOUT THIS TIMING CODE.

Phase	AVX2				AVX-512			
	Impl.	HQC-1	HQC-3	HQC-5	Impl.	HQC-1	HQC-3	HQC-5
<i>Reed-Muller decoder</i>								
expand_and_sum	[3]	30,178	66,556	108,318	[12]	2,558	4,195	6,551
	[17]	2,004	2,699	4,293	Ours	2,561	3,964	6,458
	Ours	2,003	2,671	4,368				
hadamard	[3]	5,773	6,942	11,098	[12]	4,651	5,671	8,748
	[17]	7,711	10,611	17,093	Ours	4,469	5,538	9,374
	Ours	7,486	10,847	16,911				
find_peaks	[3]	13,114	16,763	26,950	[12]	9,934	12,802	20,821
	[17]	12,947	16,535	27,040	Ours	7,902	9,734	15,461
	Ours	9,623	11,824	19,005				
Total clean	[3]	45,067	86,073	137,703	[12]	11,128	15,326	24,569
	[17]	17,696	23,899	38,237	Ours	10,714	13,813	22,403
	Ours	14,475	19,087	30,431				
<i>Reed-Solomon decoder</i>								
syndromes	[3]	5,597	6,822	21,985	[12]	3,579	4,349	13,881
	[17]*	211	240	490	Ours*	128	138	315
	Ours*	183	208	315				
elp	[3]	17,937	19,709	47,069	[12]	17,738	19,622	46,712
	[17]	17,783	19,790	47,061	Ours*	3,089	3,298	5,954
	Ours*	7,505	8,024	14,535				
roots	[3]	9,401	17,666	18,195	[12]	9,662	17,025	18,016
	[17]	10,784	16,181	18,179	Ours*	392	420	877
	Ours*	677	730	1,604				
z_poly	[3]	1,174	1,323	4,219	[12]	1,189	1,323	4,246
	[17]	1,195	1,323	4,208	Ours*	391	411	695
	Ours*	417	403	697				
err_values	[3]	26,596	29,024	76,508	[12]	10,399	12,083	23,299
	[17]	26,635	28,949	76,484	Ours*	959	1,114	1,804
	Ours*	984	1,114	1,851				
Total clean	[3]	62,533	75,294	169,708	[12]	44,101	55,398	107,107
	[17]*	56,922	67,463	147,810	Ours*	4,904	5,246	9,654
	Ours*	9,580	10,321	19,075				

* Uses the GFNI instruction set.

D. Performance

Table V reports the cycle counts of the SHA3-512 oracles and support expansion optimized in this work. The two SHA3-512 oracles G and I are compared against Cabral et al. [12], whose implementation keeps these oracles on scalar wrappers. The support-expansion function `vect_write_support_to_vector` is also compared against Cabral et al., who use the AVX2 implementation for this function. Our AVX-512 implementations reduce the per-call cost by 37.4%–46.4% for G , by 46.2%–46.7% for I , and by 34.5%–36.8% for support expansion.

The two SHA3-256 oracles H and J and the `shake256_512` implementation are taken from Cabral et al. [12]. We also call `shake256_512` in key generation to generate the public vector $\mathbf{h}$. Since this change is reflected in the key-generation cycle count, we discuss it in the end-to-end evaluation.

TABLE V
CYCLE COUNTS OF SHA3-512 ORACLES AND SUPPORT EXPANSION. SUPPORT EXPANSION DENOTES `VECT_WRITE_SUPPORT_TO_VECTOR`.

