LEAH: Lightweight and Efficient Hardware Accelerator for Code-based PQC Scheme HQC

Yazheng Tu and Jiafeng Xie
Department of Electrical and Computer Engineering. Villanova University, Villanova, USA
Email: {ytu1,jiafeng.xie}@villanova.edu

***Abstract*—The advent of quantum computing poses a significant threat to modern cryptography. To address this challenge, the National Institute of Standards and Technology (NIST) has initiated the Post-Quantum Cryptography (PQC) standardization process, with several algorithms being selected for standardization, including the recent code-based scheme HQC (Hamming Quasi-Cyclic). Meanwhile, a good number of research works in the field have switched to efficient hardware acceleration for PQC schemes. Following this trend, in this paper, we present a novel PQC hardware acceleration work, i.e., a Lightweight and Efficient hardware Accelerator for HQC (LEAH). Our design consists of three innovative hardware architectures for Key Generation, Encapsulation, and Decapsulation of HQC, respectively, while supporting all security levels. In total, we have proposed three layers of contributions, including: (i) dedicated design processes to obtain highly optimized major components for HQC, i.e., sparse polynomial multiplier, sampler, encoder, and decoder; (ii) novel data flow arrangement to design three operational phases of HQC that supports all parameter sets; (iii) a detailed comparison based on Field-Programmable Gate Array (FPGA) implementation to showcase the significant efficiency of the proposed design over the competing ones, e.g., Decapsulation architecture has at least 13.66% (at most 49.87%) less Equivalent Area-Delay Product (EADP) than the existing ones. We hope this outcome can facilitate the deployment of HQC in various applications and impact the ongoing NIST PQC standardization.**

***Index Terms*—FPGA, HQC, hardware accelerator, LEAH, PQC, decoder, sampler, sparse polynomial multiplier.**

I. Introduction

The advent of large-scale quantum computers poses a significant threat to the security of currently deployed cryptographic primitives, such as RSA and Elliptic Curve Cryptography [1], [2], and has created an urgent need for Post-Quantum Cryptography (PQC) [2]. The National Institute of Standards and Technology (NIST) has initiated the PQC standardization process in 2016 [3], [4]. After extensive evaluation, algorithms like KYBER were selected [3], [4]. However, NIST continues to evaluate alternative candidates, in order to protect against the possible weaknesses introduced by leveraging a single mathematical foundation [4]. Among these candidates, Hamming Quasi-Cyclic (HQC), a code-based KEM (Key Encapsulation Mechanism), has recently been selected for standardization as a backup scheme to KYBER [4].

HQC is a code-based IND-CCA (Indistinguishable under Chosen-Plaintext Attack) secure KEM [5]. Its security is based on the computational hardness of the syndrome decoding problem for quasi-cyclic (QC) codes [5]. Due to the increasing attention on HQC from the research community, several hardware implementation works of HQC have been reported.

Prior Works. A lightweight dense-sparse polynomial multiplication hardware accelerator was reported in [6], and a high-throughput design was proposed in [7]. A constant-time fixed weight sampler was proposed in [8]. Besides the components, several designs related to three operational phases of HQC, namely Key Generation (KeyGen), Encapsulation (Encap), and Decapsulation (Decap), were also reported. [9] proposed an architecture of HQC-1 based on Additive FFT (Fast Fourier Transform) in decoding. A design deploying the enhanced Parallel Inversionless Berlekamp-Massey (ePIBM) algorithm was proposed in [10]. A high-level synthesis-based architecture for HQC-1 was reported in [11]. Generic designs for three operational phases of HQC were reported in [12].

Challenges and Motivation. While several HQC hardware implementations have been reported, a more efficient and generic solution is still lacking. First, most existing works are incomplete, as they only address small components or a single security level (e.g., HQC-1 [11]). Besides, although implemented on the Artix-7 device, these designs require significant resources like DSPs and BRAMs, making them unsuitable for resource-constrained applications. Thus, there is an urgent need for new hardware acceleration that utilizes lightweight resources and achieves efficient performance, while supporting all HQC security levels (HQC-1, HQC-3, and HQC-5).

Contribution. Based on these considerations, this paper presents a Lightweight and Efficient hardware Accelerator for HQC (LEAH), addressing the limitations mentioned above. The proposed design comprises three innovative architectures for Key Generation (KeyGen), Encapsulation (Encap), and Decapsulation (Decap), respectively, and is fully compatible with the latest HQC specification [5]. We have proposed novel design strategies that result in substantial reductions in area-time complexity for all core HQC operations. Overall, we have proposed three layers of contributions as:

- We have optimized key components, including the fixed-weight sampler, polynomial multiplier, encoder, and decoder, which strictly adhere to the HQC specification and exhibit better performance in area-time complexities.
- We have proposed novel data flow to design generic hardware architectures (LEAH) for KeyGen, Encap, and Decap, respectively, while supporting all parameter sets.

This paper was accepted in the 2026 IEEE International Symposium on Hardware Oriented Security and Trust (HOST).

- We have conducted a comprehensive performance evaluation for LEAH on the AMD Artix-7 FPGA, demonstrating superior implementation results compared to the recently released HQC hardware design works.

The remainder of this paper is organized as follows: Section II provides a detailed background for the HQC algorithm. Section III provides a detailed process for obtaining optimized components. Section IV presents our proposed hardware accelerator (LEAH), detailing respective architectures. Section V presents the results and comparison with the existing works. Finally, Section VI concludes the paper.

II. Preliminaries

This section provides an introduction to both HQC.PKE and HQC.KEM, including the mathematical foundation as well as the algorithms for KeyGen, Encap, and Decap, respectively.

HQC. HQC is an IND-CCA secure KEM built on the hardness of a decision version of the Syndrome Decoding on structured codes [5]. It utilizes two types of linear codes: a random quasi-cyclic code for security and a fixed public code for error correction. The quasi-cyclic code is a $[2p, p, d]$ code with a public parity-check matrix $H = [I_p \mid \text{rot}(h)]$. The public fixed code is a concatenated code composed of an external shortened Reed-Solomon code and an internal duplicated Reed-Muller code. The dual-code structure allows for a precise upper bound on the decryption failure probability. Interested readers may refer to the document of [5] for details.

Algorithm 1: Fixed Hamming Weight Vector Sampling Algorithm [5]

Data: N, w, $seed$
Result: w distinct elements in range 0 to $N-1$

begin
  $raw \leftarrow$ prng(input = $seed$, output size = $32 \cdot w$);
  for $i \leftarrow w-1$ **to** 0 **do**
    $idx[i] = i + raw[32 \cdot (i+1) - 1 : 32 \cdot i] \pmod{(N-i)}$;
  end
  for $j \leftarrow w-1$ **to** 0 **do**
    $duplicate \leftarrow 0$;
    for $k \leftarrow j+1$ **to** $w-1$ **do**
      if $idx[j] == idx[k]$ **then**
        $duplicate \leftarrow 1$;
      end
    end
    if $duplicate == 1$ **then**
      $idx[j] = j$;
    end
  end
  return idx;
end

Random Number Generation. Following the HQC specification [5], we employ SHAKE256 (from the SHA-3 family) as our cryptographically secure pseudorandom number generator (CSPRNG) for all polynomial sampling operations by: 1) For dense polynomials, we generate all coefficients directly using the CSPRNG output. 2) For sparse polynomials, we generate only the nonzero indices pseudorandomly. The Fixed Hamming weight vector sampling algorithm recommended by the authors of HQC is shown in Algorithm 1.

