An Efficient and Unified RTL Accelerator Design for HQC-128, HQC-192, and HQC-256

Francesco Antognazza, Alessandro Barenghi, Gerardo Pelosi *Member, IEEE*.

Abstract—In the Post-Quantum Standardization (PQC) process held by the National Institute of Standards and Technology (NIST), the final round of evaluation of the asymmetric cryptographic schemes Classic McEliece, BIKE and HQC will elect the alternative Key Establishment Mechanism (KEM) to the FIPS 203 standard CRYSTALS-Kyber. In this work we present two configurations of a RTL hardware design of the HQC candidate, either optimized for devices exclusively working with client-server style protocols, or a unified accelerator compatible with all KEM operations, i.e. Key Generation, Encapsulation, and Decapsulation. Our designs are compatible with all the parameter sets defined by the HQC specification, providing security margins equivalent to the ones of AES-128, AES-192, and AES-256 based on a selection made at runtime. We are providing an extensive comparison with the current state-of-the-art RTL hardware designs for Artix-7 FPGAs of the schemes in the PQC process, introducing a new metric to evaluate the area utilization, historically a challenging task for such devices made of heterogeneous resources, and determining that HQC has by far the best figures among the code-based candidates in terms of latency, area occupied and efficiency, and even comparable with the lattice-based CRYSTALS-Kyber when using the parameters with lowest security margin.

Index Terms—Hardware security, Post-Quantum Cryptography, Code-based Cryptography, Key Establishment Mechanism

1 Introduction

In recent times, a significant amount of attention has been given to the design and engineering of quantum computers, aiming at obtaining high-qubit-count, reliable quantum computers. One of the most pressing challenges introduced by these advancements is the development of new asymmetric cryptographic algorithms whose security is based on hard problems not reducible to those that quantum computers can solve exponentially faster than a classical computer, such as integer factorization and discrete logarithm. For this reason, a paradigm shift towards post-quantum cryptography is required, which focuses on constructing cryptographic algorithms that remain secure even against adversaries with quantum computation capabilities.

In 2016, the National Institute of Standards and Technology, a US government agency promoting globally adopted standards like FIPS 202 [1] for the Secure Hash Algorithm SHA-3, announced the Post-Quantum cryptography Standardization contest [2], a public call for the proposals of new Key Establishment Mechanisms (KEM) and Digital Signatures (DS) schemes being able to withstand both classical and quantum-computer-aided cryptanalysis.

In November 2017, among the 82 algorithms submitted, 69 met the minimum acceptance criteria. In July 2020, NIST announced that Classic McEliece, CRYSTALS-Kyber, NTRU, and SABER were advancing to the third and final evaluation round in the KEM category, prior to select one of them for standardization. A similar procedure was followed with CRYSTALS-Dilithium, FALCON, and Rainbow for the Digital Signatures category. In addition, as most of the finalists relied on lattice-based problems, the BIKE, FrodoKEM, HQC, NTRU Prime, SIKE, GeMSS, Picnic, and SPHINCS+ were selected to proceed on a parallel standardization branch as alternate candidates to promote variety of the mathematical frameworks underlying the schemes to be included in a portfolio of post-quantum algorithms. The third selection round lasted two years, proving that a thorough evaluation of a new asymmetric algorithm is exceptionally challenging. In July 2022 NIST reported that CRYSTALS-Kyber would be standardized as the Key Establishment Mechanism [3], and that for the digital signatures category would proceed to standardize all the remained proposals: CRYSTALS-Dilithium, SPHINCS+, and FALCON [4]. Finally, in the track for alternate candidates selection, three KEMs based on the hardness of the syndrome decoding problem, namely BIKE [5], Classic McEliece [6], and HQC [7] advanced for a fourth round of evaluation.

Among these candidates, HQC received a broad interest from the industrial and academic communities as it exhibits good performance figures as well as solid security guarantees coming from the hardness of its underlying random quasi-cyclic (QC) syndrome decoding problem, without relying on the indistinguishability of a hidden QC medium density parity check code from a QC random parity check code. By contrast, Classic McEliece relies on the indistinguishability assumption of a hidden Goppa code from a random code, while BIKE relies on the indistinguishability assumption of the systematic form of a hidden QC-MDPC code from the systematic form of a random QC code.

Compared to the lattice-based CRYSTALS-Kyber scheme, which is the current NIST standard in the KEM category, the code-based HQC scheme shares the same high-level algorithmic construction applied on a different mathematical framework algebraic. The CRYSTALS-Kyber scheme works with the elements of a polynomial ring that

Francesco Antognazza, Alessandro Barenghi, and Gerardo Pelosi are affiliated with the Department of Electronics, Information and Bioengineering of Politecnico di Milano, 20133 MI, Italy. E-mail: {name.surname}@polimi.it

is highly friendly to the usage of the Number Theoretic Transform (NTT) to realize multiplication operations with the best asymptotic complexity. In addition, it benefits from a quite limited ciphertext size due to the introduction of the so called *compression* and *decompression* algorithms that mimic fast and simple decoding and encoding strategies, respectively, for transforming bit strings to polynomials and viceversa. The HQC scheme enjoys working in an algebraic structure where highly efficient multiplication operations can be obtained with an algorithmic strategy that is simpler and more convenient than the NTT one. Specifically, denoting with n the maximum degree of a polynomial, all the multiplications prescribed by the HQC specification consider as operands a *dense* and a *sparse* polynomial. A dense polynomial exhibits an average number of non-null binary coefficients which is a fraction of n, while a sparse polynomial always exhibits a number of non-null binary coefficients that is $\approx \sqrt{n}$. Applying the NTT algorithm to perform these multiplications with n in the range prescribed by HQC specification results to be less performant than algorithms leveraging the specific form of the operands. A similar conclusion, also fast implementations related to which amounts to a fraction of the degree of the polynomial. Moreover, the HQC encoding and decoding algorithms are also realized in such a way to fully exploit the correction power of the underlying error correction code minimizing the ciphertext size in compliance with parameter configurations exhibiting the prescribed security level.

Both CRYSTALS-Kyber and HQC can be configured to provide a security level addressing all categories required by NIST, namely cat1, cat3, cat5 equivalent the strength of `AES-128`, `AES-192`, and `AES-256`, respectively, against both classical and quantum cryptanalysis techniques.

A comparison between CRYSTALS-Kyber and HQC in terms of keys and ciphertext sizes shows the following triples of byte counts, one for each NIST category: $(1632, 2400, 3168)$ and $(2305, 4586, 7317)$ secret key sizes, $(800, 1184, 1568)$ and $(2249, 4522, 7245)$ public key sizes, $(768, 1088, 1568)$ and $(4433, 8978, 14421)$ ciphertext sizes.

From the perspective of software implementations, the design of CRYSTALS-Kyber spurred a flourishing line of research results; most of them focusing on the realization of NTT-based optimized polynomial multiplication across various platforms and architectures, especially AVX2 and ARM Cortex-M [8]. Software implementations of HQC include the library provided by the authors of the cryptosystem with a portable C implementation and an AVX2 optimized implementation targeting x86_64 platforms [9], and a recent study showing an optimized implementation targeting an ARM Cortex-M CPU [10]. More recently, several research efforts focused on the side-channel vulnerabilities of HQC software implementations and the corresponding countermeasures [11], [12]. From the hardware design standpoint, several designs for the CRYSTALS-Kyber scheme have been proposed [13], [14], studying the structure of the NTT-based multiplication.

The use of a dedicated hardware accelerator plays an important role in relevant use cases employing TLS communication sessions and the ones requiring hardened, tamper-resistant devices (e.g., HSM), where cryptographic tasks are assigned to specialized hardware modules for performance and/or security reasons. Recent studies [15] about the performance of TLS 1.3 communication sessions consider a statistically significant number of endpoints, under different conditions in terms of latency (including local, regional, continental and intercontinental connection paths) and packet loss rate (e.g., 3%–5%). In particular, these studies consider modified browsers and edge servers to support the use of available (not yet standardized) post-quantum KEMs combined with traditional elliptic curve Diffie-Hellman, to the end of applying the so-called "hybrid" key agreement during the initial handshake steps of the protocol. The high-level conclusions of the authors in [15] maintain that on fast, reliable network links, the completion time of TLS handshakes (using hybrid key agreement) of the median connection is dominated by the cost of public key cryptography, whereas the 95th percentile completion time is not substantially affected due to excessive congestions on some links. Analogously, on unreliable network links with packet loss rates of 3%–5% or higher, the time spent on data (re)transmissions come to govern handshake completion time. The relative cost of TLS handshake increases as a function of size of application data to be transmitted. This become particularly relevant when considering edge or IoT devices or application with intermittent TLS connections.

The RTL FPGA design appeared in [16] focuses on the study of HQC polynomial multipliers, while the one in [17] considers only the components of the HQC scheme concerning the sampling of random polynomials, without being compliant with the official HQC specification.

A complete design for each of the HQC's cryptographic primitives (i.e., Key Generation, Encapsulation, and Decapsulation), each of which tailored for a specific parameter set guaranteeing a single security margin equivalent to the one of `AES-128`, `AES-192`, or `AES-256`, respectively, is provided in [18] as a High-Level Synthesis description, and in [19]–[21] as optimized high-performance RTL designs. In particular, the design in [21] includes a polynomial multiplier that outperforms other solutions in terms of latency by a $2\times$–$5\times$ factor, depending on the security level.

Contributions. This work presents an efficient unified hardware design at the Register Transfer Level (RTL) for the HQC KEM hinging upon the separate designs of the HQC primitives (Key Generation, Encapsulation, and Decapsulation), and their specializations for the three distinct security levels mandated by the specification of the cryptoscheme, which appeared in [21]. The new unified design exhibits the functionalities of the three primitives into a single component benefiting from the observation that the Decapsulation module includes all the components needed to realize the operations prescribed by the Key Generation and Encapsulation primitives. This is obtained by providing two main enhancements. The first one consists of a careful extension of the control logic of the design of the modules implementing the Key Generation, Encapsulation, and Decapsulation primitives by realizing a global Finite State Automata (FSA) that orchestrates the FSAs of the said modules with a negligible crossbar overhead. The second major enhancement concerns the possibility to configure the unified component to work with any of the three distinct security levels described in the specification of the cryp-

toscheme. Such a goal requires a thoughtful analysis of the sub-algorithms of HQC and important design changes for realizing a unified Reed-Solomon (RS) encoder and decoder supporting the algebraic operations used by each forward error correction code associated with each security level. Differently from a simple juxtaposition of modules and sub-modules implementing the RS decoder operations for each security level, we made an effective use of the algebraic properties of the generator polynomials of the supported error correction codes obtaining a $\times 2$ and $\times 2.28$ reduction in terms of LUTs and FFs, respectively. The unified RS encoder, on the other hand, benefits from a $\times 2.16$ reduction in terms of FFs due to the sharing of the same Linear Feedback Shift Register (LFSR) structure needed to manage the non-simultaneous operations of the three error corrections codes. This unified design substantially improves in terms of flexibility and performance the designs in [19], [20] that considers only a HQC engine specialized to execute the key generation, encapsulation and decapsulation with the cat1 parameters. From the performance standpoint, our solution exhibits gains in execution time equal to $2.28\times$, $1.53\times$, and $1.63\times$ for key generation, encapsulation and decapsulation, respectively, when compared with [19] and $2.87\times$, $2.74\times$, and $3.07\times$ when compared with [20]. A notable improvement of the design introduced in this work lies in the flexibility to configure the same component to work either with parameters addressing cat1 for the best performance in terms of execution time, or with parameter addressing the higher security categories when required by the tasks making use of the accelerator. Considering the FPGA area consumption, our component requires 11% more resources than [19], and 40% less resources than [20].

Other than the performance figures about the fully unified design of an HQC accelerator meant to be employed for the establishment of mutually authenticated communication session, we also describe a client-server scenario with server only authentication, where the server is supported with a unified engine realizing only the key generation and decapsulation primitives, while the client benefits from a separate engine implementing the encapsulation algorithm only (for all possible parameter sets). In the latter case, we show a $2\times$ better area occupancy at the client side, which may be desirable in some applications, and an improvement in the working frequency of both the server engine (due to a simpler interconnection network and FSA) and the client engine (due to the lack of many unused operations).