Function	Implementation	HQC-1	HQC-3	HQC-5
G (hash_g)	[12]	1,646	3,206	2,769
	Ours	913	1,719	1,734
I (hash_i)	[12]	1,729	1,712	1,699
	Ours	921	913	914
Support expansion	[12]	5,778	17,175	35,607
	Ours	3,782	10,951	22,490

VI. EXPERIMENTAL EVALUATION

This section reports the cycle counts of our HQC implementations and compares them with the AVX2 implementation distributed with the HQC NIST submission [3] and the implementations of Jang et al. [15], Chen et al. [17], Robert and Véron [16], and Cabral et al. [12]. We verify our implementation against the HQC KAT files, which contain 100 test vectors

for each parameter set. For each of HQC-1, HQC-3, and HQC-5, our `crypto_kem_keypair`, `crypto_kem_enc`, and `crypto_kem_dec` functions reproduce the reference public key, secret key, ciphertext, and shared secret for all test vectors. Our implementation does not introduce secret-dependent branches or memory accesses. Its loop ranges and execution paths depend only on public parameters.

A. Setup

All measurements are taken on a single core of an Intel Core i5-1135G7 (Tiger Lake) clocked at 2.4 GHz, with Turbo Boost disabled through the operating system power management, Hyper-Threading disabled, and the benchmark process pinned to a single physical core. This setting avoids frequency changes during benchmarking and makes the cycle counts comparable across runs. Cycle counts are obtained through the `rdtscp` instruction with leading and trailing `lfence` barriers placed around each measurement window. Each reported value is the median of 1,000 measured samples after 10,000 warmup iterations. Each sample averages 100 iterations, and the warmup samples are excluded before aggregation. All binaries are built with `gcc 12` using `-O3`, `-funroll-all-loops`, and `-flto`. The main flags for the AVX2 builds are `-mavx2` and `-mpcmlmul`, while the AVX2+GFNI builds additionally use `-mgfni`. The AVX-512 builds use these AVX2 flags together with `-mavx512f`, `-mavx512bw`, `-mavx512dq`, `-mavx512vl`, and `-mvpclmulqdq`. Our AVX-512+GFNI builds additionally use `-mgfni`. End-to-end KEM, multiplication, and component measurements are obtained from builds without internal timing code. Reed-Muller and Reed-Solomon phase breakdowns are measured from separate builds with timing code around each phase and are used only in Table IV.

B. Component Performance

Cycle counts of the individual functions were presented alongside each component in Sections III–V. Table III compares AVX-512 single polynomial multiplication against Robert-Véron [16] and Cabral et al. [12]. Table IV compares the Reed-Muller and Reed-Solomon decoders on AVX2 and AVX-512. Table V compares the SHA3-512 oracles and support expansion. Using the clean total rows, our Reed-Solomon decoder reduces the cost by 84.7%–88.8% over the HQC AVX2 implementation and by 88.9%–91.0% over Cabral et al. on AVX-512. Our Reed-Muller decoder reduces the cost by 67.9%–77.9% over the HQC AVX2 implementation and by 3.7%–9.9% over Cabral et al. on AVX-512. The AVX-512 implementations of G , I , and `vect_write_support_to_vector` reduce the cost by 37.4%–46.4%, 46.2%–46.7%, and 34.5%–36.8%, respectively.

C. KEM Performance

Table VII reports the end-to-end cycle counts of HQC key generation, encapsulation, and decapsulation on AVX-512. We do not include SFAFFT [18] because its paper does not provide an AVX-512 implementation, and its source code was unavailable during our experiments. A comparison under

the same conditions remains future work. Ours (TC/Karat) applies the optimizations of Sections III–V with the Toom-Cook/Karatsuba multiplication of Section III-A. Ours (FAFFT) uses the same implementation except that polynomial multiplication is replaced with the FAFFT implementation of Section III-B. Ours (TC/Karat) reduces the cycle counts over Cabral et al. [12] by 24.9%–29.5% for key generation, 10.4%–11.1% for encapsulation, and 20.6%–26.1% for decapsulation.

At HQC-5, Ours (FAFFT) reduces key generation, encapsulation, and decapsulation by 5.0%, 6.7%, and 6.1%, respectively, over Ours (TC/Karat). This improvement follows from the 4.1% lower multiplication cost reported in Section III-C. At HQC-1 and HQC-3, Ours (TC/Karat) remains faster because the FAFFT transform cost is larger at these parameter sizes.