Hash Functions. The HQC.KEM algorithm employs three specialized hash functions derived from SHAKE256-512, $\mathcal{G}(\cdot)$, $\mathcal{H}(\cdot)$, and $\mathcal{K}(\cdot)$, defined as SHAKE256-512($\cdot \parallel$ `G_FCT_DOMAIN`), SHAKE256-512($\cdot \parallel$ `H_FCT_DOMAIN`), and SHAKE256-512($\cdot \parallel$ `K_FCT_DOMAIN`), respectively. Each of these functions processes the input $(\cdot)$ concatenated with a unique one-byte domain constant, and produces a 512-bit output using SHAKE256-512.

HQC.PKE. The HQC Public Key Encryption (HQC.PKE) scheme forms the basis for the HQC.KEM, consisting of three main primitives: Key Generation, Encryption, and Decryption.

Algorithm 2: HQC.PKE.Encrypt [5]

Encrypt(pk, m):
sample e $\leftarrow \mathcal{R}$, such that $\omega(e) = \omega_e$
sample $r = (r_1, r_2) \leftarrow \mathcal{R}^2$ such that and $\omega(r_1) = \omega(r_2) = \omega_r$
$u = r_1 + h \cdot r_2$
$v = mG + s \cdot r_2 + e$
return $c = (u, v)$

The HQC.PKE encryption procedure (HQC.PKE.Encrypt) (Algorithm 2) takes the public key, a message, and a public 128-bit $salt$ value as input and produces the ciphertext (u, v). It begins by forming a seed θ from the message $m \in \{0,1\}^k$, the first 32 bytes of the public key pk, and a salt. The polynomial h is then regenerated from $\mathbf{seed1}$, which consists of the first 40 bytes of pk. Next, three fixed-weight sparse polynomials, $e, r_a, r_b \in R$, are sampled using θ with specified Hamming weights, $\omega(e) = w_e$ and $\omega(r_a) = \omega(r_b) = w_r$. Then, the message m is encoded with the generator matrix G, and the resulting codeword is added to the first $n_1 n_2$ bits of $s \cdot r_b + e$ to form the polynomial v. Finally, the ciphertext $c = (u, v)$ is returned.

Algorithm 3: HQC.PKE.Decrypt [5]

Decrypt(sk, c):
return C.**Decode**$(v - u \cdot v)$

The HQC decryption process (HQC.PKE.Decrypt) is used to recover the original message m' using the private key sk. The algorithm begins by re-sampling the secret polynomials x and y based on the secret key $\mathbf{seed2}$. It then computes the polynomial $v - u \cdot y$ and truncates the result. This resulting polynomial is then subjected to a decoding procedure that involves both Reed-Muller and Reed-Solomon decoding to correct errors and recover the original message m'. The final decrypted message m' is then returned.

HQC.KEM. The HQC KEM scheme (HQC.KEM) is constructed by wrapping the HQC.PKE scheme with the Hofheinz-Hövelmanns-Kiltz (HHK) transformation [12], [13].

It derives a shared session key and includes mechanisms to detect and handle decryption errors securely.

The KeyGen for HQC.KEM (HQC.KEM.KeyGen) is responsible for generating the public key pk and the private key sk. This process includes an additional secret seed, σ, for the Decap's implicit rejection mechanism. The procedure begins by generating this secret seed σ of k bits using SHAKE256. It then proceeds with the HQC.PKE.KeyGen steps, where two random 320-bit seeds, $\mathbf{seed1}$ and $\mathbf{seed2}$, are sampled. The seed $\mathbf{seed1}$ is expanded via SHAKE256 to sample a public dense polynomial $h \in R$, which forms the parity-check matrix $H = [I_p \mid \text{rot}(h)]$. Then, $\mathbf{seed2}$ is expanded to sample two secret fixed-weight sparse polynomials, $x, y \in R$, both with a Hamming weight w. The syndrome polynomial is then computed as $s \leftarrow x + h \cdot y$. Finally, the HQC.KEM.KeyGen procedure returns the public key $pk = (\mathbf{seed1}, s)$ and the private key $sk = (\mathbf{seed2}, \sigma)$.

Algorithm 4: HQC.KEM.KeyGen [5]

global parameters **param** = $(n, k, \delta, \omega, \omega_r, \omega_e)$.
KeyGen(param):
sample $h \leftarrow \mathcal{R}$
sample $(x, y) \leftarrow \mathcal{R}^2$ such that $\omega = \omega(x) = \omega(y)$
$sk = (x, y, \sigma)$
$pk = (h, s = x + h \cdot y)$
return (pk, sk).

The HQC.KEM Encap (HQC.KEM.Encap) procedure (Algorithm 5) takes the public key pk as input and produces a ciphertext c and a shared secret key K. The procedure begins by generating a digest θ from the concatenation of the message m, the head bytes of the secret key (specifically $\mathbf{seed1}$), and the salt using a hash function $\mathcal{G}$. This digest, along with the public key pk and message m, is used as input to the HQC.PKE.Encrypt routine, which produces the ciphertext components (u, v). The shared secret key K is then derived using a key derivation function $\mathcal{K}$ on the concatenation of m and the ciphertext components. The final output is the ciphertext $c = (u, v, \text{salt})$ and the shared secret key K.

Algorithm 5: HQC.KEM.Encap [5]

Encapsulate(pk, m):
$\theta = \leftarrow \mathcal{G}(m, \mathbf{seed1}, salt)$
$c \leftarrow (u, v) = \mathbf{Encrypt}(pk, m, \theta)$
$K \leftarrow \mathcal{K}(m, c)$
Return $c = (u, v, salt, K)$

Unlike HQC.PKE.Decrypt, the algorithm of HQC.KEM Decap (HQC.KEM.Decap) (Algorithm 6) aims to recover the shared secret key K instead of the message m', with possibly triggered implicit rejection mechanism. It includes a re-encryption step for security against active attackers. First, it calls HQC.PKE.Decrypt with $\mathbf{seed2}$ and the ciphertext components (u, v) to compute the decrypted message m'. A new random value θ' is then re-formed by hashing the concatenation of m', $\mathbf{seed1}$, and the salt using $\mathcal{G}$. A new ciphertext $c' = (u', v')$ is generated by re-encrypting m' using HQC.PKE.Encrypt with $(\mathbf{seed1}, s)$ and θ'. The received ciphertext c is then compared with the re-computed ciphertext c'. If they match ($c = c'$), the shared secret key K' is derived using $\mathcal{K}$ on the concatenation of m' and the ciphertext c. If they do not match ($c \neq c'$), a new shared secret key K is derived using $\mathcal{K}$ with the concatenation of a secret value σ and the ciphertext components to prevent any information leakage. Final output is the shared secret key K.

Algorithm 6: HQC.KEM.Decap [5]

Decapsulate(pk, c, σ):
decrypt m' = $\mathbf{Decrypt}(sk, c)$
compute $\theta' = \mathcal{G}(m')$
(re-)encrypt m' to get $c' \leftarrow \mathbf{Encrypt}(pk, m', \theta')$
if $\boldsymbol{c \neq c'}$ *or* $m = \perp$ **then**
  $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$
else
  $K \leftarrow \mathcal{K}(\sigma, \mathbf{c})$
end

TABLE I: HQC Parameter Sets

Level	N	ω	$\omega_e = \omega_r$	Reed-Solomon[1]	Reed-Muller[2]
HQC-1	17,669	66	75	[46,16,31]	[384,8,192]
HQC-3	35,851	100	114	[56,24,33]	[640,8,320]
HQC-5	57,637	131	149	[90,32,59]	[640,8,320]

[1]: $[n, k, \delta]$. [2]: $[n, k, \delta]$

Security Levels. There exist three security levels of HQC, i.e., HQC-1, HQC-3, HQC-5 [5]. Each security level corresponds to a different parameter set. Interested readers may refer to [5] for more details, as shown in Table I. Note that our design is applicable to all three security levels.