Furthermore, we included a comprehensive performance comparison among the state-of-the art RTL hardware accelerators targeting an Artix-7 FPGA – the official platform required by NIST. Specifically we compare the CRYSTALS-Kyber, BIKE, Classic McEliece, and Streamlined NTRU Prime schemes, depending on the availability of a fully unified design, or a design tailored for the client-server scenario with fixed roles of the endpoints. The comparison is performed introducing also a refined metric that we firstly introduced in [21], which in turn enables to fairly evaluate jointly the area utilization and efficiency figures – an historically challenging tasks for FPGA devices with heterogeneous resources (LUTs, FFs, BRAMs, DSPs).

Finally, to properly compare the execution times of hardware and software realizations of HQC (e.g., for the realization of a HSM), we compare the latency of our hardware design with the runtime on a Rockchip RK3288 ARM Cortex-A17 CPU at 1.8MHz. This CPU is produced with a 28nm HKMG process node technology, which is the nearest to the 28nm HPL technology employed for the lineup of our Artix-7 platform. Furthermore, also the bulk price of 20 US $ applies to both RK3288 and XC7A35T platforms. We note that our design can be accommodated in each platform of the Artix-7 lineup (XC7A12T–XC7A200T), and the XC7A35T one is the least expensive. The sum of the runtime execution of key generation, encapsulation, and decapsulation algorithm amounts to (4.94, 10.2, 17.7) ms, for the three parameter sets, while our FPGA unified accelerator takes (0.25, 0.60, 1.13) ms, which makes it $19.8\times$, $16.9\times$, and $15.6\times$ faster than the execution on the said CPU.

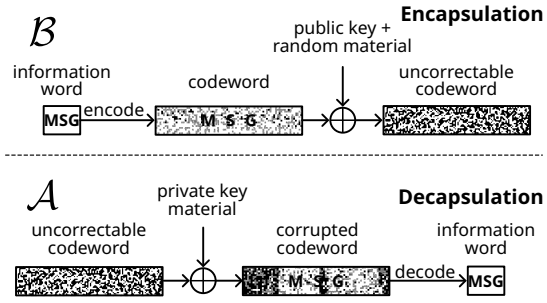


Fig. 1: overview of HQC's working principle.

2 Background

HQC is a code-based candidate employing two different linear codes: a quasi-cyclic random code that ensures the scheme's security, and a fixed public code with high error correction capacity and an efficient decoding algorithm for its designed purpose, i.e., providing the plaintext with the high error tolerance required by the scheme.

The working principle by which HQC allows to communicate a session key between two agents $\mathcal{A}$ and $\mathcal{B}$, depicted in Fig. 1, starts with $\mathcal{A}$ generating a key pair and sharing the public key with $\mathcal{B}$. Having the public key of $\mathcal{A}$, $\mathcal{B}$ starts by encoding a fully random plaintext, from which the session key will be derived, with the public, highly efficient error correcting code. Afterwards, he corrupts the resulting codeword way beyond the error-correcting capacity of the public code by employing both the public key and some randomness generated deterministically from the message, obtaining the ciphertext, i.e, the encapsulated key. The ciphertext is sent to $\mathcal{A}$, while also deriving the private shared session key from the message. Only $\mathcal{A}$, which is in possession of the private key corresponding to the public key employed by $\mathcal{B}$ is able to remove a large amount of the introduced corruptions of the codeword, reducing the corruption to a point where the public error correction code allows to correct the remaining errors. HQC employs the transformation described by Hofheinz-Hövelmanns-Kiltz (HHK) in [22] to achieve resistance against active attackers. Informally, this requires that $\mathcal{A}$ should be able to validate if the recovered message is actually the one sent by $\mathcal{B}$: this is done in practice through a re-encryption of the message performed by $\mathcal{A}$ after the decryption, and a comparison with the received ciphertext.

In the following, we detail the structure of the HQC.KEM, i.e., the cryptographic primitive obtained applying the HHK transformation to the HQC Public Key Encryption (HQC.PPKE).

2.1 HQC arithmetic

Let $\mathbb{F}_2$ be the binary finite field, and $\mathbf{R}$ be the polynomial ring $\mathbb{F}_2[x]/\langle x^p-1\rangle$. To thwart mathematical attacks based on the ring structure, HQC picks p as a prime number such that $p > 3$ and $\mathrm{ord}_2(p) = p-1$, determining that $x^p - 1 \in \mathbb{F}_2[x]$ would only admit two irreducible factors ($\mathrm{mod}\ p$).

We say that a polynomial $a \in \mathbf{R}$ has a Hamming weight $\omega(a)$ if it has $\omega(a)$ non-zero binary coefficients. The set $\mathbf{R}_w \subset \mathbf{R}$ contains all polynomials in $\mathbf{R}$ having Hamming weight equal to w, $w \geq 1$. Operatively, each polynomial $a = a_0 + a_1 x + \ldots + a_{p-1}x^{p-1} \in \mathbf{R}$ can also be considered as a p-dimensional binary vector composed by its coefficients $\mathbf{a} = [a_0, a_1, \ldots, a_{p-1}] \in \mathbb{F}_2^p$. Addition and subtraction among two polynomials $a, b \in \mathbf{R}$ corresponds to a coefficient-wise Boolean `xor` between their vector representations (i.e., $a_i \oplus b_i, 0 \leq i < p$). Their product $c = a \cdot b$ instead is obtained by the formula $c_i = \bigoplus_{j+k \equiv i \bmod p}(a_j \texttt{ and } b_k)$, with $i, j, k \in \{0, 1, \ldots, p-1\}$ due to the cyclic structure of the polynomial ring $\mathbf{R}$.

A $[n, k, d]$ binary linear forward error-correcting code $\mathcal{C}$ is a subspace of $\mathbb{F}_2^n$ having dimension k, and minimum distance d among any two (row) vectors in $\mathcal{C}$. The generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$ of the code $\mathcal{C}$ allows to generate all the 2^k codewords $\mathbf{c} \in \mathcal{C} \subset \mathbb{F}_2^n$ from the information words $\mathbf{m} \in \mathbb{F}_2^k$: $\mathcal{C} = \{\mathbf{mG}, \mathbf{m} \in \mathbb{F}_2^k\}$. Conversely, $\mathbf{H} \in \mathbb{F}_2^{(n-k)\times n}$ is the parity-check matrix of the code $\mathcal{C}$ such that $\forall \mathbf{c} \in \mathcal{C}$, $\mathbf{Hc}^T = \mathbf{0}$. The result of $\mathbf{Hv}^T$ computed from a generic $\mathbf{v} \in \mathbb{F}_2^n$ and $\mathbf{H}$ is called syndrome $\mathbf{s} \in \mathbb{F}_2^{n-k}$.

HQC uses a random quasi-cyclic $[2p, p, d]$ code with a public parity-check matrix $\mathbf{H} = [\mathbf{I}_p \mid \mathbf{rot}(h)]$, where $\mathbf{I}_p$ is the $p \times p$ identity matrix, while $\mathbf{rot}(h)$ is the $p \times p$ matrix obtained from the juxtaposition of h and a sequence of columns, each of which is obtained through a vertical rotation of the previous one by one coefficient. The public efficiently decodable code is generated by the concatenation of a shortened Reed-Solomon (RS) $[n_e, k_e, d_e]$ (external) code with a duplicated Reed-Muller (RM) $[n_i, k_i, d_i]$ (internal) code, producing a concatenated code with length, dimension and minimum distance of $[n_e n_i, k_e k_i, d_e d_i]$. We will refer to the RS-RM generator matrix of this code as $\mathbf{G}$.

We denote with $a \xleftarrow{\$} \mathbf{S}$ the procedure of picking a random element a with a uniform distribution among the elements of a set $\mathbf{S}$. When the procedure is made deterministic by using a Cryptographically Secure Pseudo-Random Number Generator (CSPRNG) and an input seed $\eta \in \{0,1\}^l, l \geq 1$, we refer to it with $a \leftarrow \mathrm{CSPRNG}(\eta, \mathbf{S})$. In particular, HQC employs the SHAKE256 algorithm from the SHA-3 NIST standard [1] to implement the CSPRNG.

Moreover, the KEM construction of HQC makes use of two different hash functions, HASH-G and HASH-K, that for efficiency reasons are both based on SHAKE256 and differentiated by a single byte absorbed as the last element, providing a domain separation to securely employ the same cryptographic primitive for different purposes.

2.2 HQC Public Key Encryption

The key generation procedure of HQC.PPKE (Alg. 1) starts by sampling two random seeds γ and φ of 320 bits each. φ is used to sample the random public polynomial h from $\mathbf{R}$, that in turn generates the parity-check matrix of a random QC code $\mathbf{H} = [\mathbf{I}_p, \mathbf{rot}(h)]$. The seed γ is used to sample the random secret polynomials $x, y \in \mathbf{R}$, both with fixed Hamming weight $\omega(x) = \omega(y) = w$. s is the syndrome polynomial of the pair (x, y) in the random quasi-cyclic code defined by $\mathbf{H}$, $s \leftarrow x + h \cdot y$, i.e., $\mathbf{s} = \mathbf{H}[\mathbf{x} \mid \mathbf{y}]^\top$. The outputs are the public key (φ, s) and the private key γ.

The HQC.PPKE encryption procedure (Alg. 2) receives as input the public key (φ, s), the random message $\mathbf{m}$ to be transmitted encoded with $k \in \{128, 192, 256\}$ bits, and a 512-bit public salt θ employed to randomize the ciphertext and achieve the IND-CPA property. Three random polynomials $e, r_a, r_b \in \mathbf{R}$, are randomly sampled using the seed θ such that $\omega(e) = w_e$ and $\omega(r_a) = \omega(r_b) = w_r$. The parity-check matrix $\mathbf{H}$ is expanded from the public seed φ and used to compute the syndrome polynomial u of the bit vector $[\mathbf{r}_a \mid \mathbf{r}_b]$ as $u \leftarrow r_a + h \cdot r_b \Leftrightarrow \mathbf{u} = \mathbf{H}[\mathbf{r_a} \mid \mathbf{r_b}]^\top$. The codeword resulting from the encoding of the message $\mathbf{m}$ with the public RS-RM code having generator matrix $\mathbf{G}$ is then corrupted adding the first $n_e n_i < p$ bits obtained from the $s \cdot r_b + e$ operation. The resulting vector $\mathbf{v} \in \mathbb{F}_2^{n_e n_i}$ cannot be used to obtain $\mathbf{m}$ using the public RS-RM decoding algorithm alone, since we added far more error bits than its maximum correcting capability. The ciphertext is $(u, \mathbf{v})$.

The HQC.PPKE decryption algorithm (Alg. 3) starts by expanding the secret vector $[\mathbf{x} \mid \mathbf{y}] \in \mathbb{F}_2^{2p}$ from the secret key $\mathsf{sk} = \gamma$ and subtracts from the polynomial v, derived from the second component $\mathbf{v}$ of the ciphertext, the product of the first one u by the second element of the secret vector y. The resulting quantity, $v - u \cdot y$ can be shown to be close to a codeword of the RS-RM public code $\mathbf{G}$ as $v - u \cdot y = \mathbf{mG} + (s \cdot r_b + e - h \cdot r_b \cdot y - r_a \cdot y)$, and, since $h \cdot y = s - x$ from the keygen algorithm, we have that $\mathbf{mG} + (s \cdot r_b + e - s \cdot r_b + x \cdot r_b - r_a \cdot y) = \mathbf{mG} + (e + x \cdot r_b - r_a \cdot y)$, all of which are polynomial with a small fixed Hamming weight. Subsequently, the errors added to the codeword $\mathbf{mG}$ are removed by applying the RS-RM decoding algorithm for the given generation matrix $\mathbf{G}$, and the original message is derived. The HQC.PPKE parameters (i.e., p, w, w_e, w_r, n_i, n_e, k_i, k_e, d_i, d_e, with $k = k_i k_e$) are tuned so that this decoding action has a negligible failure rate. In the call for post-quantum cryptographic schemes, NIST provided a security level classification to categorize each cipher. Specifically, security levels 1, 3, and 5 correspond to the lowest computational efforts needed to derive the secret key of `AES-128`, `AES-192`, `AES-256` via the best classical and quantum cryptanalytic attacks, respectively. The designers of HQC provided parameters for the cryptosystem, reported in Tab. 1, so that the decryption failures take place with a probability of $2^{-\lambda}$, where λ is the bit-length of the key of the AES cipher considered at the corresponding security level.