Table VI reports the AVX2 decapsulation cycle counts against the HQC AVX2 implementation [3], Jang et al. [15], and Chen et al. [17]. Chen et al. additionally present a cached variant that reuses the transform of r_2 for both products and stores the transforms of h and s across calls. For comparison with Ours, the row marked $\dagger$ retains the reuse of the transform of r_2 but does not store the transforms of h and s across calls. The unmarked “Chen et al.” row reports their main implementation. Ours applies the AVX2 Reed-Muller and Reed-Solomon decoding optimizations of Section IV. It uses the Jang et al. [15] multiplication at HQC-1 and HQC-3, and the AVX2 FAFFT implementation with the reduced conversion range at HQC-5. Ours reduces decapsulation by 11.1%–21.5% over the fastest prior AVX2 result for each parameter set. The reduction at HQC-1 and HQC-3 mainly comes from the GFNI-based Reed-Solomon decoder and the `find_peaks` implementation of Section IV-A3. At HQC-5, the AVX2 FAFFT implementation with the reduced conversion range provides an additional reduction.

Key generation benefits from the AVX-512 oracle I , the use of `shake256_512` to generate the public vector h , and polynomial multiplication. Encapsulation benefits from polynomial multiplication, the SHA3-512 oracle G , and fixed-weight sampling. Decapsulation additionally benefits from the optimized Reed-Muller and Reed-Solomon decoders.

TABLE VI
CYCLE COUNTS OF HQC DECAPSULATION ON AVX2.

Operation	Implementation	HQC-1	HQC-3	HQC-5
decapsulation	[3]	405,827	811,787	1,569,371
	[15]	396,427	818,611	1,482,364
	[17]	403,807	831,652	1,278,585
	[17] $\dagger$	396,546	813,573	1,242,655
	Ours	311,187	684,708	1,105,270

$\dagger$ Reuses the transform of r_2 for the two polynomial products.

VII. TLS 1.3 INTEGRATION AND EVALUATION ON X86 IOT GATEWAYS

Section VI measures each KEM operation in isolation. This section evaluates the same implementation configurations in TLS 1.3. We measure handshake latency, server and client CPU costs, and the effects of network RTT and bandwidth.

TABLE VII
CYCLE COUNTS OF HQC KEM OPERATIONS ON AVX-512.

Operation	Implementation	HQC-1	HQC-3	HQC-5
key generation	[16]	63,430	144,125	275,392
	[12]	65,670	148,789	289,996
	Ours (TC/Karat)	48,257	111,724	204,420
	Ours (FAFFT)	60,221	132,985	194,107
encapsulation	[16]	143,942	321,527	604,048
	[12]	101,345	229,154	441,300
	Ours (TC/Karat)	90,557	203,740	395,605
	Ours (FAFFT)	114,147	253,005	369,206
decapsulation	[16]	376,412	758,571	1,395,975
	[12]	214,674	434,076	837,664
	Ours (TC/Karat)	158,685	344,760	647,488
	Ours (FAFFT)	194,590	416,355	607,767

A. Setup

We integrate each implementation into liboqs 0.16.0 and expose it to OpenSSL 3.3.2 through oqs-provider 0.12.0-dev [22]. We keep each implementation’s original source tree unchanged and compile it as a separate position-independent static library with the flags of Section VI-A. We merge the objects into one relocatable object and link it into liboqs. Only the KEM API symbols `crypto_kem_keypair`, `crypto_kem_enc`, and `crypto_kem_dec` are exported. All remaining symbols are localized. Link-time optimization is disabled for every variant in this pipeline. This is the only deviation from the flags of Section VI-A. For every variant and parameter set, `kat_kem` reproduces the liboqs known-answer hashes. Thus, all stacks implement the same HQC v5.0.0 specification.