III. Optimized Architectural Details of Major Low-Level Components

This section presents a detailed description of the proposed hardware design processes for key components of HQC. Following the stated algorithmic details of HQC in Section II, we briefly list the involved key components as: (i) Keccak core; (ii) sampler, (iii) polynomial arithmetic; (iv) encoder; and (v) decoder. Accordingly, we provide a detailed design process for these components, from algorithm to architecture.

A. Keccak

The Keccak core, more specifically, SHAKE256, is responsible for generating the raw data of non-zero indices for the fixed-weight polynomials, dense polynomials, and the shared secret. In this work, we adopt the Keccak core used in [14] and modified the wrapper so that the input/output length matches the memory width of the proposed accelerator (LEAH). Interested readers may refer to [14] for more details.

B. Sampler

A few efficient sparse polynomial samplers have been reported. However, existing works still involve extra resource

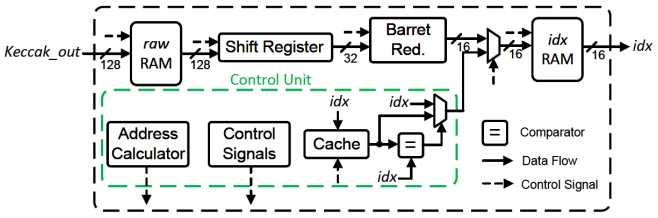


Fig. 1: The proposed sampler (*idx*: index).

usages such as DSP [9] and RAM tiles [12]. To further reduce the resource usage, following Algorithm 1, we propose a constant-time fixed-weight sparse polynomial sampler. As shown in Fig. 1, this sampler consists of a Barrett reduction unit, two single-port RAMs (one for raw data and the other for reduced indices, both with a 128-bit data width and a 128-to-64-bit shift register), and a control unit. The sampling process involves three phases: the first phase receives and stores raw data, while the remaining two phases correspond to the two outer loops in Algorithm 1. During operation, the raw data RAM initially stores incoming 128-bit random numbers from the Keccak core. Once all data is stored, they are transferred to the shift register and then segmented into four 32-bit chunks to be fed into the Barrett reduction unit. The 32-bit width ensures sufficient randomness for the indices, given that the reduced indices require at most 16 bits. The Barrett reduction unit utilizes pipelined Karatsuba multipliers, optimized for both 16-bit and 8-bit variants, to reduce the critical path and increase operating frequency. After reduction, the indices are stored in the reduced indices RAM. The entire second phase is fully pipelined, ensuring consecutive data processing with a latency of $\mathcal{O}(\omega + 18)$ cycles, where 18 refers to the pipelined delay.

The third phase verifies duplicate index entries in the reduced indices' RAM through the control unit. This phase executes ω sequential rounds, where each round retrieves one index from RAM, stores it in an internal 16-bit register, and compares it against all other stored indices. When duplicates are detected, the control unit writes the current round number to the duplicate locations. The verification process requires $\omega(\omega + 1)/2$ cycles due to exhaustive pairwise comparisons, resulting in $\mathcal{O}(\omega^2)$ time complexity. This approach guarantees uniform randomness in the sampled indices while eliminating collision artifacts. The total cycle count can be expressed as $(\omega(\omega + 3)/2)$, maintaining the constant-time execution.

C. Polynomial Arithmetic Components

The proposed polynomial arithmetic components are responsible for performing calculations related to polynomials involved in HQC. We have designed the components by optimizing the existing designs or designing them from scratch.

Unit-1: Polynomial Multiplier. The polynomial multiplier computes the product of a dense polynomial and a fixed-weight polynomial during KeyGen, Encap, and Decap. The process can be expressed as a matrix-vector product, where the matrix is the rotation matrix of the dense polynomial and the vector represents the sparse polynomial, and can be optimized from an $\mathcal{O}(n^2)$ operation to an $\mathcal{O}(\omega \cdot n)$ process due to the sparsity of the polynomials involved, which is realized by only accumulating the columns of the rotation matrix that correspond to non-zero entries in the sparse polynomial. While many existing works have been reported, there is still room for optimization. For example, the existing designs introduce redundant RAM usage by relying on dual-port RAMs [6] or increasing the number of reading ports [12]. This often leads to high logical complexity and inefficient resource utilization, especially when forming column segments.

Following the strategy mentioned above and drawbacks of the existing works, we have extended the work proposed in [6] with the following optimizations (the proposed polynomial multiplier is shown in Fig. 2): 1) we use single port RAM for storing and fetching the coefficients of the dense polynomial while [6] used dual port RAM; 2) we optimize the segment former so that the register used for storing unused coefficients decreases from $2N_mem$ to N_mem; 3) we introduce a cache to store the last $N - n_1 n_2$ bits to reduce the complexity of the control logic for determining the reading address. In this manner, the whole design leads to fewer resource usage (especially in RAM) and lower computation latency because it avoids the extra cycle used to fetch coefficients from three consecutive addresses, typically the last address (where the final $N - n_1 n_2$ bits reside) and the adjacent one.

Overall, it takes a 16-bit index every N/N_{mem} cycles and an N_{mem}-bit data read from the accumulation RAM every 2 cycles as inputs. It then calculates the corresponding 128-bit coefficient segment using the dense polynomial RAM data and index input, XORs it with the accumulation input, and feeds the resulting N_{mem}-bit output back to the accumulation RAM. The entire multiplication process completes in $\mathcal{O}(\omega(N/N_{mem} + 1))$ time. Note this design is a generic design where N_{mem} can be multiple values (64, 128, 256, etc.). However, to better align with other components in the proposed upper-level architectures, we use $N_{mem} = 128$.

Unit-2: Polynomial Adders. The index-based polynomial adder is responsible for calculating the sum of a dense polynomial and a sparse polynomial. We fully take advantage of the sparse structure of (one of) the involved polynomials. Each time it takes a non-zero index of the sparse polynomial, calculating the address (0 to $\lceil N/N_{mem} \rceil - 1$) and the position (0 to $N_{mem} - 1$) among the bits stored in the address, negating the bit, and finally produces the result by writing the result back to the dense polynomial RAM. This process is depicted by a control unit (that calculates the address and position), a barrel shift register (that shifts '1' by correct bits according to the calculated position), and an N_{mem}-bit long XOR gate to calculate the final output by XORing the shifted '1' and the data read from the dense polynomial RAM. The whole process takes $\mathcal{O}(\omega)$ time to finish the addition of two polynomials.

The other polynomial adder, namely the XOR-based polynomial adder, is responsible for calculating the sum of two dense polynomials. This component is simply implemented by a control unit that calculates the reading address for two RAMs storing the two dense polynomials and N_{mem} XOR gates to calculate the addition under the ambient space $\mathbb{F}_2^n$.

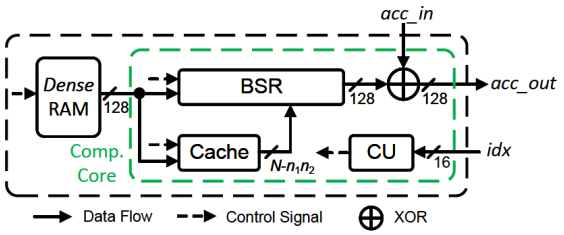


Fig. 2: Proposed polynomial multiplier (CU: control unit; BSR: barrel shift-register).

