Fast and Efficient Hardware Implementation of HQC

Sanjay Deshpande[1], Chuanqi Xu[1], Mamuri Nawan[2], Kashif Nawaz[2], and Jakub Szefer[1]

[1] CASLAB, Department of Electrical Engineering, Yale University, New Haven, USA
sanjay.deshpande@yale.edu, chuanqi.xu@yale.edu, jakub.zefer@yale.edu
[2] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
mamuri@tii.ae, kashif.nawaz@tii.ae

Abstract. This work presents a hardware design for constant-time implementation of the HQC (Hamming Quasi-Cyclic) code-based key encapsulation mechanism. HQC has been selected for the fourth round of NIST's Post-Quantum Cryptography standardization process and this work presents the first, hand-optimized design of HQC key generation, encapsulation, and decapsulation written in Verilog targeting implementation on FPGAs. The three modules further share a common SHAKE256 hash module to reduce area overhead. All the hardware modules are parametrizable at compile time so that designs for the different security levels can be easily generated. The design currently outperforms the other hardware designs for HQC, and many of the fourth-round Post-Quantum Cryptography standardization process, with one of the best time-area products as well. For the combined HighSpeed design targeting the lowest security level, we show that the HQC design can perform key generation in 0.09 ms, encapsulation in 0.13 ms, and decapsulation in 0.21 ms when synthesized for an Xilinx Artix 7 FPGA. Our work shows that when hardware performance is compared, HQC can be a competitive alternative candidate from the fourth round of the NIST PQC competition.

Keywords: HQC · Hamming Quasi-Cyclic · PQC · KEM · Key Encapsulation Mechanism · Post-Quantum Cryptography · FPGA · Hardware Implementation

1 Introduction

Since 2016 NIST has been conducting a standardization process with the goal to standardize cryptographic primitives that are secure against attacks aided by quantum computers. There are today five main families of post-quantum cryptographic algorithms: hash-based, code-based, lattice-based, multivariate, and isogeny-based cryptography. Very recently NIST has selected one algorithm for standardization in the key encapsulation mechanism (KEM) category, CRYSTALS-Kyber, and four fourth-round candidates that will continue in the process. One of the four fourth-round candidates is HQC. It is a code-based KEM based on structured codes.

As the standardization process is coming to an end after the fourth round, the performance as well as hardware implementations of the algorithms are becoming very important factor in selection of the algorithms to be standardized. The motivation for our work is to understand how well hand-optimized HQC hardware implementation can be designed and realized on FPGAs. To date, most of the post-quantum cryptographic hardware has focused on lattice-based candidates, with code-based algorithms receiving much less attention. All existing hardware implementations for HQC are based on either high-level synthesis (HLS) [1,3] or are hardware-software co design [22]. Our design is first full hardware, hand-optimized design of HQC. While HLS can be used for rapid prototyping, in our experience it cannot yet outperform Verilog or other hand optimized designs. Indeed, as we show in this work, our design outperforms the existing HQC HLS design. Further, our design beats the existing hardware-software co-design implementation in terms of time taken to perform key generation, encapsulation, and decapsulation.

In addition, our hardware design competes very well with the hardware designs for other candidates currently in the fourth round of NIST's process: BIKE, Classic McEliece, and SIKE. The presented design has best time-area product as well as time for key generation and decapsulation compared to the hardware for these designs. We also achieve similar time-area product for encapsulation when compared to BIKE. Due to limited breakdown of data for SIKE's hardware [17] comparison to SIKE for all aspects is more difficult, but we believe our design is better since for similar area cost, their combined encapsulation and decapsulation times are two orders of magnitude larger. Detailed comparison to related work is given in Section 3. As this work aims to show, code-based designs such as HQC can be realized very efficiently when optimized hardware is developed. Further, our design is constant-time, eliminating timing-based attacks. We believe our work shows that HQC can be a strong contender in the fourth round of NIST's process. The list of contributions our design includes:

- We provide the first hand-optimized, fully specification-compliant FPGA implementation of HQC, that includes key generation, encapsulation, and decapsulation, as well as a joint design of all three operations, adherent to the latest (fourth-round) HQC specification.
- We provide an improved SHAKE256 module which is based on Keccak module given in [25]. With our improvement, our hash module design runs two time faster than the existing one from [25]. Also, improving the overall time-area product.
- We provide first hardware implementations and evaluation for two variants of constant-time fixed-weight vector generation, namely Constant Weight Word fixed-weight vector generation [24] and a novel Fast and Non-Biased fixed-weight vector generation algorithm, which is based on fixed-weight vector generation process given in [1].
- We also provide an implementation of a parameterized binary field polynomial multiplication unit that uses half the Block RAM when compared to the existing state-of-the-art while providing better performance.
- Our designs are constant-time providing protection against the timing side-channel attacks and providing compile-time parameters to switch between different security levels and performances.

We evaluate the resource requirements and performance numbers of our designs on a Xilinx Artix 7 FPGA as it is a defacto standard for the evaluation of NIST PQC hardware designs. For all our hardware designs, we report the resource utilization in terms of Slices, Look Up Tables (LUTs)[3], Digital-Signal Processing Units (DSPs), FlipFlops (FF) Block RAM (BRAM). We also report Time which is computed by dividing the number of clock cycles taken per operation by design with the maximum clock frequency of the design. In order to consolidate the overall performance and for comparison with other hardware designs from the literature, we use Time Area Product (T×A = Time×Slices) as a metric. Functional correctness of our modules is ensured by generating the testvectors from the reference software implementation (provided in [2]). These testvectors are then fed into our design via testbenches performing pre- and post-synthesis simulations. The hardware design generated output is then compared with the reference software implementation output.