The rows labeled “Ours” use the compositions of Section VI-C. At levels 1 and 3, the AVX2 configuration combines the multiplication of Jang et al. [15] with our decoding optimizations. At level 5, it combines the FAFFT multiplication of Chen et al. [17] with the reduced conversion range and our decoding optimizations. The AVX-512 rows “Ours (TC/Karat)” and “Ours (FAFFT)” use the multiplication implementations of Section III.

We run the measurements on the Tiger Lake machine of Section VI-A under Ubuntu 24.04 on WSL2. Turbo Boost is disabled, and the clock is fixed at 2.4 GHz. The server and client are pinned to distinct physical cores. The server is `openssl s_server` with an ECDSA P-256 certificate and TLS 1.3 only. The client performs sequential full handshakes. Each iteration uses a fresh session without resumption and enables `TCP_NODELAY`. We measure TCP connection setup and the TLS handshake separately. Unless stated otherwise, each configuration uses 10 independent runs of 200 handshakes after warmup. We report the median of the 10 run medians.

End-to-end handshake time is measured at the client from the start of `SSL_connect`, after TCP connection setup, to its successful return. In addition, we report server and client CPU costs as on-CPU time. The server processes connections sequentially and waits for the client Finished message. Because of this wait, its wall-clock time per connection includes client computation and network delay. For server and client

CPU costs, we divide the CPU time accumulated by each endpoint process by the number of handshakes. We record the server time across a run and the client time around the handshake loop, which excludes provider initialization. The client performs key generation and decapsulation, whereas the server performs encapsulation. We compare HQC with the x86-64 assembly implementation of OpenSSL X25519 and with ML-KEM through the liboqs mlkem-native backend [22], which uses AVX2 on this platform.

B. Handshake Performance

Table VIII reports TLS 1.3 handshake performance measured over loopback. The “End-to-end” columns report handshake time, while the “Server CPU” and “Client CPU” columns report CPU costs. The liboqs reference implementation of HQC yields handshakes of 18.4 ms, 50.5 ms, and 121.0 ms for the three parameter sets. Our AVX-512 implementation reaches 1.57 ms, 2.05 ms, and 2.96 ms, respectively.

Compared with Cabral et al., our AVX-512 TC/Karat implementation reduces handshake time by 4.6% at HQC-1, 5.8% at HQC-3, and 8.0% at HQC-5. The HQC-1 difference varied across measurement sessions, whereas the reductions at HQC-3 and HQC-5 were robust. Our AVX-512 FAFFT implementation provides the lowest HQC-5 handshake time at 2.92 ms.

At HQC-1, our AVX-512 TC/Karat implementation completes the handshake in 1.57 ms, compared with 1.62 ms for the non-SIMD OpenSSL x86-64 assembly baseline. The AVX2 implementation of Faz-Hernández et al. [25] and the AVX-512-IFMA implementation of ENG25519 [26] take 1.51 ms and 1.49 ms, respectively. Thus, our AVX-512 TC/Karat implementation is 5.1% slower than ENG25519, the fastest classical implementation in Table VIII.

The AVX2 mlkem-native backend remains faster at matched security. The handshake times of our AVX-512 TC/Karat implementation are 12%, 40%, and 97% higher at HQC-1, HQC-3, and HQC-5, respectively. These differences reflect the different computational costs and protocol message sizes of HQC and ML-KEM. Section VII-C shows that HQC message sizes dominate handshake time on constrained links.

Reported AVX-512-optimized ML-KEM versions cannot be included because the source code of the versions reported by Zheng et al. [8] and He et al. [27] is unavailable. In addition, multi-way implementations such as the 32-way AVX-512 Kyber of Yu et al. [28] process many independent operations together. They are designed to improve total throughput, not the latency of a single handshake.

The “End-to-end” time does not show how the handshake cost divides between the two roles. The “Server CPU” and “Client CPU” columns of Table VIII show the per-role on-CPU costs. The client cost is larger than the server cost in every HQC configuration because the client performs key generation and decapsulation, whereas the server performs encapsulation. At HQC-1, our AVX-512 TC/Karat implementation has a server on-CPU cost of 736 μ s per handshake.