2.3 HQC Key Encapsulation Method

The HQC.KEM is obtained wrapping the HQC.PPKE with the HHK transformation [22] which, from a functional

Algorithm 1 HQC.PPKE-KEYGEN

Require: None
Ensure: $\mathsf{pk} = (\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$,
$\mathsf{sk} = \gamma \in \{0,1\}^{320}$
1: $(\gamma, \varphi) \xleftarrow{\$} \{0,1\}^{320} \times \{0,1\}^{320}$
2: $h \leftarrow \text{CSPRNG}(\varphi, \mathbf{R})$
3: $(y, x) \leftarrow \text{CSPRNG}(\gamma, \mathbf{R}_w \times \mathbf{R}_w)$
4: $s \leftarrow x + h \cdot y$
5: **return** $\mathsf{pk} = (\varphi, s)$, $\mathsf{sk} = \gamma$

Algorithm 2 HQC.PPKE-ENCRYPT

Require: $\mathsf{pk} = (\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$,
$\mathbf{m} \in \{0,1\}^k, \theta \in \{0,1\}^{512}$
Ensure: $\mathsf{ctx} = (u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i})$
1: $(r_b, e, r_a) \leftarrow \text{CSPRNG}(\theta, \mathbf{R}_{w_r} \times \mathbf{R}_{w_e} \times \mathbf{R}_{w_r})$
2: $h \leftarrow \text{CSPRNG}(\varphi, \mathbf{R})$
3: $u \leftarrow r_a + h \cdot r_b$
4: $\mathbf{v} \leftarrow \text{ENCODE}_{\mathbf{G}}(\mathbf{m}) + \text{TRUNC}\,(s \cdot r_b + e)$
5: **return** $\mathsf{ctx} = (u, \mathbf{v})$

Algorithm 3 HQC.PPKE-DECRYPT

Require: $\mathsf{sk} = \gamma \in \{0,1\}^{320}$, $\mathsf{ctx} = (u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i})$
Ensure: $\mathbf{m}' \in \{0,1\}^k$
1: $(y, x) \leftarrow \text{CSPRNG}(\gamma, \mathbf{R}_w \times \mathbf{R}_w)$
2: $\mathbf{m}' \leftarrow \text{DECODE}_G(\text{TRUNC}([\mathbf{v} \| 0^{p - n_e n_i}] - u \cdot y))$
3: **return** $\mathbf{m}'$

Algorithm 4 HQC.KEM-KEYGEN

Require: None
Ensure: $\mathsf{pk} = (\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$,
$\mathsf{sk} = (\gamma \in \{0,1\}^{320}, \sigma \in \{0,1\}^k, \varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
1: $\sigma \xleftarrow{\$} \{0,1\}^k$
2: $((\varphi, s), \gamma) \leftarrow$ HQC.PPKE-KEYGEN()
3: **return** $\mathsf{pk} = (\varphi, s)$, $\mathsf{sk} = (\gamma, \sigma, \varphi, s)$

Algorithm 5 HQC.KEM-ENCAPSULATE

Require: $\mathsf{pk}{=}(\varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
Ensure: $\mathsf{ctx}{=}(u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i}, \text{salt} \in \{0,1\}^{128})$, $K{\in}\{0,1\}^{512}$
1: $\mathbf{m} \xleftarrow{\$} \{0,1\}^k$, $\text{salt} \xleftarrow{\$} \{0,1\}^{128}$
2: $\theta \leftarrow \text{HASH-G}(\mathbf{m} \| \text{HEADBYTES}(\mathsf{sk}, 32) \| \text{salt})$
3: $(u, \mathbf{v}) \leftarrow$ HQC.PPKE-ENCRYPT$((\varphi, s), \mathbf{m}, \theta)$
4: $K \leftarrow \text{HASH-K}(\mathbf{m} \| u \| \mathbf{v})$
5: **return** $\mathsf{ctx} = (u, \mathbf{v}, \text{salt}), K$

Algorithm 6 HQC.KEM-DECAPSULATE

Require: $\mathsf{ctx} = (u \in \mathbf{R}, \mathbf{v} \in \mathbb{F}_2^{n_e n_i}, \text{salt} \in \{0,1\}^{128})$,
$\mathsf{sk} = (\gamma \in \{0,1\}^{320}, \sigma \in \{0,1\}^k, \varphi \in \{0,1\}^{320}, s \in \mathbf{R})$
Ensure: $K \in \{0,1\}^{512}$
1: $\mathbf{m}' \leftarrow$ HQC.PPKE-DECRYPT$(\gamma, (u, \mathbf{v}))$
2: $\theta' \leftarrow \text{HASH-G}(\mathbf{m}' \| \text{HEADBYTES}(\mathsf{sk}, 32) \| \text{salt})$
3: $\mathsf{ctx}' \leftarrow$ HQC.PPKE-ENCRYPT$((\varphi, s), \mathbf{m}', \theta')$
4: **if** $(\mathsf{ctx} \neq \mathsf{ctx}')$ $K' \leftarrow \text{HASH-K}(\sigma \| u \| \mathbf{v})$ **else** $K' \leftarrow \text{HASH-K}(\mathbf{m}' \| u \| \mathbf{v})$
5: **return** K'

TABLE 1: HQC parameter sets. Public fixed codes are specified via the $[n, k, d]$ notation. The rightmost column reports the Decoding Failure Rate (DFR) of each parameter set.

Sec. level	Param. set	Public concatenated code: Reed-Solomon	Public concatenated code: Reed-Muller	Polynomial ring $\mathbf{R}$: p	w	$w_e{=}w_r$	DFR
1	HQC-128	$[46, 16, 31]$	$[384, 8, 192]$	17669	66	75	2^{-128}
3	HQC-192	$[56, 24, 33]$	$[640, 8, 320]$	35851	100	114	2^{-192}
5	HQC-256	$[90, 32, 59]$	$[640, 8, 320]$	57637	131	149	2^{-256}

standpoint, feeds the HQC.PPKE with a random message, from which the secret to be employed as a session key K is derived, and adds to the ciphertext additional information which allows to check if a decryption error took place. In this case, to avoid information leakage, the construction emits a random string, deterministically derived with a CSPRNG from the ciphertext and a secret seed stored within the private key. As a consequence, the key generation of HQC.KEM matches the one of HQC.PPKE, save for the generation of the additional seed, denoted as σ, of 128, 192, or 256 bits (depending on the security level), stored together with the private seed γ.

The HQC.KEM encapsulation algorithm (Alg. 5) starts by picking a uniformly distributed random message $\mathbf{m}$ encoded with 128, 192, or 256 bits, depending on the security level, and a 128-bit public salt. The two quantities are concatenated with the public key (φ, s) and hashed via the HASH-G function to get a digest θ, which is subsequently employed as the ephemeral value required as an input by the HQC.PPKE encryption algorithm. After calling the routine HQC.PPKE-ENCRYPT, the shared secret session key K is derived through a different hash function, HASH-K, fed with the concatenation of the message $\mathbf{m}$ and the components $u, \mathbf{v}$ of the ciphertext.

The HQC.KEM decapsulation procedure (Alg. 6) starts by computing the output value $\mathbf{m}'$ of the routine HQC.PPKE-DECRYPT fed with the secret key seed and the received components $u, \mathbf{v}$ of the ciphertext ctx, which should match the original confidential message $\mathbf{m}$ unless a decryption failure occurs. To distinguish between the two possible scenarios and provide security against active attackers, which may have mangled the received ciphertext, the HHK transformation followed by the decapsulation procedure mandates to compare the received ctx with the output obtained from the re-computation of the HQC.PPKE-ENCRYPT routine, which in turn requires the re-computation of the ephemeral value θ fed to it as last input parameter. If the retrieved secret message $\mathbf{m}'$ matches the one that was actually encrypted, the outcome of such process (lines 2 and 3 of Alg. 6) will yield a value ctx′ matching the received ciphertext ctx (check performed at line 4). In case the re-computed ciphertext ctx′ does not match the received one, the *implicit* rejection mechanism requires the computation of the session key K as the result of the HASH-K function fed with the concatenation of the received ctx and the binary string σ (included in the secret key), instead of the mangled message $\mathbf{m}'$, which may provide information to an attacker.

3 HQC.KEM UNIFIED DESIGN

In this Section we describe the RTL HW design of our unified HQC module, which is composed of independent specialized units, such as the arithmetic functional units (Subsection 3.1), the RS-RM encoder and decoder (Subsection 3.2), and the CSPRNG and Hash function unit (Subsection 3.3), that are orchestrated by a centralized Finite State Machine (FSM) implementing a schedule of the operations for the KEYGEN, ENCAPSULATE, and DECAPSULATE primitives (Subsection 3.4). Our unified design is a versatile component that allows to support the KEM operations at both endpoints, $\mathcal{A}$ and $\mathcal{B}$, of the communication as described in Section 2. In scenarios where the mutual authentication is not necessary, only a subset of the functionalities of the unified HQC KEM accelerator are meant be employed. Specifically, the HQC.KEM-KEYGEN

and HQC.KEM-DECAPSULATION algorithms are meant to be executed at served-side, whereas the HQC.KEM-ENCAPSULATION is meant to be executed at client-side.

A careful analysis of the operations prescribed by each of the algorithms reported in the official specification of the HQC cryptoscheme allows to infer the following grouping: KEYGEN $\subset$ ENCAPSULATION $\subset$ DECAPSULATION. Indeed, as reported in the previous section the operations of the HQC.PPKE-KEYGEN (lines 2-4 of Alg. 1) are included in the HQC.PPKE-ENCRYPTION (lines 1-3 of Alg. 2), while the operations in the HQC.KEM-ENCAPSULATION (line 2-4 of Alg. 5) are included in the HQC.KEM-DECAPSULATION (line 2-4 of Alg. 6).

The reported observation allows to infer that the realization of a unified design (in terms of algorithms and support for all parameter sets) is most effective in a mutual authentication scenario. When the authentication is performed by a single party (e.g., the server), a fully unified accelerator is effective to be used only by the said party, whereas the counterpart may benefit from an engine implementing only the HQC.KEM-ENCAPSULATION algorithm, with support for all parameter sets (as such a selection is performed by its counterpart, when running the key generation algorithm).

The modules within the design communicate using one or more shared memories having B=128-bit wide word size and 64-bit wide stream interfaces. Interconnections between modules are managed via simple circuit-switched networks managed by the schedule FSM to solve contention issues.

Polynomials in $\mathbf{R} = \mathbb{F}_2[x]/(x^p - 1)$ are represented as a list of the p binary coefficients in increasing degree order. Differently, polynomials in $\mathbf{R}$ with a fixed weight $w \ll p$ are stored as a list of exponents of the non-null monomial term x^i, $0 \leq i \leq p-1$, each encoded as a 16-bit unsigned integer. We refer to the first type of element as *dense polynomial*, and store them in $\lceil \frac{p}{B} \rceil$ memory words, whereas the second type of elements, here referred as *sparse polynomial*, are efficiently stored in $\lceil \frac{16*w}{B} \rceil < \lceil \frac{p}{B} \rceil$ memory words.

In the following we analyze each component realizing a crucial operation prescribed by the HQC specification. Each subsection reports a baseline description of the design appeared in [21], and subsequently the design enhancements needed for an accelerator supporting all HQC parameter sets, and, consequently, all security levels.

3.1 Ring Arithmetic Units

HQC.KEM uses only additions and multiplications between polynomials in $\mathbf{R}$, as the single required subtraction is also realized as an addition due to the arithmetic in $\mathbb{F}_2[x]$.

3.1.1 Polynomial Adder

We add/subtract two dense polynomials `xor`-ing them coefficient-wise. As the operands are stored in memory as sequences of B-bit wide words, $\lceil \frac{p}{B} \rceil$ word-wise `xor`s are carried out. Moreover, HQC.PPKE-ENCRYPT requires two additions between a sparse and a dense polynomial, and HQC.PPKE-KEYGEN also performs another one. We realize this operation by flipping the bits of the dense operand in the positions indicated by the sparse one. Recalling that the indexes of the sparse polynomial are stored as 16-bits unsigned integers, we split it to determine the dense operand word address as the highest $16 - \log_2(B)$ bits of the index encoding, while the remaining bits of the index are encoding the bit position within the word to flip. Since two consecutive indexes may flip bits in the same memory word, creating a *read-after-write* data dependency, we employed a straightforward sequential architecture that, even considering the non-negligible memory access latencies, does not penalize the overall performance thanks to the low weight of the sparse polynomials (see Tab. 1).