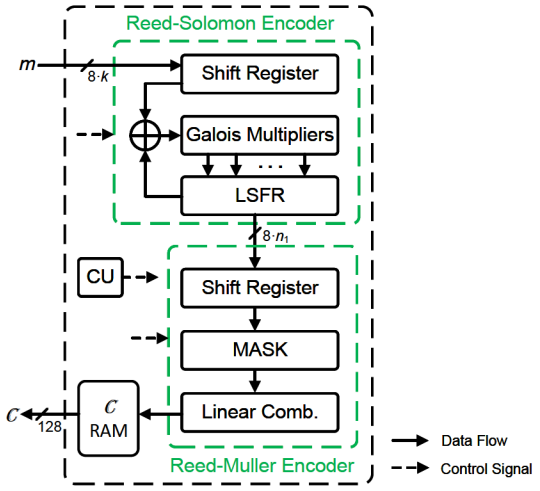


Fig. 3: Internal structure of the proposed Encoder.

The whole addition takes $\mathcal{O}(N/N_{mem})$ time to complete, with one additional reading delay of the RAMs.

D. Encoder

The encoder is responsible for encoding the message m during Encap or Decap by using Reed-Solomon and Reed-Muller encoding. We found that abundant logic is introduced when all Galois Multipliers are individually implemented in [15] and thus can be improved to achieve a better resource usage. The design in [15] uses p individual hard-coded Galois field multipliers to calculate the product of the gate value and coefficients of the generator polynomial, where p is the degree of the generator polynomial. In this paper, we optimized the design by merging all similar hard-coded Galois multipliers to reduce area usage without compromising timing performance.

The proposed encoder consists of Reed-Solomon encoder, Reed-Muller encoder, and a control unit, processing k bytes of message m to produce an n_1n_2-bit codeword $\mathcal{C}$ (see Fig. 3). The encoding begins with the Reed-Solomon encoder, which generates an n_1-byte codeword over k cycles using a systematic linear feedback shift register (LFSR) structured around the Reed-Solomon generator polynomial. The LFSR, initialized to zero, processes one byte of the message per cycle, XORing it with the prior codeword and multiplying the result with polynomial coefficients using parallel Galois field multipliers. The resulting codeword is stored in a shift register and passed byte-by-byte to the Reed-Muller encoder, which performs Reed-Muller encoding to convert each byte into a 128-bit Reed-Muller(1,7) codeword. The Reed-Muller

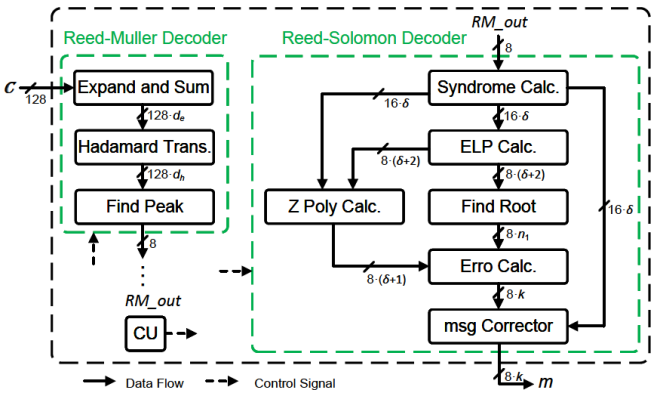


Fig. 4: Internal structure of the proposed Decoder.

encoding involves filling 32-bit chunks with message bits and computing the codeword through XOR gates, AND gates, and bit-shifters. The Reed-Muller encoder completes encoding in $\mathcal{O}(n_1)$ cycles. The final HQC codeword for HQC-1, HQC-3, and HQC-5 consists of 3, 5, or 5 identical copies of the concatenated Reed-Solomon-Reed-Muller codeword, respectively, though only one copy is stored in memory to optimize resources, with repeated reads generating the full codeword.

E. Decoder

Unlike existing design which requires huge resource usage [12] or high computation time [9]. A key feature of the proposed decoder is that it combines a parallel byte-level Reed-Muller decoder with a consistent resource usage Reed-Solomon decoder in order to strike the optimal balance between resource usage and speed. The proposed decoder consists of a Reed-Muller decoder cascaded with a Reed-Solomon decoder to decode the received codeword and recover the original message. It processes an n_1n_2-bit codeword as input through a two-stage decoding pipeline to generate the final k-byte recovered message.

Reed-Muller Decoder. The Reed-Muller decoder consists of three cascaded units: ExpandSum, Hadamard transform, and FindPeak. The decoding process operates in three sequential phases. First, the ExpandSum module receives 128-bit data from the input RAM and performs accumulation. This data then feeds into the Hadamard transform component, which executes a Fast Walsh-Hadamard Transform using 8 hierarchical butterfly operations on 128 parallel data paths with Ping-Pong architecture. Each butterfly stage performs simultaneous additions and subtractions, completing the transform in a fixed 8-cycle latency. The transformed coefficients are then delivered to the FindPeak component. This component implements a 7-stage binary tree search to identify the index of the maximum value. The time complexity of the decoding is $\mathcal{O}(n_1n_2/128 \times (\text{Multiplicity} + 8 + 7))$ cycles, where the Multiplicity is 3, 5, or 5 depending on the security level.

Reed-Solomon Decoder. The Reed-Solomon decoder consists of five cascaded processing modules—syndrome computation, error locator polynomial (ELP) calculation, polynomial $z(x)$ computation, root finding, and error value calculation, along with internal memory buffers and a centralized control

unit. The decoding process operates in five sequential phases corresponding to the classical Berlekamp-Massey algorithm [16] implementation. During the first phase, the syndrome computation module receives 8-bit input data and calculates $2 \cdot \delta$ syndrome values through Galois field multiplications and accumulations, requiring $\mathcal{O}(n_1 \cdot \delta)$ cycles with polynomial evaluation complexity. The second phase employs the ELP module to execute the Berlekamp-Massey algorithm over $2 \cdot \delta$ iterations, where each iteration performs discrepancy calculations, polynomial updates, and degree comparisons using parallel Galois field multipliers, completing in $\mathcal{O}(\delta^2)$ time complexity. The third phase feeds the computed syndromes and ELP coefficients into the z-polynomial module, which calculates the error evaluator polynomial through nested loops with δ outer iterations and variable inner iterations based on the ELP degree, resulting in $\mathcal{O}(\delta^2)$ complexity. The fourth phase adopts additive FFT for root finding, transforming the ELP coefficients to identify error locations through field element evaluation. The final phase employs the error value computation module that uses Forney's algorithm [17] through four sequential states, i.e., initialization, beta calculation, error evaluation, and final error computation, using RAMs for coefficient storage and parallel Galois field arithmetic units.

Overall, our design strategy offers a distinct advantage because its resource usage remains consistent, in contrast to the linear increase seen in [12] as the security level, and thus the codeword length, changes. This leads to superior decoder performance at a higher security level. Also, the smaller, more consistent design area prevents a frequency drop that can occur with larger designs, which improves the performance of the higher-level architecture (Decap).

IV. Innovative Architectural Design for Upper-Level Operations

This section outlines a detailed design process for LEAH, comprising three innovative architectures for KeyGen, Encap, and Decap, respectively. The obtained hardware architectures deploy the components in Section III and follow the novel data flow introduced here for efficient computation.