TABLE VIII
TLS 1.3 HANDSHAKE PERFORMANCE OVER LOOPBACK IN μ S. END-TO-END TIME IS THE MEDIAN OF 10 RUN MEDIANS.

ISA	Scheme	Implementation	End-to-end			Server CPU			Client CPU		
			Level 1 [§]	Level 3	Level 5	Level 1	Level 3	Level 5	Level 1	Level 3	Level 5
x86 [‡]	HQC	liboqs ref. [22]	18,356	50,489	121,014	7,896	21,729	51,240	10,560	29,060	69,367
	X25519	OpenSSL x86-64 asm	1,619	–	–	732	–	–	794	–	–
AVX2	HQC	NIST submission [3]	1,801	2,425	3,712	790	968	1,255	970	1,261	1,707
		Jang et al. [15]	1,787	2,418	3,564	793	959	1,211	968	1,245	1,650
		Chen et al. [17]	1,864	2,483	3,421	823	1,017	1,179	999	1,284	1,547
		Chen et al. [†] [17]	1,858	2,454	3,381	825	993	1,167	1,007	1,267	1,540
		Ours	1,737	2,332	3,307	787	958	1,145	924	1,174	1,456
	ML-KEM	mlkem-native [22]	1,399	1,462	1,502	654	663	676	709	728	744
	X25519	Faz-Hernández et al. [25]	1,514	–	–	695	–	–	738	–	–
AVX-512	HQC	Robert-Véron [16]	1,709	2,313	3,370	778	904	1,135	958	1,187	1,563
		Cabral et al. [12]	1,645	2,178	3,222	750	898	1,088	891	1,106	1,389
		Ours (TC/Karat)	1,570	2,052	2,963	736	875	1,061	853	1,029	1,282
		Ours (FAFFT)	1,652	2,165	2,923	791	904	1,016	918	1,078	1,236
	X25519	ENG25519 [26]	1,494	–	–	686	–	–	744	–	–

[§] Columns give the NIST security level, namely HQC-1, HQC-3, and HQC-5 for HQC and ML-KEM-512, ML-KEM-768, and ML-KEM-1024 for ML-KEM. X25519 provides 128-bit classical security and is listed at level 1 for reference.

[‡] Base x86 instruction set without SIMD extensions.

[†] Reuses the transform of r_2 for the two polynomial products.

C. Network Performance

IoT networks are often constrained by both latency and bandwidth. We therefore evaluate paths with added RTTs of 0 and 50 ms using `netem`, with no rate limit or rate limits of 10 Mbit/s and 1 Mbit/s. The emulated path uses a virtual Ethernet pair with an MTU of 1500 bytes. Table IX reports median handshake times over 100 handshakes.

Without a rate limit, the local computation differences remain visible at added RTT 0. With no rate limit and an added RTT of 50 ms, HQC-1 and HQC-3 take about 52 ms in every optimized configuration, whereas HQC-5 takes about 104 ms. The 14,421-byte HQC-5 ciphertext increases the server flight to about 15.5 KB, which exceeds the default TCP initial congestion window of 10 segments at this MTU. Thus, the large HQC-5 ciphertext can cause an additional round trip by exceeding the initial congestion window, as observed on the faster paths. Sosnowski et al. [13] report the same congestion-window effect for post-quantum handshakes with large messages.

At 10 Mbit/s and 1 Mbit/s, the time to transmit the handshake messages increasingly dominates. At 1 Mbit/s, the slower transmission lets acknowledgments arrive before the initial window is exhausted, eliminating the distinct RTT-scale penalty seen on faster paths. Under this rate limit, all HQC implementations take about 193 ms at added RTT 0 and 243 ms at added RTT 50 for HQC-5, while ML-KEM-1024 takes 37 ms and 87 ms, respectively. Thus, on constrained links, the communication sizes of HQC, rather than the choice between the AVX2 and AVX-512 implementations, determine the overall handshake time.