Differently from [21], the support for multiple parameter sets (i.e., different p, w values), impacts the management of dense or sparse polynomials with different maximum degrees p or different weights w, respectively. In particular, it mandates to make available the values for the number of memory words (i.e., $\lceil \frac{p}{B} \rceil$) and for the weight of sparse polynomials w in read-only memories (ROM) and use them as input to registers employed as counters.

3.1.2 Polynomial Multiplier

The HQC.KEM primitives require only multiplications where one operand is a sparse polynomial and the other is dense one. While sub-quadratic approaches for generic polynomial multiplication exist, we note that, in the HQC case where the amount of non-null coefficients of one of the factors is $w \approx \mathcal{O}(\sqrt{p})$ out of p, the schoolbook approach has an asymptotic complexity of $\mathcal{O}(p^{1.5})$, which is already better than the one of the Karatsuba approach. Furthermore, the schoolbook method does not hide large constants within the asymptotic notation, and allows for a resource-sparing implementation. Therefore, we adopt a *shift-and-add* approach for our multiplier unit, where the dense operand is shifted by an amount of bits specified by each index of the sparse operand, and accumulated into the result. We reduce the number accumulator memory words to the minimum possible by immediately reducing modulo $x^p - 1$ the shifted polynomial to be added. Due to the structure of the polynomial ring $\mathbf{R} = \mathbb{F}_2[x]/(x^p - 1)$, this is possible through a simple change in the location where the coefficients of the dense polynomial are added, as the polynomial modular reduction modulo $x^p - 1$ amounts to a bit-wise `xor` of the coefficients of degrees greater or equal than p onto the coefficients of the monomials of the result having a degree lower by exactly p units. Our word-wise shift-and-add approach rotates and accumulates a B-bit word at a time, in turn taking $w \cdot \lceil \frac{p}{B} \rceil$ clock cycles and using $\lceil \frac{p}{B} \rceil$ memory blocks to store the result. We note that this algorithm runs in constant time as the memories of our RTL design do not feature caches, thus fully eliminating the timing side channel which would be present in an analogue software implementation. Furthermore, our design improves by $l\times$ the latency of the dense by sparse polynomial multiplication processing l sparse indexes in parallel. To this end, we employ l `read` ports accessing the dense binary polynomial operand, a `read` and `write` port for the accumulated result, and a `read` port for the sparse polynomial. Due to the potentially prominent size of the memory blocks B, we employ a pipelined Barrel module to perform the shift operation, and we parametrized it in the number of pipeline stages to break the possibly long critical path and improve performance. To accomplish the support of all parameter set, the number of indexes i of the

sparse polynomial operand to process is received as input to the module upon the start of operation, masking out the accumulation of invalid indexes in case l does not divide i.

Differently from [21], the support for multiple parameter sets (i.e., different p, w values), mandates to make available the values for the number of memory words (i.e., $\lceil \frac{p}{B} \rceil$) of the dense polynomial operand and for the weight w of the sparse polynomial operand in read-only memories (ROM) and use them as input to registers employed as counters.

3.2 Reed-Solomon/Reed-Muller Encoder and Decoder

HQC uses a public concatenated code composed by a shortened Reed-Solomon (RS) as the external code, and a duplicated Reed-Muller (RM) as the internal one. The RS-RM encoding procedure expands each byte in input, taken as a message symbol of the Reed-Solomon code, into a 384-bit or 640-bit codeword of the Reed-Muller code. While the RM and RS encoders can work in a pipelined fashion processing the original input message byte-wise, this is not possible for the concatenated RS-RM decoder, since the RS decoder requires all the codeword symbols.

3.2.1 Reed-Solomon Encoder and Decoder

The HQC official specification reports an instantiation of a $[n_e, k_e, d_e]$ RS code for each NIST security level, all able to correct $t = \frac{d_e-1}{2}$ erroneous codeword symbols: `RS-1`$=[255, 225, 31]$, `RS-2`$=[255, 223, 33]$, and `RS-3`$=[255, 197, 59]$. A shortened variant of these codes are actually used in the HQC.KEM primitives and are obtained by fixing 209, 199, or 165 symbols of the message, respectively, avoiding their transmission in the codeword: `RS-S1`$=[46, 16, 31]$, `RS-S2`$=[56, 24, 33]$, and `RS-S3`$=[90, 32, 59]$. Each code treats each 8-bits block of data as an element of $\mathbb{F}_{2^8}$, called symbol, represented employing the irreducible polynomial $y^8 + y^4 + y^3 + y^2 + 1 \in \mathbb{F}_2[y]$. In particular, the message to be encoded is used to build a *message polynomial* $u(x) \in \mathbb{F}_{2^8}[x]$, with degree $k_e - 1$. An n_e-symbol codeword $c(x) = u(x)g(x) \in \mathbb{F}_{2^8}[x]$ is obtained multiplying the input message polynomial $u(x) \in \mathbb{F}_{2^8}[x]$ by the code *generator polynomial* $g(x) \in \mathbb{F}_{2^8}[x]$, which has degree $d_e = n_e - k_e + 1$.

The RS code generator polynomial is defined [1] as $g(x) = (x - \alpha) \cdot (x - \alpha^2) \cdot (x - \alpha^3) \cdots (x - \alpha^{d_e-1})$, where α is the primitive element in $\mathbb{F}_{2^8}$ having representation $\alpha = y$. As a consequence, the first d_e powers of α are roots of both $g(x)$ and any error-free codeword $c(x)$. A systematic encoding procedure requires that the resulting codeword contains the sequence of symbols of the message polynomial $u(x)$ as a prefix and the error correcting symbols as a suffix, while still being a multiple of $g(x)$. Such encoding is obtained computing $c(x) = x^{n_e-k_e}u(x) - \left(x^{n_e-k_e}u(x) \bmod g(x)\right)$.

Reed-Solomon Encoder. Fig. 2 shows the LFSR based design to compute $\left(x^{n_e-k_e}u(x) \bmod g(x)\right)$ in exactly k_e clock cycles [23]. The RS codeword in systematic form is obtained as the sequence of the first k_e symbols produced in output, corresponding to the original message, followed by the symbols read out from the LFSR registers after the last input symbol is consumed.

1. In the HQC specification (4 Oct. 2024), the constant term of the gen. polynomial for the RS code in `HQC-128` is 9 instead of 89

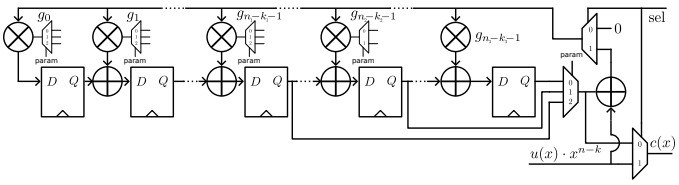


Fig. 2: Circuit computing $u(x) \cdot x^{n_e-k_e} \bmod g(x)$

Differently from [21], to support multiple parameter sets in the unified design, we instantiated a single LFSR able to fit the coefficients of the generator polynomial with largest degree. During the computation, the coefficients of the correct generator polynomial are selected from a ROM (upper muxes in Fig.2) and used as one operand of the $\mathbb{F}_{2^8}$ multipliers, while the symbol to be sent along the feedback and output path is chosen via a small multiplexer (the rightmost one taking as many input as the number of possible generator polynomials, i.e., 3). We note that, the $(n_e - k_e)$ $\mathbb{F}_{2^8}$ multipliers in Fig.2 need to be coupled with a mux taking as input three or two values in $\mathbb{F}_{2^8}$ or directly connected with a value in $\mathbb{F}_{2^8}$, depending on the generator polynomial (for `HQC-128`,`-192`,`-256`, respectively). Only multipliers that takes a direct operand, i.e., the ones not conveyed through a mux, have been specialized for the fixed operand. The said specialization is also applied to every $\mathbb{F}_{2^8}$ multiplier in the non-unified designs in [21]. Overall, the unified RS encoder allows us to save 10% of area w.r.t. the three encoders in [21], in which the halving of the number of used flip-flops is particularly notable for an ASIC target.

Reed-Solomon Decoder. The typical algebraic RS decoder takes as input an error-affected codeword and considers it as a polynomial $r(x) \in \mathbb{F}_{2^8}[x]$, with degree n_e, obtained from the addition of an error-free codeword $c(x) \in \mathbb{F}_{2^8}[x]$, with degree n_e, to an unknown error polynomial $e(x) \in \mathbb{F}_{2^8}[x]$ having $\nu \leq t$ terms: $e(x) = \sum_{k=1}^{\nu} e_{i_k} x^{i_k}$, where i_k identifies the error coefficient with non-zero value. A decoder algorithm performs the following steps: **i)** computes the syndrome of the received codeword $r(x)$; **ii)** derives the number and the positions i_k of errors; **iii)** computes the values of the coefficients of $e(x)$, and then retrieves the error-free codeword $c(x) = r(x) - e(x)$.

Step i) The coefficients of the *syndrome polynomial* $S(x) = S_1 + S_2 x + \ldots + S_{2t} x^{2t-1} \in \mathbb{F}_{2^8}[x]$, with $2t = d_e - 1 = n_e - k_e$, are derived by evaluating the received codeword $r(x)$ at each root α^j of the generator polynomial $g(x)$, i.e., $S_j = r(\alpha^j) = c(\alpha^j) + e(\alpha^j) = e(\alpha^j)$. Horner's method for polynomial evaluation [24] can be used to perform the evaluation of α^j in the n_e-degree $r(x)$ polynomial using n_e multiplications and n_e additions. If all $2t$ syndrome values are zero, then $c(x) = r(x)$ and no errors have occurred. Otherwise $S_j = e(\alpha^j) = \sum_{i=0}^{n_e-1} e_i(\alpha^j)^i = \sum_{k=1}^{\nu} e_{i_k}(\alpha^j)^{i_k} = \sum_{k=1}^{\nu} Y_k X_k^j$ for each $j \in \{1, \ldots, 2t\}$, where $X_k = \alpha^{i_k}$ indicates the presence of an error in the i_k-th received symbol, and $Y_k = e_{i_k}$ is the error symbol value. A decoding algorithm attempts to solve the following set of simultaneous equations, which does not have a solution:

$$\begin{cases} Y_1 X_1 + Y_2 X_2 + \ldots + Y_\nu X_\nu = S_1 \\ Y_1 X_1^2 + Y_2 X_2^2 + \ldots + Y_\nu X_\nu^2 = S_2 \\ \vdots \\ Y_1 X_1^{2t} + Y_2 X_2^{2t} + \ldots + Y_\nu X_\nu^{2t} = S_{2t} \end{cases} \quad (1)$$

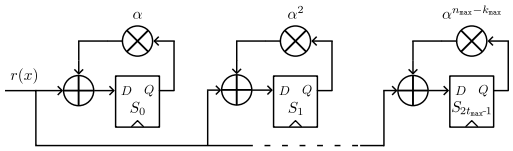


Fig. 3: Syndrome polynomial computation

A strategy to obtain the information captured by Eq. 1 considers the methods for solving the following equations, based on the definitions of *error locator polynomial* $\Lambda(x)$ and *error evaluator polynomial* $\Omega(x)$:

$$\Lambda(x) \triangleq \prod_{k=1}^{\nu}(1 - X_k x) = 1 + \Lambda_1 x + \ldots + \Lambda_\nu x^\nu \tag{2}$$

$$\begin{aligned}\Omega(x) &\triangleq \sum_{k=1}^{\nu} Y_k X_k \prod_{j=1, j\neq k}^{\nu}(1 - X_j x) = \Lambda(x)S(x) \bmod x^{2t} \\ &= \Omega_0 + \Omega_1 x + \ldots + \Omega_{\nu-1} x^{\nu-1}\end{aligned} \tag{3}$$