A. KeyGen

Data Flow Overview. The KeyGen design optimizes performance through concurrent operations (connecting Algorithm 4). This process begins with Keccak generating random data for polynomial h and subsequent sampling. The data flow involves the parallel execution of polynomial multiplication and the sampling of another polynomial, which masks sampling latency and ensures no duplicated hardware. The architecture immediately proceeds to polynomial addition upon completion of multiplication, minimizing idle time to form the final keys.

Architectural Details. The KeyGen includes the following components: a sampler, a polynomial multiplier, an index-based polynomial adder, a control unit, and RAMs for storing both the public key (pk) and secret key (sk), along with intermediate data including non-zero indices of polynomials x and y and raw sampling data from Keccak. The architecture

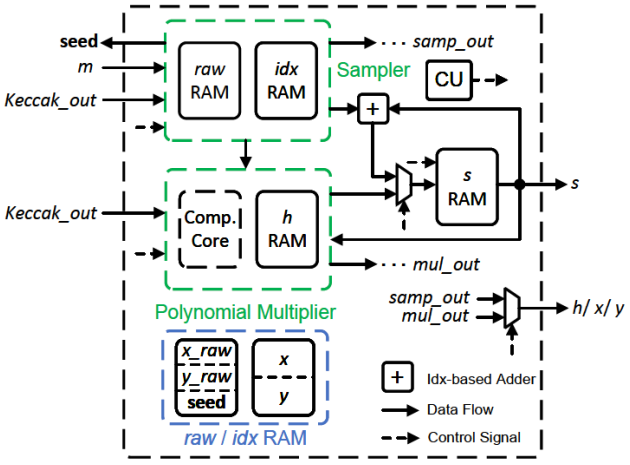


Fig. 5: The proposed architecture for KeyGen.

takes Keccak outputs as its input and generates the final pk and sk upon completion of the entire key generation process.

When the architecture begins to work, it feeds the initial seed **seed1** into the Keccak function. The resulting random output is stored in the h-RAM, where it serves as the coefficients for polynomial h. Subsequently, the module processes **seed2** through Keccak and stores the generated random numbers in the raw data RAM. The first half of this stored data is specifically utilized to sample the non-zero indices of polynomial y. Following sampling of y, two operations are executed in parallel: polynomial multiplication between h and y, and sampling of non-zero indices for polynomial x. This parallel execution strategy effectively conceals the sampling latency within the longer polynomial multiplication operation, thereby optimizing timing performance without requiring duplicate hardware components. Given that polynomial multiplication counts as the dominant time factor compared to sampling, the module proceeds to the polynomial addition phase immediately upon multiplication completion. This stage computes polynomial s and forms the final outputs pk and sk. Note the HQC specification's use of seeds for polynomial representation requires seed re-expansion [5], which adds processing time during Encap and Decap. Our design offers an alternative with a configurable output format. Instead of just seeds, our architecture can also output the actual polynomials, controlled by a selection signal, allowing for faster processing by eliminating the need for re-expansion.

B. Encap

Data Flow Overview. The proposed Encap's data flow is designed for maximum concurrency. This process begins with forming a seed and using it with Keccak to generate random numbers, while simultaneously encoding the message. This is followed by the parallel execution of two polynomial multiplications, which cuts processing time in half. Then three additions compute the ciphertext, and the final stage uses Keccak to calculate the shared secret key. This simultaneous and parallel approach reduces overall latency. Note the con-

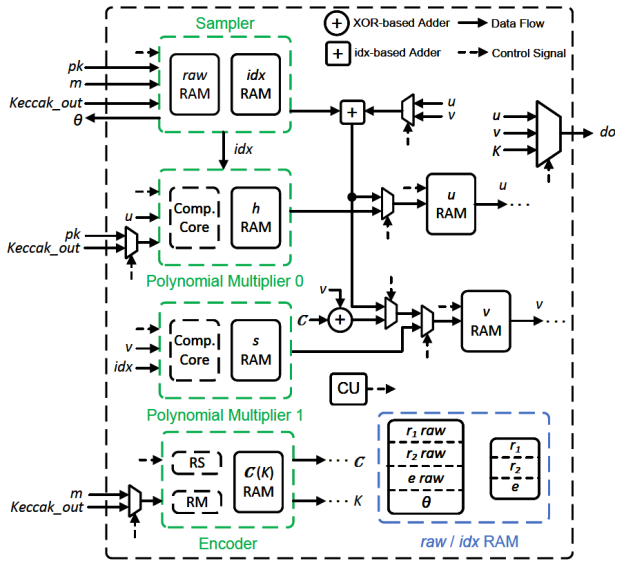


Fig. 6: The proposed architecture for Encap.

figurable polynomial format also contributes to this efficiency by potentially saving the time required for seed re-expansion.

Architectural Details. Overall, the Encap architecture consists of a sampler, two polynomial multipliers, an index-based polynomial adder, a XOR-based polynomial adder, an encoder, a control unit, and RAMs for storing the ciphertext c and the shared secret key K, along with intermediate data such as raw data for sampling r_1 and r_2. The architecture takes the public key pk as input and outputs the ciphertext c as well as the shared secret key K upon completion of the Encap process.

The architecture begins operation by forming the seed θ using the message m, public key pk (**seed1**), and $salt$. The seed θ is then sent to Keccak to await the derivation of random numbers for use as raw data in sampling r_1 and r_2. Concurrently, the encoder begins converting the message m to codeword $\mathcal{C}$, with the encoding time being masked by the Keccak operation. Once all random numbers have been stored, the architecture retrieves h based on the format selection, either through re-generation using **seed1** or by direct reading from RAM. Subsequently, r_2 is sampled and utilized in the next stage, where two polynomial multiplications, $h \cdot r_2$ and $s \cdot r_2$, are executed in parallel. Two distinct polynomial multipliers are employed to perform these multiplications concurrently, as polynomial multiplication dominates the processing time during Encap, and eliminating one multiplication cycle significantly reduces processing time, thereby enhancing overall performance. Meanwhile, the sampler concurrently samples e to save sampling time for subsequent operations. Following this, three polynomial additions are executed serially using the index-based and XOR-based polynomial adders: two dense-sparse polynomial additions and one dense-dense addition, which collectively compute the ciphertext c. The final stage involves sending the message m and the ciphertext c to Keccak to calculate the shared secret key K. Note that the calculated K is stored in the same RAM where the raw data for r_2 resides, thereby reducing the number of memory tiles required

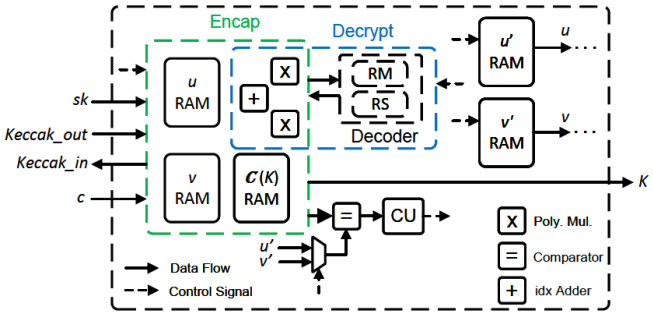


Fig. 7: Decap (RS: Reed-Solomon; RM: Reed-Muller).

in the implementation.

C. Decap

Data Flow Overview. The proposed Decap design achieves high efficiency through its inherited parallel execution and shared memory strategies. This process starts with initial polynomial operations to recover the message, while the encoder's operation is masked by Keccak's random number generation. Re-encryption then begins, where the inherited parallel execution significantly reduces processing time. Following re-encryption, verification is performed using XORs. The final stage involves generating the shared secret key with Keccak and storing it in shared memory to optimize resource usage.