For Table IX, “Ours (AVX-512)” uses TC/Karat at all levels. “Ours (AVX2)” uses TC/Karat at HQC-1 and HQC-3, and FAFFT at HQC-5.

D. Storage and Memory Cost

The liboqs shared library built from the reference implementation occupies 0.77 MB. Our AVX-512 TC/Karat build occupies 1.14 MB, close to the 1.04 MB of Cabral et al., while every FAFFT-based build occupies 2.5 MB–2.75 MB due to transform tables. The performance results favor TC/Karat at HQC-1 and HQC-3, and FAFFT at HQC-5. For an HQC-5 deployment with limited storage space, TC/Karat remains a smaller alternative to FAFFT. Peak resident memory of the TLS endpoints is 9 to 10 MB in every configuration and is dominated by OpenSSL itself, so the choice of HQC implementation does not change process memory in practice.

VIII. CONCLUSION

This paper presented optimized HQC implementations for AVX2 and AVX-512 + GFNI, and evaluated them in TLS 1.3 on x86 IoT gateways. In these implementations, we extended GFmulOpt to AVX-512 for polynomial multiplication and used the AVX-512 base multiplication of Robert and Véron [16] inside the selected decomposition trees. We also ported FAFFT multiplication to AVX-512 and reduced its transform cost with precomputed twiddle tables and reduced basis conversions. For decoding, we optimized both Reed-Muller and Reed-Solomon decoding in the AVX2 and AVX-512 settings by combining vector data movement with GFNI-based arithmetic over $\mathbb{F}_{2^8}^{RS}$. For the Keccak-based functions, we implemented the SHA3-512 oracles with AVX-512 and ported support expansion in fixed-weight sampling to AVX-512.

At the KEM level, our AVX-512 TC/Karat implementation reduces key generation, encapsulation, and decapsulation by 24.9%–29.5%, 10.4%–11.1%, and 20.6%–26.1%, respectively, relative to Cabral et al. Our AVX2 implementation reduces decapsulation by 11.1%–21.5% relative to the fastest prior AVX2 result.

TABLE IX
MEDIAN TLS 1.3 HANDSHAKE TIME IN MS OVER EMULATED PATHS WITH ADDED RTT AND BANDWIDTH CONSTRAINTS. EACH ENTRY IS THE MEDIAN OF 100 HANDSHAKES.

RTT	Rate	Cabral et al. [12] (AVX-512)			Ours (AVX-512)			Ours (AVX2)			ML-KEM (AVX2)			
		HQC-1	HQC-3	HQC-5	HQC-1	HQC-3	HQC-5	HQC-1	HQC-3	HQC-5	-512	-768	-1024	X25519
0 ms	No rate limit	1.14	1.37	1.76	1.09	1.08	1.67	1.18	1.52	1.90	1.11	1.13	1.16	1.23
	10 Mbit/s	7.91	13.83	21.43	7.89	13.93	21.43	7.87	13.94	21.98	3.74	4.40	5.27	2.50
	1 Mbit/s	66.89	124.37	193.34	66.83	124.15	193.20	66.96	124.37	193.47	23.86	29.61	36.94	11.08
50 ms	No rate limit	52.39	52.92	104.37	52.23	52.05	104.16	52.50	52.65	104.66	52.29	52.35	52.40	52.54
	10 Mbit/s	58.58	64.37	113.42	58.42	64.43	113.32	58.43	64.70	113.64	54.60	55.22	56.08	53.33
	1 Mbit/s	116.89	174.07	243.41	116.77	174.23	243.04	116.90	174.23	243.52	74.08	79.82	87.44	61.63

For TLS 1.3, we measure end-to-end handshake latency and client and server on-CPU cost on loopback, then evaluate paths with added RTT and bandwidth constraints. On loopback, our AVX-512 TC/Karat implementation reduces the median handshake time from 2.18 ms to 2.05 ms at HQC-3 relative to Cabral et al., while our AVX-512 FFAFFT implementation reduces it from 3.22 ms to 2.92 ms at HQC-5. With a 50 ms added RTT and no rate limit, HQC-1 and HQC-3 take about 52 ms, whereas HQC-5 takes about 104 ms. At 1 Mbit/s, HQC-5 takes about 193 ms and 243 ms at added RTTs of 0 and 50 ms, compared with 37 ms and 87 ms for ML-KEM-1024. Our measurements confirm that, on constrained links, the communication sizes of HQC determine the overall handshake time more than the HQC implementation.