Step ii) Considering Eq. 2, $X_k^{-1}{=}\alpha^{-i_k}$ is a root of $\Lambda(x)$ that determines an unique error location i_k. Given that $\Lambda(X_k^{-1}) = 0$ for $1 \leq k \leq \nu$, multiplying both sides of Eq. 2 by $Y_k X_k^{j+\nu}$, we have that $\sum_{k=1}^{\nu}(Y_k X_k^{j+\nu} + \Lambda_1 Y_k X_k^{j+\nu-1} + \ldots + \Lambda_\nu Y_k X_k^j) = 0$. Knowing that $S_j = \sum_{k=1}^{\nu} Y_k X_k^j$, the formula simplifies in

$$S_j\Lambda_\nu + S_{j+1}\Lambda_{\nu-1} + \ldots + S_{j+\nu-1}\Lambda_1 = -S_{j+\nu} \tag{4}$$

Having up to $2t$ syndromes, we can construct a system of t linear equations having the coefficients of Λ as unknowns:

$$\begin{bmatrix} S_1 & S_2 & \ldots & S_\nu \\ S_2 & S_3 & \ldots & S_{\nu+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_\nu & S_{\nu+1} & \ldots & S_{2\nu-1} \end{bmatrix} \begin{bmatrix} \Lambda_\nu \\ \Lambda_{\nu-1} \\ \vdots \\ \Lambda_1 \end{bmatrix} = \begin{bmatrix} -S_{\nu+1} \\ -S_{\nu+2} \\ \vdots \\ -S_{2\nu} \end{bmatrix} \tag{5}$$

Being the number of errors $\nu{\leq}t$ unknown, the Peterson-Gorenstein-Zierler (PGZ) algorithm tries all $1 \leq\nu\leq t$ in decreasing order until the matrix has nonzero determinant, and computes its inverse to determine $\Lambda(x)$. A more efficient algorithm is given by the Berlekamp-Massey algorithm [25]. The Eq. 4 describes the output of an LFSR with weights on the feedback path given by $\Lambda_1, \ldots, \Lambda_\nu$. Determining the coefficients of $\Lambda(x)$ is equivalent to synthesizing the minimum length LFSR generating the entire sequence of syndromes $S_1, S_2, \ldots, S_{2t}$ when the LFSR registers are initialized with $S_1, S_2, \ldots, S_\nu$. Once $\Lambda(x)$ is computed, the Chien Search quickly determines its roots. Given a generic evaluation $\Lambda(\alpha^{-i}) = \sum_{k=1}^{t} \Lambda_k \alpha^{-ik}$, the evaluation with the following alpha power is $\Lambda(\alpha^{-i+1}) = \sum_{k=1}^{t} \Lambda_k \alpha^{-ik+k} = \sum_{k=1}^{t}(\Lambda_k \alpha^{-ik})\alpha^k$, which only requires to multiply the previous result by α^k.

Step iii) Once the error locations are known, the error values Y_k can be obtained possibly solving Eq. 1 via Gaussian elimination, or using Forney's formula, which exploits the fact that the matrix of X_k^j is a Vandermonde matrix. Consider now that the formal derivative $\Lambda'(x)$ is $\frac{d}{dx}\prod_{i=1}^{\nu}(1 - X_i x) = -\sum_{l=1}^{\nu} X_l \prod_{i=1, i\neq l}^{\nu}(1 - X_i x)$ due to Leibniz rule. Evaluated in X_k^{-1}, it yields $-X_k \prod_{j=1, j\neq k}^{\nu}(1 - X_j X_k^{-1}) = \frac{-1}{Y_k}\Omega(X_k^{-1})$ which is used to retrieve $Y_k = -\frac{\Omega(X_k^{-1})}{\Lambda'(X_k^{-1})}$. After the computation of the error polynomial, the error-free codeword can be reconstructed as $c(x) = r(x) - e(x)$.

HW design. We used Horner's method to determine the syndrome in step i). The corresponding evaluation circuit, shown in Fig. 3, processes a symbol of the input polynomial $r(x)$ at each clock cycle, from r_{n_e-1} to r_0, evaluating all the $2t$ alpha powers in parallel. After processing all n_e symbols of $r(x)$, the j-th memory element, with $1 \leq j \leq 2t$, will store the j-th syndrome value, i.e., $r_0 + r_1\alpha^j + \ldots + r_{n_e-1}(\alpha^j)^{(n_e-1)} = r(\alpha^j) = e(\alpha^j) = S_j$. By construction, the generator polynomials $g(x)$ of the three codes share the same roots, therefore the $\mathbb{F}_{2^8}$ multipliers in the circuit, that use a power of α as one operand, have a fixed input for all parameter sets. Starting from the fully combinatorial network of the Mastrovito bit-parallel design of binary finite fields multiplier, which includes both `and` and `xor` gates, we specialized each multiplier instance for each distinct value of an α power, with a net advantage both in terms of critical path length and number of logic gates. The number of $\mathbb{F}_{2^8}$ multiplier instances is given by the largest number of roots of the RS code generator polynomials defined by the parameter sets.

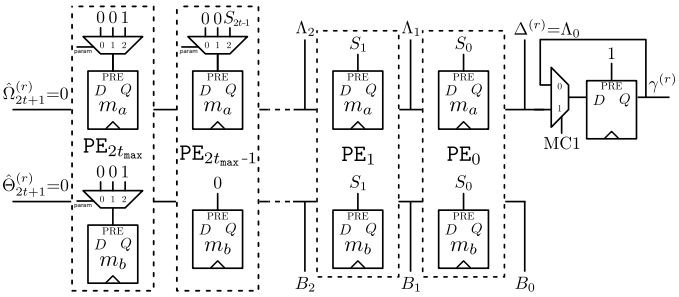


Fig. 4: ePIBMA architecture.

We employed the Enhanced Parallel Inversionless Berlekamp-Massey Algorithm (ePIBMA) to derive $\Lambda(x)$, and the Enhanced Chien Search and Error Evaluation (eCSEE) to find both the error locations and values. The two architectures, introduced in [26], are depicted in Fig. 4 and Fig. 5, and perform in parallel both steps ii) and iii). To support all parameter sets in the ePIBMA design, differently from [21], $2t_{\mathtt{max}}{+}1$ Processing Elements (PE) are instantiated (where $t_{\mathtt{max}}$ denotes the largest t among the values mandated by each parameter set), taking care to properly preset their registers m_a and m_b with the correct amount of coefficients from the previously computed syndrome polynomial $S(x)$, or with zero symbols if the PEs are unused (see leftmost muxes in Fig. 4). The computation is completed after $2t{-}1$ clock cycles. The eCSEE reuses the evaluation of X_k on the even powers of the unknown of $\Lambda(x)$ for both the Chien Search and the Error Evaluation, as $\Lambda'(x) = \frac{d}{dx}(\Lambda_0 + \Lambda_1 x + \ldots + \Lambda_\nu x^\nu) = \Lambda_1 + 2\Lambda_2 x + 3\Lambda_3 x^2 + \ldots + \nu\Lambda_\nu x^{\nu-1} = \Lambda_1 + \Lambda_3 x^2 + \ldots = x^{-1}\Lambda_{\text{odd}}(x)$. Due to the small size of the finite field used by the RS code, the finite field inversion necessary to compute Y_k is obtained via a 2 KiB ROM. The same design of eCSEE employed for `HQC-256` in [21] is sufficient to support all the parameter sets of HQC because the zero symbols introduced in the ePIBMA PEs ensured the correct computation of the error vector in n_e clock cycles. To retain a constant-time behavior, the decoding procedure is always executed, even if there are no errors detected during the syndrome computation.

Thanks to the alpha powers that the generator polynomials of the three RS codes have in common, we reused most of the resources to compute the syndrome and to generate the

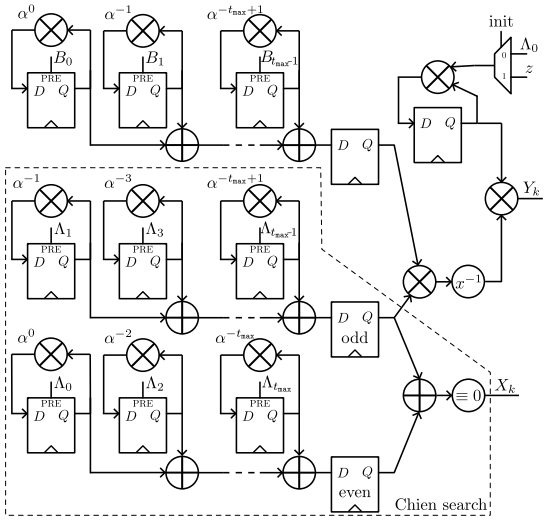


Fig. 5: eCSEE architecture.

error locator and evaluator polynomials. This optimization significantly improved the area usage of the unified decoder by $2\times$ compared to a naive strategy of replicating three separated decoders specialized for a specific RS code.

3.2.2 Reed-Muller Encoder and Decoder

The HQC specification describes a binary 1-st order Reed-Muller (RM) code $[n_i{=}2^l,\ k_i{=}l{+}1,\ d_i{=}2^{l-1}] = [128, 8, 64]$, $l{=}7$, extended with a repetition code with codeword composed by m replicas of the same RM original codeword. For the NIST security level 1, the extended RM code $[384, 8, 192]$ is obtained with $m{=}3$; for the security levels 3 and 5, the extended RM code $[640, 8, 320]$ is obtained with $m{=}5$.

Reed-Muller Encoder. To derive the codewords corresponding to each 8-bit input message, our design considers the RM generator matrix specified in HQC and follows the traditional vector-matrix multiplication strategy.

A straightforward realization would employ $k_i{=}8$ multiplexers with 128 input bits, and $k_i{-}1{=}7$ `xor` gates 128-bits wide. A simple, yet effective optimization, we applied in our design of the RM encoder considers each row in the (fixed) generation matrix as a sequence of 32-bits words and observes the presence during the multiplication computations yielding the same intermediate values. As a consequence, we realize the vector-matrix multiplication with only 9 multiplexers, with 32 input bits, and 9, 32-bits, banks of `xor` gates. The effectiveness of our optimization was recognized by the HQC team, which included it in the latest cipher specification [7]. Adapting the component to act with a different parameter set only requires to consider the appropriate multiplicity m replicating the RM codeword in $m{-}1$ memory locations.

Reed-Muller Decoder. Since the RS decoder acting as the second stage of the RS-RM concatenated decoder needs the entire output of the RM decoder, minimizing the latency of the RM decoder plays a significant role in our design. Therefore, in designing the RM decoder we prioritized performance over resource reuse. The extended RM codeword is initially de-duplicated accumulating in a flip-flop-based buffer the bitwise sum of the m-fold 128-bit replicas, via 128 independent adders, to the end of dealing with the plain 1st order RM code $[n_i{=}2^l,\ k_i{=}l{+}1,\ d_i{=}2^{l-1}]{=}[128, 8, 64]$. Our design of the RM decoder follows the implementation of the Maximum Likelihood (ML) decoder computing a fast Hadamard transform [27], which requires only $\mathcal{O}(n_i \log(n_i))$ binary operations and $\log(n_i)$ clock cycles in contrast with the $\mathcal{O}(n_i^2)$ binary operations and $\mathcal{O}(n_i)$ clock cycles provided by a plain ML-based RM decoder. We find the maximum absolute value in the Hadamard transform result with a pipelined comparator tree computing pairwise maxima, acting on a tunable-sized input vector.

To support all parameter sets, also for this component it is sufficient to employ the appropriate multiplicity value m during the de-duplication step previously described.