Architectural Details. The proposed Decap architecture consists of a sampler, two polynomial multipliers, an index-based polynomial adder, a XOR-based polynomial adder, an encoder, a decoder, a control unit, and RAMs storing intermediate data such as raw data for sampling r_1, r_2, and e, along with the recovered message and validation results. The architecture takes the public key $pk = (h, s)$ and ciphertext $c = (u, v)$ as input, and outputs the shared secret key K upon completion of the Decap process.

The architecture begins operation by computing $u \cdot y$ using one of the polynomial multipliers, where y is part of the private key. Subsequently, it calculates $v - u \cdot y$ using the XOR-based polynomial adder to obtain the syndrome for error correction. The result is then fed to the decoder, which performs error correction to recover the original message m' from the noisy codeword. Once the message m' is recovered, the architecture proceeds to form the seed θ using the recovered message m', public key components, and $salt$. This seed θ is sent to Keccak to generate random numbers for sampling r_1, r_2, and e. Concurrently, the encoder begins converting the recovered message m' back to codeword $\mathcal{C}$ for verification purposes, with the encoding time being masked by the Keccak operation and sampling processes. After all random numbers are generated and stored, the sampler sequentially samples r_2, r_1, and e from the raw data. The architecture then performs two polynomial multiplications in parallel: $h \cdot r_2$ and $s \cdot r_2$, using two distinct polynomial multipliers to maximize performance. This parallel execution significantly reduces processing time since polynomial multiplication is the most computationally intensive operation during Decap. Following the multiplications, three polynomial additions are executed serially using the index-based and XOR-based polynomial adders. These operations

TABLE II: Comparison With The Existing Works for Sampler

Design	Device	LUT	FF	DSP	BRAM	Slice	Fmax[1]	Latency[2]	Delay[3]	ELUT[4]	EADP[5]	EADPR[6]
HQC-1												
[9]	Artix-7	201	229	4	1	67	201	3,062	15.23	1,956	29,791	59.80%
[12]	Artix-7	1,055	1,189	0	2	NA	230	2,335	10.15	1,287	13,070	8.37%
TW	Artix-7	1,046	1,370	0	0.5	455	279	3,037	10.85	1,104	11,976	NA
HQC-3												
[9]	Artix-7	221	245	5	1	71	200	6,817	34.09	2,385	81,300	67.37%
[12]	Artix-7	1,010	1,178	0	2	389	237	5,207	21.97	1,242	27,296	2.83%
TW	Artix-7	1,027	1,289	0	0.5	482	279	6,820	24.44	1,085	26,525	NA
HQC-5												
[9]	Artix-7	216	248	5	1	72	204	11,487	56.31	2,380	134,026	65.92%
[12]	Artix-7	1,027	1,172	0	2	393	225	8,834	39.26	1,259	49,447	7.63%
TW	Artix-7	1,049	1,360	0	0.5	452	279	11,510	41.25	1,107	45,673	NA

Note: The notations below apply to the other tables in this paper.
[1]: Fmax: Maximum frequency. Unit: MHz [2]: Latency: Calculation latency (number of cycles). [3]: Delay = Latency/Fmax. unit: μs.
[4]: ELUT: Equivalent LUT, following [18]. 1 DSP = 102.4 Slices; one 18K BRAM = 116.2 Slices, 1 Slice = 4 LUTs.
[5]: EADP: Equivalent ADP. EADP = #ELUT$\times$delay. [6]: EADPR: EADP reduction. TW: This work.

TABLE III: Comparison With The Existing Works for Polynomial Multiplier

Design	Device	LUT	FF	BRAM	Slice	Fmax	Latency	Delay	ELUT	EADP	EADPR
HQC-1											
[9]	Artix-7	1,439	496	4	486	238	9,374	39.39	1,904	74,984	11.88%
[6]	Artix-7	2,453	663	4	678	196	9,241	47.15	2,918	137,568	51.97%
TW	Artix-7	1,628	291	2	564	264	9,374	35.52	1,860	66,079	NA
HQC-3											
[9]	Artix-7	1,445	500	4	488	240	28,042	116.84	1,910	223,144	10.15%
[6]	Artix-7	2,370	664	4	657	210	27,901	132.86	2,835	376,637	46.77%
TW	Artix-7	1,662	298	2	539	265	28,402	105.83	1,894	200,485	NA
HQC-5											
[9]	Artix-7	1,448	474	4	488	240	59,476	247.82	1,913	474,024	13.79%
[6]	Artix-7	2,247	665	8	648	207	59,213	286.05	3,177	908,676	55.03%
[12](1 read port)	Artix-7	1,442	1,315	6.5	348	254	59,744	235.21	2,197	516,833	20.93%
[12](4 read ports)	Artix-7	5,101	3,903	11	1,254	233	15,057	64.62	6,379	412,239	0.86%
TW	Artix-7	1,609	293	2	528	268	59,476	221.94	1,841	408,674	NA

compute $u' = r_1 + h \cdot r_2$ and $v' = \mathcal{C} + s \cdot r_2 + e$, reconstructing the expected ciphertext components for verification. The architecture includes comparators that continuously check whether $u = u'$ and $v = v'$ throughout the addition processes. The final stage involves validation and key generation. If the reconstructed ciphertext matches the input ciphertext (indicating successful decapsulation), the architecture sends the recovered message m' and verified ciphertext components to Keccak to calculate the shared secret key K. The calculated K is stored in the same RAM previously used for intermediate data, optimizing memory utilization and reducing the number of required memory tiles in the implementation.

V. Implementation Results

A. Experimental Setup

The experimental setup is as follows: (i) all the designs (under different parameter settings) are coded in VHDL, simulated and tested through ModelSim, and finally implemented using Vivado 2023.2; (ii) we have used the NIST-recommended Artix-7 (XC7a200t) FPGA to implement LEAH; (iii) key results obtained from the implementation, including resource usage, maximum frequency (Fmax), and delay, are listed and compared with the existing works; (iv) we have followed the approach used in [6], [18] to calculate Equivalent Area-Delay Product (EADP) for a balanced area-time overall consideration (see the footnote of tables);

B. Performance Discussion

Sampler. As shown in Table II, the proposed sampler design demonstrates competitive performance across all HQC parameter sets. For HQC-1, our design achieves the highest operating frequency of 279 MHz while maintaining reasonable resource utilization with and minimal BRAM usage. Compared to [9], our design uses 5.2$\times$ more LUTs but achieves 38.8% higher frequency and reduces BRAM requirements by 50%. For HQC-3 and HQC-5, our sampler maintains consistent performance with slight increases in latency but achieves competitive EADP, better than [9] and [12].

Polynomial Multiplier. Table III reveals that our polynomial multiplier achieves excellent area-time efficiency across all parameter sets. For HQC-1, our design operates at 264 MHz with 1,628 LUTs and only 2 BRAMs, which represents an 11.9% improvement over [9] and 51.9% better than [6]. Notably, our design uses 50% fewer BRAMs compared to existing works while achieving higher operating frequencies. For HQC-5, our multiplier significantly outperforms the single-port design [12] by 20.9% in EADP while using substantially fewer resources than their 4-port variant.