Our evaluation has two limitations. We use a single Tiger Lake processor with Turbo Boost and Hyper-Threading disabled, and the TLS tests target x86 IoT gateways over emulated network paths. Future work includes evaluating the implementations on additional server platforms, resource-constrained end devices, and deployed wireless IoT networks. For the HQC implementation, future work includes applying SFAFFT [18] to AVX-512 and integrating the caching techniques studied by Chen et al. [17] and Jang et al. [15] to evaluate the tradeoff between memory use and cycle counts.

REFERENCES

- [1] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.
- [2] National Institute of Standards and Technology, "Module-lattice-based key-encapsulation mechanism standard," U.S. Department of Commerce, Federal Information Processing Standards Publication (FIPS) NIST FIPS 203, Aug. 2024. [Online]. Available: <https://doi.org/10.6028/NIST.FIPS.203>
- [3] C. Aguilar-Melchor, N. Aragon, S. Betteaie, L. Bidoux, O. Blazy, J.-C. Deneuville, P. Gaborit, E. Persichetti, G. Zémor, J. Bos, A. Dion, J. Lacan, J.-M. Robert, and P. Veron, "Hamming Quasi-Cyclic (HQC)," National Institute of Standards and Technology, Tech. Rep., 2022, available at <https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions>.
- [4] D. M. Alter, "Optimizing the nist post quantum candidate sphincs+ using avx-512," Bachelor thesis, Radboud University, 2021.
- [5] H. Cheng, G. Fotiadis, J. Großschädl, P. Y. A. Ryan, and P. B. Rønne, "Batching CSIDH group actions using AVX-512," *IACR TCHEs*, vol. 2021, no. 4, pp. 618–649, 2021. [Online]. Available: <https://tches.iacr.org/index.php/TCHEs/article/view/9077>
- [6] H. Cheng, G. Fotiadis, J. Großschädl, and P. Y. A. Ryan, "Highly vectorized SIKE for AVX-512," *IACR TCHEs*, vol. 2022, no. 2, pp. 41–68, 2022.
- [7] N. Drucker, S. Gueron, and D. Kostic, "bike-kem: Additional optimized implementation of BIKE," <https://github.com/aws-labs/bike-kem>, accessed 2026-07-24.
- [8] J. Zheng, H. Zhu, Y. Dong, Z. Song, Z. Zhang, Y. Yang, and Y. Zhao, "Faster post-quantum tls 1.3 based on ml-kem: Implementation and assessment," *arXiv:2404.13544*, 2024.
- [9] R. Xu, D. He, M. Luo, C. Peng, and X. Zeng, "Optimizing dilithium implementation with avx2/-512," *ACM Transactions on Embedded Computing Systems*, vol. 23, no. 6, p. Article 98, 2024.
- [10] J. Zhang and J. Zhang, "Vectorized falcon-sign implementations using SSE2, AVX2, AVX-512F, NEON, and RVV," *Cryptology ePrint Archive*, Report 2025/1867, 2025. [Online]. Available: <https://eprint.iacr.org/2025/1867>
- [11] S.-W. Lee, S.-W. Eum, and H.-J. Seo, "Accelerating the aimer post-quantum signature with avx-512: A field-keccak speedup analysis," 2026.
- [12] R. Cabral, A. Faz-Hernandez, and J. López, "Accelerating hqc key encapsulation mechanism with avx-512," in *Proceedings of the ACM ASIA Public-Key Cryptography Workshop*, 2026, pp. 1–10.
- [13] M. Sosnowski, F. Wiedner, E. Hauser, L. Steger, D. Schoinianakis, S. Gallenmüller, and G. Carle, "The performance of post-quantum TLS 1.3," in *Companion of the 19th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT Companion '23)*. ACM, 2023.