3.3 CSPRNG and Hash Functions

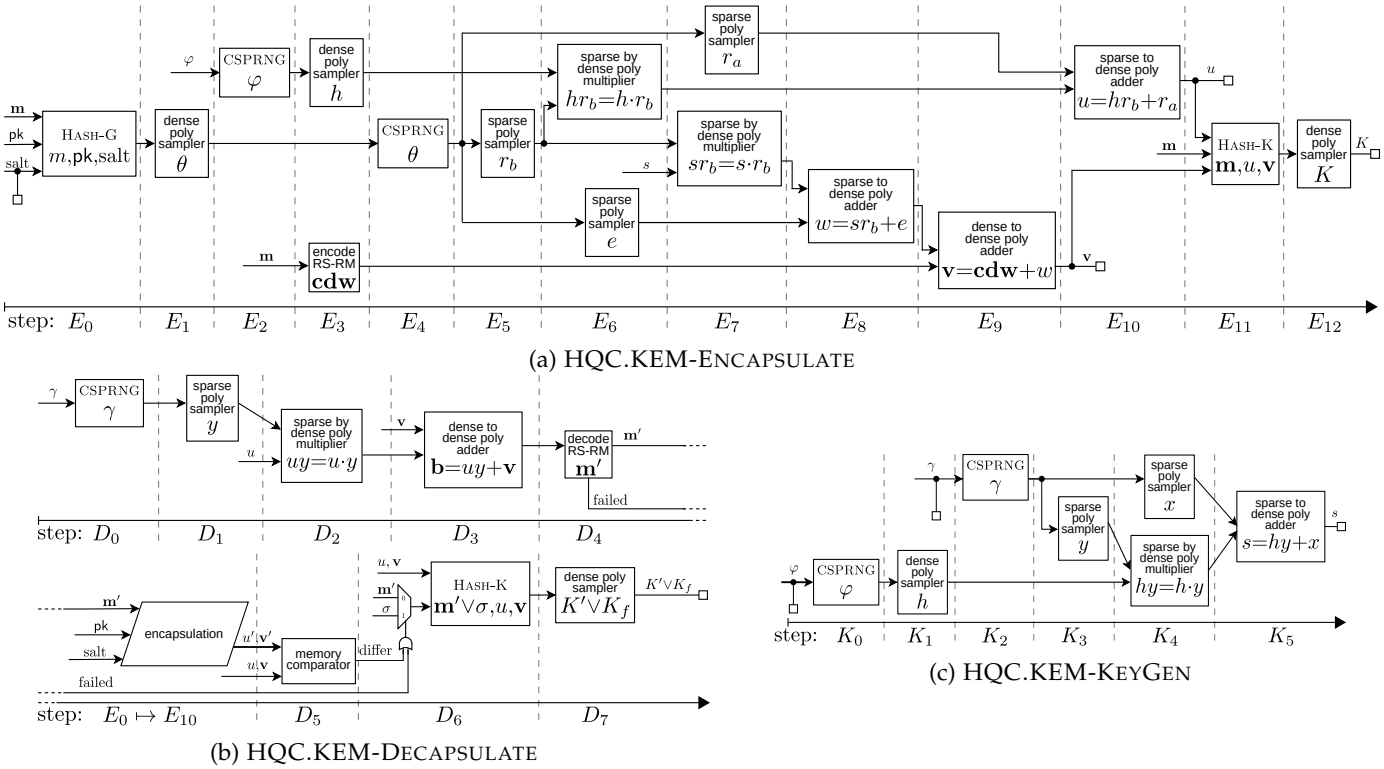
In the official specification of HQC.KEM, the Cryptographically Secure Pseudo Random Number Generators (CSPRNGs) and the cryptographic hash functions (HASH-G and HASH-K) that appear in the definition of KEYGEN, ENCAPSULATE and DECAPSULATE algorithms must be based on the SHAKE256 algorithm compliant with in the SHA-3 NIST standard specification [1]. In our design we decided to not include a True Random Number Generator to sample the seeds γ, ϕ, σ, m, and salt as the security and quality of this component is a completely orthogonal challenge deeply depending on the target technology realizing the design. We therefore have those elements already present in memory as inputs of the appropriate KEM operation.

In our design the realization of the component implementing the CSPRNG(`seed`, $\mathbf{S}$) function expands the `seed` taken as input employing a high-performance SHAKE256 module [28] to obtain a stream of pseudorandom bits. Subsequently, depending on the specification of the set of objects $\mathbf{S}$, which can be dense or sparse polynomials, we use suitable amount of bits from the said stream to derive an item in $\mathbf{S}$. The two hash functions HASH-G and HASH-K are realized with the same SHAKE256 module, but with a fixed 512-bit output. To provide domain separation among the CSPRNG function, and the HASH-G and HASH-K function, a one-byte constant equal to `0x02`, `0x03`, `0x04`, respectively, is appended to the bitstrings fed to SHAKE.

3.3.1 Dense and Sparse Polynomial Samplers

Sampling polynomials uniformly from $\mathbf{R}$, requires to truncate the stream of pseudorandom bits computed as a output by the SHAKE module to obtain a bitstring of length p. In addition, HQC requires to sample two or more $(p{-}1)$-degree polynomials from $\mathbf{R}$, each with a fixed number of non-null coefficients (i.e., CSPRNG($\{0,1\}^{320}$, $\mathbf{R}_w \times \mathbf{R}_w$), and CSPRNG($\{0,1\}^{512}$, $\mathbf{R}_{w_r} \times \mathbf{R}_{w_e} \times \mathbf{R}_{w_r}$)). To this end, the constant-time algorithm appeared in [29], has been included in the official specification of HQC starting from the fourth revision. We point out that changing this sampling approach will lead to non interoperable HQC implementations, as the pseudorandom sampling is repeated by both the encapsulation and decapsulation module.

Furthermore, in [30] the authors proved that the a straightforward, non-constant time rejection sampling approach allows to observe timing variations dependent on the secret key in the deterministic re-encryption executed in the HQC.KEM-DECAPSULATE primitive, leading to a



(a) HQC.KEM-ENCAPSULATE

(b) HQC.KEM-DECAPSULATE

(c) HQC.KEM-KEYGEN

Fig. 6: Operation schedules for HQC.KEM scheme. A white box represents an output value.

successful key recovery attack based on such a side channel. Thus, previous HQC implementations adopting a rejection-sampling based approach for the generation of sparse polynomials, as the HLS HQC implementation presented in [18], do not provide timing-side channel immunity.

From an hardware design perspective, the notable aspects of the sampling algorithm for sparse polynomials with Hamming weight w, described in [29], are the use of an exact amount of randomness, $32 \cdot w$ bits, and the requirement to perform modulo operations between a 32-bit dividend and a 16-bit divisor, for which we used a *shift-and-subtract* algorithm. In our design, we realized a pipelined divisor where each stage can be configured to merge multiple iterations of the *shift-and-subtract* algorithm. We evaluated the maximum operating frequency of this design while varying the number of merged iterations. We determined that, to prevent the divider functional unit from being the limiting factor in the overall design operating frequency, each stage must compute a single iteration of the *shift-and-subtract* algorithm. Differently from [21], the support for multiple parameter sets mandates to manage multiple values for the weight w of sparse polynomials to the end of extracting the correct amount of random bits.

3.4 Schedule of Operations

To provide a hardware design that limits the resources needed for the memories and their input/output ports (which is also particularly relevant for a ASIC deployment), we limited the number of memories to an industrially reasonable quantity of 5, trading-off some performance advantages coming from the concurrent execution of some operations (e.g., polynomial multiplications). Furthermore, each memory port is exclusively bound to a single computing module during the execution of a specific state of the global FSA, using the simplest arbitration logic. Fig. 7 shows the top level design and the interconnection between the main modules and memories. The interconnection has been optimized manually to pair a module and memory only if it is strictly needed. The choice among the algorithms HQC.KEM-KEYGEN, HQC.KEM-ENCAPSULATE and HQC.KEM-DECAPSULATE is performed via a selection signal in the global FSA that activates the execution of one of the three sub-FSA from [21], here reported in Fig. 6. The schedule of operations for the unified design is the same, regardless of the parameter sets. The choice of the parameter set is added by means of a selection signal in the global FSA that is meant to determine both the behavior of the main modules and the portions of memories to be addressed. Compared with the algorithmic description of the HQC components provided in Section 2, in its actual hardware design, we opted for a flattened HDL module hierarchy, without a separate component for the HQC.PPKE module, to avoid replicated components in HQC.KEM.

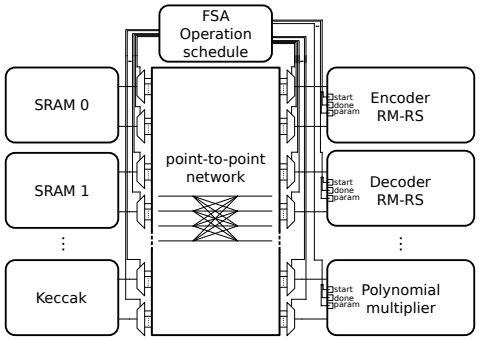


Fig. 7: Datapath of modules controlled by the global FSA.

3.4.1 HQC.KEM *Key Generation Module*

The HQC.KEM-KeyGen schedule, reported in Fig. 6c, starts by expanding the seed φ into the dense public polynomial h (step K_0 and K_1), and expanding the seed γ into the two private polynomials x and y (step K_2, K_3, K_4).

Devising the scheduling of the HQC.KEM-KeyGen, allowed us to propose a variation to the HQC specification to speed up the algorithm, at no resource cost or security loss. Since the sparse polynomial y is employed before the polynomial x, and the latter is not used during decapsulation, it makes sense to sample y as the first element. This avoids to waste time discarding the first $w \cdot 32$-bits linked to x squeezed from the SHAKE module, as per the HQC specification, also simplifying the procedure. Moreover, the proposed solution may also be beneficial in terms of lower memory usage, although our low-latency design is not taking advantage in that regard. The HQC.KEM-KeyGen schedule goes on by sampling the sparse polynomial y in step K_3, and multiplying it by h in step K_4, while x is being sampled. Finally in step K_5 we construct the syndrome s adding the sparse polynomial x to the result of the multiplication. It would be possible to sample the sparse polynomials x and y in parallel with h by replicating the required SHAKE module, but we sticked to a sequential schedule to not raise the hardware resource requirements.

3.4.2 HQC.KEM *Encapsulation Module*

As depicted in Fig. 6a, the module starts by absorbing with Hash-G the message $\mathbf{m}$, the salt, the syndrome s, and the public key seed φ (step E_0) to generate the encryption seed θ (step E_1). Subsequently, while the parity-check polynomial h is expanded from φ, the message $\mathbf{m}$ is encoded by the RS-RM encoder (steps E_2 and E_3). After absorbing the encryption salt θ in the CSPRNG domain, the sparse polynomials r_a, r_b, and e are sampled. Scheduling these operations, we propose a further optimization of the HQC specification, in the same line of thought as the one in HQC.KEM-KeyGen. We sample the values r_b, e, and then r_a, allowing the parallel execution of two sparse-to-dense multiplications during the first two sampling actions (steps E_5 to E_7). Also this optimization was recently included by the latest HQC specification [7]. The sparse polynomial e is added to result of the multiplication of s by r_b in step E_8 by flipping the binary coefficients at the index locations specified by e, and the result is added to the codeword obtaining $\mathbf{v}$ (step E_9). Finally, the addition of the sparse polynomial r_a to the result of the multiplication of h by r_b is carried out producing the polynomial u (step E_{10}). After absorbing the message $\mathbf{m}$ and the ciphertext $u\|\mathbf{v}$ in the Hash-K domain (step E_{11}), the first 512-bits of SHAKE's internal state will contain the session key K, securely stored in memory.

3.4.3 HQC.KEM *Decapsulation Module*

The scheduling of this module, represented in Fig. 6b, starts benefiting from the optimization of the HQC.KEM-KeyGen procedure, as it is possible to sample the sparse polynomial y (step D_1) without discarding any output from the SHAKE based CSPRNG, as it would have been necessary if the x polynomial were sampled first. The sparse polynomial y is multiplied by the first part of the ciphertext u (step D_2) and added to the second part $\mathbf{v}$ (step D_3). After step D_3, the result is a decodable message $\mathbf{m}'$, fed to the concatenated RS-RM decoder in step D_4, concluding the HQC.PPKE decryption. Due to the HHK framework, the steps from E_0 to E_{10} repeat the encapsulation procedure using the decoded message $\mathbf{m}'$, obtaining $u'\|\mathbf{v}'$: if these values do not match received ciphertext $u\|\mathbf{v}$ (step D_5), or the decoding of the message was not successful, the secret value σ from HQC's private key is employed to derive a fake session key through SHAKE, instead of $\mathbf{m}'$. The session key is generated absorbing the ciphertext $u\|\mathbf{v}$ (step D_6), and storing the first 512-bits of SHAKE's internal state (step D_7).

TABLE 2: RS encoder/decoder area.

Component	Resources (eSlices)			
	`HQC-128`	`HQC-192`	`HQC-256`	Unified
Encoder	63	65	112	219
Decoder	793	912	1537	1607
Total	856	977	1649	1826

TABLE 3: Top-level design area.