TABLE IV: Comparison With The Existing Works for Encoder

Design	Device	LUT	FF	Slice	BRAM	Fmax	Cycle	Delay	ELUT	EADP	EADPR
HQC-1											
[15]	Artix-7	680	617	NA	2	228	62	0.27	1,610	438	19.91%
[9]	Artix-7	858	922	280	2	270	97	0.36	1,788	642	45.41%
[12]	Artix-7	532	620	269	2	374	148	0.40	1,462	578	24.32%
[11]	Artix-7	2,019	603	NA	0	152	7,244	47.66	2,019	96,221	99.64%
TW	Artix-7	597	863	294	2	270	62	0.23	1,527	351	NA
HQC-3											
[15]	Artix-7	906	845	NA	2	235	80	0.34	1,836	625	3.55%
[12]	Artix-7	570	644	279	2	394	290	0.74	1,500	1,104	43.39%
[9]	Artix-7	1,011	1,088	358	2	298	131	0.44	1,941	853	29.35%
TW	Artix-7	871	1,099	331	2	239	80	0.33	1,801	603	NA
HQC-5											
[15]	Artix-7	1,205	1,329	NA	2	236	112	0.47	2,135	1,013	3.19%
[12]	Artix-7	743	854	322	2	386	460	1.19	1,673	1,993	49.18%
[9]	Artix-7	1,503	1,689	514	2	294	189	0.64	2,433	1,564	37.28%
TW	Artix-7	1,137	1,326	392	2	236	112	0.47	2,067	981	NA

TABLE V: Comparison With The Existing Works for Decoder

Design	Device	LUT	FF	BRAM	Slice	Fmax	Latency	Delay	ELUT	EADP	EADPR
HQC-1											
[9]	Artix-7	2,817	3,779	2.5	952	205	4,611	22.49	4,270	96,033	42.26%
[12]	Artix-7	5,896	3,364	2.5	1,794	219	1,293	5.90	7,349	43,386	-27.80%
TW	Artix-7	4,014	7,133	2.5	1,843	231	2,343	10.14	5,467	55,446	NA
HQC-3											
[9]	Artix-7	3,257	4,727	2.5	1,100	212	5,485	25.87	4,710	121,847	43.79%
[12]	Artix-7	7,219	3,561	2.5	2,125	219	1,685	7.69	8,672	66,719	-2.65%
TW	Artix-7	4,435	7,998	2.5	1,996	229	2,664	11.63	5,888	68,490	NA
HQC-5											
[9]	Artix-7	3,679	5,574	2.5	1,240	206	9,199	44.66	5,132	229,149	42.87%
[12]	Artix-7	10,090	4,472	2.5	2,843	225	2,705	12.02	11,543	138,767	5.67%
TW	Artix-7	4,856	8,844	2.5	2,166	232	4,814	20.75	6,309	130,901	NA

TABLE VI: Comparison With The Existing Works for KeyGen

Design	Device	LUT	FF	DSP	BRAM	Slice	Fmax	Latency	Delay	ELUT	EADP	EADPR
HQC-1												
[9]	Artix-7	2,369	901	4	10	809	179	15,674	87.56	5,169	452,655	96.74%
[12]	Artix-7	6,827	5,444	0	20	3,067	208	6,203	29.82	9,151	272,902	44.65%
TW	Artix-7	2,834	1,626	0	7	959	218	11,276	51.72	3,647	188,661	NA
HQC-3												
[9]	Artix-7	2,342	926	5	10	791	189	42,106	222.78	5,552	1,236,892	85.47%
[12]	Artix-7	7,150	5,462	0	20	3,148	207	14,940	72.17	9,474	683,776	4.79%
TW	Artix-7	2,841	1,882	0	7	971	220	39,281	178.55	3,654	652,493	NA
HQC-5												
[9]	Artix-7	2,342	942	5	10	791	188	82,331	437.93	5,552	2,431,392	96.57%
[12]	Artix-7	6,845	5,457	0	20	3,072	201	27,825	138.43	9,169	1,269,291	5.28%
TW	Artix-7	2,782	1,594	0	7	912	220	77,369	351.68	3,428	1,205,620	NA

Encoder. The encoder comparison in Table IV shows our design achieving optimal cycle counts across all parameter sets. For HQC-1, our encoder completes encoding in just 62 cycles at 270 MHz, matching the best cycle count from [15] while achieving better timing performance. The EADP of 351 represents the most efficient implementation among all compared designs. For HQC-3, our design maintains 80 cycles with competitive resource usage of 871 LUTs and achieves an EADP of 603, significantly better than other implementations. The HQC-5 variant shows similar efficiency with 112 cycles and an EADP of 981, demonstrating consistent performance scaling while maintaining the lowest area-time product.

Decoder. Table V shows that our decoder design provides a more balanced approach compared to [9], using more FFs but achieving better timing performance. When compared to [12], though the proposed design has slightly larger EADP at HQC-1 and HQC-3. Nevertheless, the consistency in resource usage offers a distinct advantage over the linear increase seen in [12] when the security level changes to HQC-5, i.e., a 5.67% reduction in EADP (which also leads to upper-level architecture efficiency, see the following comparisons).

KeyGen. The results in Table VI show that our design achieves significant area-time efficiency. For HQC-1, our design operates at 218 MHz with 2,834 LUTs and achieves an EADP of 188,661, representing substantial improvements over existing works. Compared to [9], we achieve 58.3% better

TABLE VII: Comparison With The Existing Works for Encap

Design	Device	LUT	FF	DSP	BRAM	Slice	Fmax	Latency	Delay	ELUT	EADP	EADPR
HQC-1												
[9]-I	Artix-7	4,145	2,128	4	13	1,400	179	28,217	157.64	7,294	1,149,803	57.15%
[9]-II	Artix-7	5,828	2,531	4	15	1,969	179	22,423	125.27	9,209	1,153,645	57.30%
[12]	Artix-7	7,064	6,378	0	20	3,126	168	13,314	79.25	9,388	743,999	33.78%
TW	Artix-7	5,455	3,280	0	12.5	1,756	199	14,193	71.32	6,908	492,654	NA
HQC-3												
[9]-I	Artix-7	4,278	2,412	5	15	1,445	182	90,346	496.41	8,069	4,005,505	66.78%
[9]-II	Artix-7	6,434	2,531	4	15	2,174	196	57,314	292.42	9,815	2,870,203	53.65%
[12]	Artix-7	7,629	6,427	0	20	3,268	179	31,460	179.77	9,953	1,789,265	25.64%
TW	Artix-7	5,459	3,397	0	12.5	1,833	198	38,115	192.50	6,912	1,330,464	NA
HQC-5												
[9]-I	Artix-7	4,809	3,041	5	15	1,625	182	177,154	973.37	8,600	8,371,013	67.56%
[9]-II	Artix-7	6,898	3,417	5	15	2,330	196	108,527	553.71	10,689	5,918,597	54.12%
[12]	Artix-7	7,406	6,631	0	20	3,212	187	58,588	313.30	9,730	3,048,456	10.92%
TW	Artix-7	5,740	4,047	0	12.5	1,993	200	75,508	377.54	7,193	2,715,456	NA

TABLE VIII: Comparison With The Existing Works for Decap

Design	Device	LUT	FF	DSP	BRAM	Slice	Fmax	Latency	Delay	ELUT	EADP	EADPR
HQC-1												
[9]-I	Artix-7	8,984	6,596	4	20	3,035	192	48,212	251.10	20,604	5,173,750	49.87%
[9]-II	Artix-7	10,959	7,003	4	22	3,368	179	37,197	207.80	23,741	4,933,486	47.42%
[12]	Artix-7	13,327	9,488	0	20.5	3,496	167	19,880	119.04	25,238	3,004,320	13.66%
TW	Artix-7	9,878	10,212	0	17.5	3,687	191	24,715	129.40	20,046	2,593,846	NA
HQC-3												
[9]-I	Artix-7	10,931	9,424	5	22	3,693	186	248,338	1335.15	23,713	31,660,425	49.50%
[9]-II	Artix-7	12,868	9,794	5	24	4,347	186	179,711	966.19	26,812	25,905,437	38.28%
[12]	Artix-7	18,310	10,989	0	27.5	6,448	151	86,028	569.72	34,288	19,534,338	18.15%
TW	Artix-7	11,304	11,050	0	17.5	3,796	186	138,509	744.67	21,472	15,989,226	NA
HQC-5												
[9]-I	Artix-7	10,931	9,424	5	22	3,693	186	248,338	1335.15	23,713	31,660,425	49.50%
[9]-II	Artix-7	12,868	9,794	5	24	4,347	186	179,711	966.19	26,812	25,905,437	38.28%
[12]	Artix-7	18,310	10,989	0	27.5	6,448	151	86,028	569.72	34,288	19,534,338	18.15%
TW	Artix-7	11,304	11,050	0	17.5	3,796	186	138,509	744.67	21,472	15,989,226	NA