- [14] K. Souvatzidaki and K. Limnietis, "Post-quantum key exchange in TLS 1.3: Further analysis on performance of new cryptographic standards," *Cryptography*, vol. 9, no. 4, p. 73, 2025.
- [15] J. Jang, M. Lee, S. Kim, S. C. Seo, and S. Hong, "Faster gf(2)[x] polynomial multiplication using simd instructions," *Journal of Cryptographic Engineering*, vol. 16, no. 1, p. 3, 2026.
- [16] J.-M. Robert and P. Véron, "Faster multiplication over $f_2[x]$ using avx512 instruction set and vpclmulqdq instruction," *Journal of Cryptographic Engineering*, vol. 13, no. 1, pp. 37–55, 2023.
- [17] M.-S. Chen, C.-M. Chiu, C.-T. Peng, and B.-Y. Yang, "Accelerating HQC with additive FFT," *Cryptology ePrint Archive*, Paper 2026/014, 2026. [Online]. Available: <https://eprint.iacr.org/2026/014>
- [18] M.-S. Chen, T.-Y. Chien, C.-M. Chiu, C. Huang, H.-H. Lin, C.-T. Peng, and B.-Y. Yang, "Strided frobenius additive FFT and its application to HQC," *Cryptology ePrint Archive*, Paper 2026/1588, 2026. [Online]. Available: <https://eprint.iacr.org/2026/1588>
- [19] C. Paquin, D. Stebila, and G. Tamvada, "Benchmarking post-quantum cryptography in TLS," in *Post-Quantum Cryptography (PQCrypto 2020)*, ser. Lecture Notes in Computer Science, vol. 12100. Springer, 2020, pp. 72–91.
- [20] 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. Springer, 2022.
- [21] Intel, "Intel Intrinsics Guide," <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>, 2024, accessed 2026-07-24.
- [22] D. Stebila and M. Mosca, "Post-quantum key exchange for the internet and the open quantum safe project," in *Selected Areas in Cryptography (SAC) 2016*, ser. Lecture Notes in Computer Science, vol. 10532. Springer, 2017, pp. 14–37, liboqs and oqs-provider, <https://openquantumsafe.org>.
- [23] M. J. Kannwischer, P. Schwabe, D. Stebila, and T. Wiggers, "Improving software quality in cryptography standardization projects," in *IEEE European Symposium on Security and Privacy, EuroS&P 2022 - Workshops, Genoa, Italy, June 6-10, 2022*. Los Alamitos, CA,

- USA: IEEE Computer Society, 2022, pp. 19–30. [Online]. Available: <https://eprint.iacr.org/2022/337>
- [24] Keccak Team, “extended keccak code package,” <https://github.com/XKCP/XKCP>, accessed: August 7, 2026.
 - [25] A. Faz-Hernández, J. López, and R. Dahab, “High-performance implementation of elliptic curve cryptography using vector instructions,” *ACM Transactions on Mathematical Software*, vol. 45, no. 3, 2019, artifact: <https://github.com/armfazh/fld-ecc-vec>.
 - [26] J. Zhang, J. Huang, L. Zhao, D. Chen, and Ç. K. Koç, “ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519,” in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024.
 - [27] Y. He, X. Hao, J. Li, and Y. Yu, “PQMagic: Towards secure and efficient post quantum cryptography implementations,” in *Applied Cryptography and Network Security Workshops (ACNS 2025 Satellite Workshops)*, ser. Lecture Notes in Computer Science, vol. 15654. Springer, 2025.
 - [28] X. Yu, J. Zhang, J. Huang, D. Chen, and L. Zhou, “Multi-way high-throughput implementation of Kyber,” in *Information Security (ISC 2024)*, ser. Lecture Notes in Computer Science. Springer, 2024.