Component	Resources (eSlices)			
	Keygen	Encap	Decap	Unified
Computing modules	3302	3912	6717	6736
Crossbar and global FSA	145	261	261	354
Static RAMs	4278	4280	5976	5976

4 Experimental Results

We used the SystemVerilog language to realize a register transfer level (RTL) unified design compatible with all HQC KEM operations and parameter sets, along with a RTL design specialized for each endpoint in a client/server communication, also supporting all parameter sets. The design for the accelerator at client side includes the encapsulation operation only, while the one at server side includes the key generation and decapsulation operations. We performed the synthesis using Vivado 2023.1 (with default synthesis and implementation strategies) and the `xc7a200tfbg484-3` target, which is the largest chip belonging to the Artix-7 family sold by AMD (Xilinx), with the best speed grade, to align with the setup used by other works and the reference platform chosen by NIST. Our design was tested via behavioral simulations with the `cocotb` framework and following the IEEE 1800.2 Universal Verification Methodology, and validated in post-implementation via the Known Answer Tests (KATs) distributed with the latest HQC specification. The designs were transferred via an UART module on a Digilent Arty A7-100T board with a `xc7a100tcsg324-1` FPGA. To compare the area results of a target with many heterogeneous resources like FPGAs, we use an improved *equivalent slices* (eSlice) synthetic area indicator similar to the one introduced in [21], which takes into account the usage of lookup tables (LUT), flip-flops (FF), block memories (BRAM), and digital signal processing (DSP) resources of the specific 7 series FPGAs at hand. Specifically, the eSlice metric considers the area of LUTs and FFs resulting from the out-of-context synthesis of the original component without the possibility to use BRAMs and DSPs. For the Artix-7 family, the available `DSP48E1`s realize a integer multiply and accumulate operation with the possibility to perform

TABLE 4: Comparison with other unified Artix-7 FPGA accelerators for post-quantum KEM algorithms. The highlighted designs also support all parameter sets. The Area × Time (AT) product is expressed in eSlices · ms.

security	Design scheme	ref.	variant	Resource LUT	FF	BR	DSP	eSlice	Freq. MHz	Keygen μs	Keygen AT	Encapsulation μs	Encapsulation AT	Decapsulation μs	Decapsulation AT
AES-128	CRYSTALS-Kyber	[13]	-	9347	8186	6	4	4147	220	10	40	15	62	20	85
	CRYSTALS-Kyber	[14]	-	7412	4644	3	2	2758	161	24	65	32	87	42	115
	HQC	our	-	26561	13636	28	0	12471	143	39	488	82	1027	128	1597
	HQC	[19]	balanced	18662	7088	22	8	10406	164	96	1000	204	2122	294	3059
	HQC	[19]	high-speed	20011	7484	24	8	11167	178	89	989	126	1407	209	2333
	HQC	[20]	-	24591	11270	68	0	20564	178	112	2311	225	4621	393	8087
	BIKE	[31]	lightweight	12319	3896	9	7	5930	121	3826	22691	446	2646	6950	41216
	BIKE	[31]	trade-off	19607	5008	17	9	9717	100	1870	18171	280	2721	4210	40909
	BIKE	[31]	high-speed	25549	5462	34	13	15344	113	1681	25800	133	2037	1168	17924
	Classic McEliece	[32]	lightweight	23890	45658	138	5	36007	112	1161	41794	1518	54653	79286	2854841
	Classic McEliece	[32]	high-speed	40018	61881	178	4	48173	113	265	12789	885	42631	8584	413520
AES-192	S. NTRU Prime	[33]	low area	9574	4399	8	18	6617	128	4917	32535	228	1512	669	4427
	S. NTRU Prime	[33]	high-speed	41428	26381	36	31	22265	140	457	10182	36	796	78	1748
	CRYSTALS-Kyber	[13]	-	10434	9473	8	6	5218	220	12	64	18	93	23	119
	CRYSTALS-Kyber	[14]	-	7412	4644	3	2	2758	161	39	108	49	135	62	171
	HQC	our	-	26561	13636	28	0	12471	143	96	1200	200	2497	305	3799
	BIKE	[31]	lightweight	13850	4010	15	7	7584	116	15302	116048	1353	10265	20526	155668
	BIKE	[31]	trade-off	20049	5039	17	9	9827	100	6930	68101	800	7862	11980	117727
	BIKE	[31]	high-speed	25811	5460	34	13	15410	113	6027	92869	372	5728	5354	82505
AES-256	CRYSTALS-Kyber	[13]	-	11527	10767	10	8	6184	220	16	101	22	135	27	169
	CRYSTALS-Kyber	[14]	-	7412	4644	3	2	2758	161	58	161	70	194	86	238
	HQC	our	-	26561	13636	28	0	12471	143	181	2262	378	4718	574	7158
	BIKE	[31]	lightweight	13973	4002	34	7	11643	113	42558	495497	3035	35341	46168	537536
	BIKE	[31]	trade-off	21373	5160	34	9	13762	94	19649	270409	1851	25474	27872	383579
	BIKE	[31]	high-speed	26441	5601	34	13	15567	111	16198	252157	811	12622	11901	185261

a preliminary addition on one of the input operands. The eSlice metric in [21] did not consider any DSP. In this work, we derived 538 LUTs and 232 FFs as the equivalent cost of a DSP by synthesizing a RTL description of the `DSP48E1` reported in the AMD user guide UG479 with the attribute `use_dsp = "no"`. As far as the cost of one BRAM goes, in [21] the eSlice metric did not consider the area due to the `read` and `write` ports of the memories, while in this work we explicitly annotated the RTL description of a whole BRAM realizing a 32 kb Simple Dual Port RAM (in compliance with the AMD user guide UG901) with the `ram_style = "distributed"` attribute and obtained from the out-of-context synthesis an equivalent cost equal to 848 LUTs and 548 FFs. Since one slice in a AMD 7-Series FPGA includes four 6-input LUTs and eight FFs, the number of eSlices is derived as: $\mathbf{max}\{\lceil(\text{Num. of LUTs})/4\rceil, \lceil(\text{Num. of FFs})/8\rceil\}$. Although the eSlice metric allows to compare designs with different FPGA resources (e.g., ours that is w/o DSP, and the one from [19]), it assumes a BRAM and/or a DSP used at the fullest and does not consider the area needed for the routing in the post-implementation step. The design of the RS encoder and decoder described in the previous section allows us to exibit two components suitable for all parameter sets that improve the area figures of the designs based on separate decoders/encoders in [21]. Tab. 2 shows that the unified decoder introduces only a 10% area overhead when compared with the decoder used in `HQC-256` and requires only half of the area needed to accomodate one decoder for each parameter set. The unified encoder doubles the area required by the one in `HQC-256`; nonetheless, it benefits from a 10% area reduction when compared with the sum of the areas of three separate components. We note that in the small area savings exhibited by the unified encoder, a notable figure (of interest for ASIC targets) is the halving in the number of used Flip-Flops. Tab. 3 shows the area figures of the computing modules (arithmetic units, RMRS enc./dec., CSPRNG, etc.), the crossbar interconnection network, and the SRAMs composing the top modules that realize the primitives HQC.KEM-KeyGen, HQC.KEM-Encapsulate, HQC.KEM-Decapsulate from [21], as well as a unified top-level module. The unified design comes at a negligible cost w.r.t. the decapsulation primitive, and mainly determined by the more complex global FSA and interconnection network. Moreover, the area occupied by the logic is comparable to the one dedicated to the SRAMs, typical condition of many code-based schemes. Tab. 4 compares several Artix-7 FPGA accelerators of Post-Quantum KEM from the NIST PQC contest. The lattice-based candidates Streamlined NTRU Prime and CRYSTALS-Kyber are reported in the first rows in each security category, followed by the code-based schemes HQC, BIKE and Classic McEliece. Highlighted rows point out accelerators with a design compliant with all parameter sets mandated by the official specification of the corresponding KEM. Our unified solution have lower latency and higher efficiency than the 128-bit security (only) design in [19]. In particular we exhibit a 2.28×, 1.53×, and 1.63× gain in terms of execution time for keygen, encap. and decap., respectively, while taking only 11% more resources than [19]. In comparison with the design providing (only) 128-bit security that is reported in [20], we use less than half of the BRAMs and supports all the parameters sets, showing a remarkable efficiency. More specifically, our design shows a 2.87×, 2.74×, and 3.07× improved latency for the keygen, encap., and decap. primitives, respectively, while taking 40% less resources than [20]. CRYSTALS-Kyber has several low-latency, compact, and efficient hardware implementations: among all the other code-based candidate designs, our solution is the only one showing metrics in the same order of magnitude. This further confirms that HQC is a noteworthy alternative to

TABLE 5: Comparison with other client/server Artix-7 FPGA accelerators for post-quantum KEM algorithms. The highlighted designs also support all parameter sets. The Area × Time (AT) product is expressed in eSlices · ms.

Design				Resource (server/client)					Freq.	Server		Client	
security	scheme	ref.	variant	LUT	FF	BR	DSP	eSlice	MHz	μs	AT	μs	AT
AES-128	BIKE	[34]	high-speed	126510/91422	51492/46208	357/276	0/0	107312/81262	91/91	580	62241	30	2438
	BIKE	[34]	lightweight	31792/19804	17805/11401	44/30	0/0	17170/11311	91/91	5710	98041	30	339
	HQC	our	-	26533/15847	13688/9335	28/20	0/0	12464/8202	156/172	153	1911	68	562
	Kyber	[14]	-	7412/6785	4644/3981	3/3	2/2	2758/2602	161/167	65	179	30	79
AES-192	BIKE	[34]	high-speed	124891/72725	53067/37795	360/236	0/0	107543/68108	91/91	1710	183899	60	4086
	BIKE	[34]	lightweight	31411/19979	20181/12282	46/28	0/0	17499/10931	91/91	19270	337206	80	874
	HQC	our	-	26533/15847	13688/9335	28/20	0/0	12464/8202	156/172	367	4579	166	1365
	Kyber	[14]	-	7412/6785	4644/3981	3/3	2/2	2758/2602	161/167	102	280	47	123
AES-256	HQC	our	-	26533/15847	13688/9335	28/20	0/0	12464/8202	156/172	692	8630	314	2580
	Kyber	[14]	-	7412/6785	4644/3981	3/3	2/2	2758/2602	161/167	145	399	68	176

CRYSTALS-Kyber, particularly for the AES-128 equivalent security margin, as the penalties in latency and efficiency when using higher security guarantees grow faster in all considered code-based schemes compared to CRYSTALS-Kyber. When compared to BIKE, the high-speed variant of [31] reports a similar but higher eSlice resource usage, while being from $1.62\times$ to $89\times$ slower than our solution. The HW design of the Classic McEliece scheme in its 128-bit security variant [32] (named mceliece-3488064) in the official specification) is the only one suitable for a low-end target as an Artix-7. As far as performance goes, the execution times of the key generation, encap. and decap. operations results to be from $6.8\times$ to $67\times$ slower than ours. Our HQC design can compete with hardware designs for lattice-based Streamlined NTRU Prime, roughly positioning itself somewhere between the low area and high-performance solutions of [33]. As for any NTRU-derived scheme, the key generation is the most expensive operation and not compare favorably with HQC. Tab. 5 shows the results of the syntheses of the designs of two distinct accelerators meant to be used by a client and a server, respectively (e.g., to execute the cryptographic operations mandated by a TLS communication handshake). The BIKE designs in [34] maximized the parallelism of the computing modules to fill the entire FPGA fabric of an Artix-7 50 (lightweight variant) and an Artix-7 200 (high-speed variant), providing top level designs specialized for the parameter sets having security margins of AES-128 and AES-192. Due to the expensive inversion operation in the keygen, the server designs are penalized both in terms of latency and, by a larger margin, in efficiency with respect to our HQC design. On the other end, the lightweight client designs show $\approx 2\times$ better latency and efficiency figures than our design, partially due to the larger number of operations carried out during the HQC encapsulation. Finally, the CRYSTALS-Kyber client and server designs in [14] have from $2.3\times$ to $4.7\times$ better latency than ours, with an efficiency improved by $7\times$ to $21\times$. For the sake of completeness, we evaluated our design using an ASIC toolchain composed of Yosys and OpenROAD for synthesis and place-and-route, respectively. We used the FreePDK45 technology library to complete the open-source ASIC toolchain, and evaluated using the typical process corner of 1.1V at 25°C. Due to the current lack of maturity of the OpenRAM memory compiler we resorted to exposing the memory buses as I/O pins of the chip. Despite this fact, the resulting design reached a maximum operating frequency of 419 MHz, occupying an area of 0.496mm^2. Finally, we briefly comment on the implementation of a HSM employing either our accelerator or a software realization of HQC running on a general purpose CPU. We compare the latency of our hardware design with the runtime on a Rockchip RK3288 ARM Cortex-A17 CPU at 1.8MHz, produced with a 28nm HKMG process node technology (similar to the 28nm HPL used for the AMD Artix-7 XC7A35T), both having a similar bulk price of 20 US $. We note that our design can be accommodated in each platform of the Artix-7 lineup (XC7A12T–XC7A200T), and the XC7A35T one is the least expensive. The sum of the runtime execution of key generation, encapsulation, and decapsulation algorithm amounts to (4.94, 10.2, 17.7) ms, for the three parameter sets, while our FPGA unified accelerator takes (0.25, 0.60, 1.13) ms, which makes it $19.8\times$, $16.9\times$, and $15.6\times$ faster than the execution on the said CPU.