EADP while using comparable resources but eliminating DSP usage entirely. The [12] design, while faster in terms of latency, requires 2.4× more LUTs and 2.9× more BRAMs. For HQC-3 and HQC-5, our design maintains consistent resource usage and achieves EADP values that significantly outperform [9] and provide better resource efficiency than [12].

Encap. Table VII demonstrates that our Encap design achieves excellent performance across all parameter sets. For HQC-1, our implementation operates at 199 MHz with 5,455 LUTs and completes encapsulation in 14,193 cycles, achieving an EADP of 492,654. This represents significant improvements over existing works, with 57.2% better EADP than [9]-I and 33.8% better than [12]. Our design uses 37.5% fewer BRAMs than [12] while maintaining competitive timing. For HQC-5, our Encap achieves 75,508 cycles at 200 MHz with an EADP of 2,715,456, outperforming [9] variants by 67.6% and 54.1% respectively, while requiring fewer resources and achieving better frequency than most competing designs.

Decap. The Decap comparison in Table VIII shows our implementation achieving balanced performance with good resource efficiency. For HQC-1, our design completes decapsulation in 24,715 cycles at 191 MHz using 9,878 LUTs, resulting in an EADP of 2,593,846. While [12] achieves lower latency, it requires 34.9% more LUTs and 17.1% more BRAMs. Our design provides better resource efficiency compared to [9] variants while maintaining competitive timing performance. For HQC-5, our Decap processes 138,509 cycles with an EADP of 15,989,226, demonstrating substantial improvements over [9] designs (49.5% and 38.3% better EADPR, respectively) while using significantly fewer resources than [12].

Discussion. Overall, the efficiency of the proposed LEAH comes from: (i) At the component level, the proposed sampler, polynomial multiplier, and encoder all achieve lower area-time delay than the others. Moreover, the proposed decoder achieves consistently low resource consumption across three security levels. (ii) At the upper system level, due to the use of novel data flow, the architectures of KeyGen, Encap, and Decap achieve better performance than the existing ones.

VI. Conclusion

This paper has presented LEAH, which includes optimized components such as the fixed-weight sampler, polynomial multiplier, encoder, and decoder, as well as three generic architectures for KeyGen, Encap, and Decap that strictly adhere to the latest HQC specification. The proposed design achieves significant improvements in latency and EADP while maintaining compatibility with resource-constrained FPGA platforms. Experimental results on Artix-7 device demonstrate superior performance compared to existing state-of-the-art implementations, making our accelerator a practical solution for post-quantum secure applications. This work advances the

field of code-based cryptography and provides a foundation for further optimizations and deployments of HQC in real-world scenarios. We hope this research will contribute to the ongoing PQC standardization efforts and inspire future innovations in efficient hardware implementations of PQC schemes.

ACKNOWLEDGMENT

This was supported by Villanova College of Engineering Last-Year Ph.D. Fellowship and in part by NSF CNS-2453370.

REFERENCES

[1] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, 1994.

[2] J. Xie, K. Basu, K. Gaj, and U. Guin, "Special session: The recent advance in hardware implementation of post-quantum cryptography," in *2020 IEEE 38th VLSI Test Symposium (VTS)*, pp. 1–10, 2020.

[3] "Announcing Approval Three as Federal Information Processing Standards FIPS for Post-Quantum Cryptographuy." https://csrc.nist.gov/news/2024/postquantum-cryptography-fips-approved.

[4] "NIST Selects HQC as Fifth Algorithm for Post-Quantum Ecryption." https://www.nist.gov/news-events/news/2025/03/nist-selects-hqc-fifth-algorithm-post-quantum-encryption.

[5] C. A. Melchor *et al.*, "Hamming Quasi-Cyclic (HQC) (Fourth Round Version)." https://pqc-hqc.org/doc/hqc-specification$_2025-02-19.pdf$.

[6] Y. Tu, P. He, K. Koç, and J. Xie, "LEAP: Lightweight and efficient accelerator for sparse polynomial multiplication of HQC," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 6, pp. 892–896, 2023.

[7] P. He, Y. Tu, T. Bao, c. c. Koç, and J. Xie, "HSPA: High-throughput sparse polynomial multiplication for code-based post-quantum cryptography," *ACM Trans. Embed. Comput. Syst.*, vol. 24, Dec. 2024.

[8] P. He, Y. Tu, and J. Xie, "LOCS: LOw-latency and ConStant-timing implementation of fixed-weight sampler for HQC," in *2023 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2023.

[9] S. Deshpande, C. Xu, M. Nawan, K. Nawaz, and J. Szefer, "Fast and efficient hardware implementation of HQC." Cryptology ePrint Archive, Paper 2022/1183, 2022.

[10] C. Li, S. Song, J. Tian, Z. Wang, and K. Koç, "An efficient hardware design for fast implementation of hqc," in *2023 IEEE 36th International System-on-Chip Conference (SOCC)*, pp. 1–6, 2023.

[11] C. Aguilar-Melchor, J.-C. Deneuville, A. Dion, J. Howe, R. Malmain, V. Migliore, M. Nawan, and K. Nawaz, "Towards automating cryptographic hardware implementations: A case study of HQC," in *Code-Based Cryptography: 10th International Workshop, CBCrypto 2022, Trondheim, Norway, May 29–30, 2022, Revised Selected Papers*, (Berlin, Heidelberg), p. 62–76, Springer-Verlag, 2022.

[12] F. Antognazza, A. Barenghi, G. Pelosi, and R. Susella, "A high efficiency hardware design for the post-quantum KEM HQC," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 431–441, 2024.

[13] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A modular analysis of the Fujisaki-Okamoto transformation," in *Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part I*, (Berlin, Heidelberg), p. 341–371, Springer-Verlag, 2017.

[14] P. He, Y. Tu, T. Bao, L. Sousa, and J. Xie, "COPMA: compact and optimized polynomial multiplier accelerator for high-performance implementation of LWR-based PQC," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 31, no. 4, pp. 596–600, 2023.

[15] Y. Tu, P. He, C.-H. Chang, and J. Xie, "LTE: Lightweight and time-efficient hardware encoder for post-quantum scheme HQC," *IEEE Computer Architecture Letters*, vol. 23, no. 2, pp. 187–190, 2024.

[16] J. Massey, "Shift-register synthesis and bch decoding," *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969.

[17] G. Forney, "On decoding bch codes," *IEEE Transactions on Information Theory*, vol. 11, no. 4, pp. 549–557, 1965.

[18] W. Liu, S. Fan, A. Khalid, C. Rafferty, and M. O'Neill, "Optimized schoolbook polynomial multiplication for compact lattice-based cryptography on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 10, pp. 2459–2463, 2019.