5 Concluding Remarks

We presented a unified and versatile HQC hardware accelerator supporting all KEM operations and parameter sets, reporting the best latency, area and efficiency figures among the available HQC and other code-based designs. Our design can be enhanced with countermeasures against power-based and fault-based side-channel attacks (SCAs). Since HQC is not conceived to minimize the SCA attack surface (as, e.g., Raccoon [35] does), applying SCA protections to it must consider not only the polynomial operations and the details related to the HHK transformation, but also the structures of encoder and decoder. Further developments can consider a broader design exploration to obtain the least silicon area with given time constraints, which can be interesting for some IoT devices where an acceptable latency for each KEM operation is in the few tenths of ms.

Acknowledgments

This work was supported in part by project SERICS (PE00000014) under the NRRP MUR program funded by the EU-NGEU, and the by the project POINTER (ID-2022M2JLF2) under the PRIN 2022 MUR program.

References

[1] "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions," NIST standard FIPS 202, 2015. [Online]. Available: https://doi.org/10.6028/NIST.FIPS.202

[2] NIST Information Technology Laboratory, "NIST post-quantum standardization contest," Accessed: Feb. 2025. [Online]. Available: https://csrc.nist.gov/pqc-standardization

[3] "Module-lattice-based key-encapsulation mechanism standard," RFC NIST standard FIPS 203, Aug. 2023. [Online]. Available: http://dx.doi.org/10.6028/NIST.FIPS.203.ipd

[4] "Announcing Approval of Three Federal Information Processing Standards (FIPS) for Post-Quantum Cryptography," Accessed: Feb. 2025, Aug. 2024. [Online]. Available: https://csrc.nist.gov/news/2024/postquantum-cryptography-fips-approved
[5] N. Aragon *et al.*, "BIKE Supporting Documentation," Accessed: Feb. 2025, 2022. [Online]. Available: https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf
[6] D. J. Bernstein *et al.*, "Classic McEliece Supporting Documentation," Accessed: Feb. 2025, 2022. [Online]. Available: https://classic.mceliece.org/mceliece-spec-20221023.pdf
[7] C. Aguilar Melchor *et al.*, "HQC (Hamming Quasi-Cyclic) Specification," Accessed: Feb. 2025, 10 2024. [Online]. Available: https://pqc-hqc.org/doc/hqc-specification_2024-10-30.pdf
[8] C. M. Chung *et al.*, "NTT Multiplication for NTT-unfriendly Rings New Speed Records for Saber and NTRU on Cortex-M4 and AVX2," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 2, pp. 159–188, 2021. [Online]. Available: https://doi.org/10.46586/tches.v2021.i2.159-188
[9] C. Aguilar Melchor *et al.*, "Optimized HQC implementation for AVX2," Accessed: Feb. 2025. [Online]. Available: https://pqc-hqc.org/doc/hqc-optimized-implementation_2024-02-23.zip
[10] R. Aissaoui, J. Deneuville, C. Guerber, and A. Pirovano, "A Performant Quantum-Resistant KEM for Constrained Hardware: Optimized HQC," in *Proceedings of the 21st International Conference on Security and Cryptography, SECRYPT 2024, Dijon, France, July 8-10, 2024*. SCITEPRESS, 2024, pp. 668–673. [Online]. Available: https://doi.org/10.5220/0012757800003767
[11] R. L. Schröder, S. Gast, and Q. Guo, "Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks," in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/schr%C3%B6der
[12] G. Goy, J. Maillard, P. Gaborit, and A. Loiseau, "Single trace HQC shared key recovery with SASCA," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2024, no. 2, pp. 64–87, 2024. [Online]. Available: https://doi.org/10.46586/tches.v2024.i2.64-87
[13] V. B. Dang, K. Mohajerani, and K. Gaj, "High-Speed Hardware Architectures and FPGA Benchmarking of CRYSTALS-Kyber, NTRU, and Saber," *IEEE Trans. Computers*, vol. 72, no. 2, pp. 306–320, 2023. [Online]. Available: https://doi.org/10.1109/TC.2022.3222954
[14] Y. Xing and S. Li, "A Compact Hardware Implementation of CCA-Secure Key Exchange Mechanism CRYSTALS-KYBER on FPGA," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2021, no. 2, pp. 328–356, 2021. [Online]. Available: https://doi.org/10.46586/tches.v2021.i2.328-356
[15] C. Paquin, D. Stebila, and G. Tamvada, "Benchmarking Post-quantum Cryptography in TLS," in *Post-Quantum Cryptography - 11th International Conference, PQCrypto 2020, Paris, France, April 15-17, 2020, Proceedings*, ser. LNCS, vol. 12100. Springer, 2020, pp. 72–91. [Online]. Available: https://doi.org/10.1007/978-3-030-44223-1_5
[16] Y. Tu, P. He, Ç. K. Koç, and J. Xie, "LEAP: Lightweight and Efficient Accelerator for Sparse Polynomial Multiplication of HQC," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 31, no. 6, pp. 892–896, 2023. [Online]. Available: https://doi.org/10.1109/TVLSI.2023.3246923
[17] P. He, Y. Tu, and J. Xie, "LOCS: LOw-Latency and ConStant-Timing Implementation of Fixed-Weight Sampler for HQC," in *IEEE International Symposium on Circuits and Systems (ISCAS 2023), Monterey, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 1–5. [Online]. Available: https://doi.org/10.1109/ISCAS46773.2023.10181319
[18] C. Aguilar Melchor *et al.*, "Towards Automating Cryptographic Hardware Implementations: A Case Study of HQC," in *CBCrypto 2022, Trondheim, Norway, May 29-30, 2022*, ser. LCNS, vol. 13839. Springer, 2022, pp. 62–76. [Online]. Available: https://doi.org/10.1007/978-3-031-29689-5_4
[19] S. Deshpande, C. Xu, M. Nawan, K. Nawaz, and J. Szefer, "Fast and Efficient Hardware Implementation of HQC," in *Selected Areas in Cryptography - SAC 2023*, ser. LNCS, vol. 14201. Springer, 2023, pp. 297–321. [Online]. Available: https://doi.org/10.1007/978-3-031-53368-6_15
[20] C. Li, S. Song, J. Tian, Z. Wang, and Ç. K. Koç, "An Efficient Hardware Design for Fast Implementation of HQC," in *36th IEEE International System-on-Chip Conference, SOCC 2023, Santa Clara, CA, USA, September 5-8, 2023*. IEEE, 2023, pp. 1–6. [Online]. Available: https://doi.org/10.1109/SOCC58585.2023.10257054
[21] F. Antognazza, A. Barenghi, G. Pelosi, and R. Susella, "A High Efficiency Hardware Design for the Post-Quantum KEM HQC," in *IEEE International Symposium on Hardware Oriented Security and Trust (HOST 2024), Tysons Corner, VA, USA, May 6-9, 2024*. IEEE, 2024, pp. 431–441. [Online]. Available: https://doi.org/10.1109/HOST55342.2024.10545409
[22] D. Hofheinz, K. Hövelmanns, and E. Kiltz, "A Modular Analysis of the Fujisaki-Okamoto Transformation," in *15th International Conference on Theory of Cryptography (TCC 2017), Baltimore, MD, USA, November 12-15, 2017*, ser. LNCS, vol. 10677. Springer, 2017, pp. 341–371. [Online]. Available: https://doi.org/10.1007/978-3-319-70500-2_12
[23] S. Lin and D. J. Costello Jr., *Error control coding - fundamentals and applications*. Prentice Hall, 1983.
[24] V. Y. Pan, "Methods of computing values of polynomials," *Russian Mathematical Surveys*, vol. 21, no. 1, pp. 105 – 136, 1966. [Online]. Available: https://api.semanticscholar.org/CorpusID:250869179
[25] J. L. Massey, "Shift-register synthesis and BCH decoding," *IEEE Trans. Inf. Theory*, vol. 15, no. 1, pp. 122–127, 1969. [Online]. Available: https://doi.org/10.1109/TIT.1969.1054260
[26] Y. Wu, "New Scalable Decoder Architectures for Reed-Solomon Codes," *IEEE Trans. Commun.*, vol. 63, no. 8, 2015. [Online]. Available: https://doi.org/10.1109/TCOMM.2015.2445759
[27] Y. Be'ery and J. Snyders, "Optimal soft decision block decoders based on fast Hadamard transform," *IEEE Trans. Inf. Theory*, vol. 32, no. 3, pp. 355–364, 1986. [Online]. Available: https://doi.org/10.1109/TIT.1986.1057189
[28] G. Bertorni, J. Daemen, M. Peeters, and G. V. Assche, "The Keccak reference," Accessed: Feb. 2025, 2011. [Online]. Available: https://keccak.team/files/Keccak-reference-3.0.pdf
[29] N. Sendrier, "Secure Sampling of Constant-Weight Words - Application to BIKE," *IACR Cryptol. ePrint Arch.*, p. 1631, 2021. [Online]. Available: https://eprint.iacr.org/2021/1631
[30] Q. Guo, C. Hlauschek, T. Johansson, N. Lahr, A. Nilsson, and R. L. Schröder, "Don't Reject This: Key-Recovery Timing Attacks Due to Rejection-Sampling in HQC and BIKE," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 3, pp. 223–263, 2022. [Online]. Available: https://doi.org/10.46586/tches.v2022.i3.223-263
[31] J. Richter-Brockmann, M. Chen, S. Ghosh, and T. Güneysu, "Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 1, pp. 557–588, 2022. [Online]. Available: https://doi.org/10.46586/tches.v2022.i1.557-588
[32] P. Chen *et al.*, "Complete and Improved FPGA Implementation of Classic McEliece," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2022, no. 3, pp. 71–113, 2022. [Online]. Available: https://doi.org/10.46586/tches.v2022.i3.71-113
[33] B. Peng, A. Marotzke, M. Tsai, B. Yang, and H. Chen, "Streamlined NTRU Prime on FPGA," *J. Cryptogr. Eng.*, vol. 13, no. 2, pp. 167–186, 2023. [Online]. Available: https://doi.org/10.1007/s13389-022-00303-z
[34] A. Galimberti *et al.*, "FPGA implementation of BIKE for quantum-resistant TLS," in *25th Euromicro Conference on Digital System Design (DSD 2022), Maspalomas, Spain, August 31 - Sept. 2, 2022*. IEEE, 2022, pp. 539–547. [Online]. Available: https://doi.org/10.1109/DSD57027.2022.00078
[35] R. del Pino *et al.*, "Raccoon Documentation," Accessed: Feb. 2025, 2023. [Online]. Available: https://raccoonfamily.org/wp-content/uploads/2023/07/raccoon.pdf

6 Biography Section

Francesco Antognazza is a Ph.D. student at Politecnico di Milano, Italy. His research interests include efficient and side channel-resistant hardware implementations of cryptographic primitivies, with a focus on post-quantum asymmetric cryptosystems.

Alessandro Barenghi is an associate professor at Politecnico di Milano, Italy. His interests include computer and network security, formal languages and compilers and he has published more than 100 papers in international peer reviewed venues.

Gerardo Pelosi is an associate professor at Politecnico di Milano, Italy. His main research interests are in computer security, cryptography, security in hardware and in the area of data security and privacy. He has published more than 100 papers in international peer reviewed journals and conference proceedings and is co-inventor of 10 patents concerning the design of cryptographic systems.