1.1 Open-Source Design

All our hardware designs reported in this paper are fully reproducible. The source code of our hardware designs is available under an open-source license at https://github.com/caslab-code/pqc-hqc-hardware.

2 Hardware Design of HQC

HQC Key Encapsulation Mechanism (HQC-KEM) consists of three main primitives: Key Generation, Encapsulation, and Decapsulation. The algorithms for each primitive are shown in Appendix 1.A: Algorithm 2, Algorithm 5, and Algorithm 6, respectively. These primitives are built upon the HQC Public Key Encryption (HQC-PKE) primitives shown in Appendix 1.A: Algorithm 2, Algorithm 3, and Algorithm 4, which in turn are composed of more basic building blocks. In this work, we implement optimized and parameterizable hardware designs for all the primitives and the building blocks from scratch. In the following subsections, we briefly discuss all the building blocks and provide comparisons with any existing designs. The main building blocks involved for each of the primitives are as follows:

- Key Generation: Fixed weight vector generator, PRNG based random vector generator, polynomial multiplication, modular addition, and SHAKE256
- Encapsulation: Encrypt, SHAKE256
- Decapsulation: Decrypt, Encrypt, SHAKE256

2.1 Modules Common Across the Design

In this section, we give a high-level overview of hardware designs of the building blocks that are used across the HQC-KEM and HQC-PKE.

[3] We report both Slices and LUTs in our tables since slices can be often partially used based on the optimization strategy of the synthesis tool, which makes slice utilization not a complete indication of the density of the design.

SHAKE256 HQC uses SHAKE256 for multiple purposes e.g., as a PRNG for fixed weight vector generation and random vector generation in Key Generation, as a PRNG for fixed weight vector generation in Encryption, and for hashing in encapsulation and decapsulation. We improve the SHAKE256 module described in [8] (which was originally designed based on `Keccak` design from [25]) to perform SHAKE256 operations. We further tailor the SHAKE256 hardware module as per the requirement for our hardware design. Following is a list of improvements we make to the design of the `SHAKE256` module:

The `SHAKE256` from [25] module has a fixed 32-bit data input and output ports, and has a performance parameter (`parallel_slices`) which represents the number of combinatorial logic units that can be run in parallel inside the round function. The SHAKE256 from [25] did not work for `parallel_slices` > 16. We made significant changes in the control logic to fix this issue. The design now supports up to `parallel_slices = 32`. The time and area results can be seen in Table 1. We note from Table 1, that the time area product improves as we increase the `parallel_slices`. However, we could not add the support `parallel_slices` beyond 32 due to the other constraints of the state size of SHAKE256 (1600 bits) and fixed data port sizes (32 bits) in the way that SHAKE256 is used in our HQC implementation.

The existing `SHAKE256` module from [25] operates with a command-based interface where the number of input bits to be processed, and the number of output bits required is specified before starting the hash operation. Based on the required weight, the fixed weight vector generation process requires pseudorandom bits to be generated from the `SHAKE256` module for a specified input seed. Suppose the generated pseudorandom bits fail to satisfy the conditions to achieve the necessary weight or need a second fixed-weight vector generated from the same seed. In that case, another round of pseudorandom bits is generated from SHAKE256. As per the HQC specification, the internal SHAKE256 state is maintained as starting point for generation of the next set of pseudorandom bits. The original `SHAKE256` module from [25] was not optimized nor designed to support preserving of the SHAKE256 state between invocations. We modified parts of datapath and the control logic to preserve the state. Since our modification of `SHAKE256` holds the current state and does not automatically return to its new input loading state, we modify the operation of the existing forced exit signal to return the `SHAKE256` module to the default state.

Using the existing module from [25] directly, it was not possible to implement the constant-time solution for the fixed weight vector generation since there is no command to request for additional bytes. We modify the existing design and add an additional command that can request additional bytes. The purpose of adding this command is to support and optimize the overall time taken to generate pseudo-random bits required for the fixed weight vector generation process described in Section 2.1. This optimization gives a significant amount of improvement in terms of clock cycles (time), for e.g., in the case of the `hqc128` parameter set, the number of clock cycles taken by the `SHAKE256` module from [25] to facilitate the random bits needed for generating the second fixed weight

Table 1: `SHAKE256` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip. The formula for the time-area product, T×A, is (SLICES * Time).

Parallel Slices	Resources					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
1	496	1,437	0	498	0	163	2,408	14.77	7,325
2	537	1,558	0	466	0	167	1,206	7.22	3,877
4	560	1,625	0	370	0	157	604	3.85	2,156
8	675	1,958	0	280	0	158	302	1.91	1,289
16	972	2,819	0	236	0	164	150	0.91	884
Our Improvement									
32	1,654	4,797	0	191	0	166	74	0.45	744

vector is equal to 639 clock cycles. With our improvements to the module, we achieve the same in 434 clock cycles (i.e., 32% improvement in time). And this improvement is up to 65% in larger parameter sets of HQC.

In addition to the aforementioned changes, we further explored options for optimizing the maximum clock frequency by pipelining the critical path. We note that there are several such critical paths throughout the design, and pipelining each path added severe overhead in terms of clock cycles with minimal improvement in the maximum clock frequency. Consequently, the results presented in Table 1 are optimal time and area results for the given hardware architecture. We use a similar performance parameter `parallel_slices` as described in the original `keccak/SHAKE256` design in [25]. The `SHAKE256` module has a fixed 32-bit data ports, and data input and output is based on typical ready-valid protocol. The results targeting Xilinx Artix 7 `xc7a200t` FPGA are shown in Table 1. The clock cycle numbers provided in the Table 1 are for processing 320-bits input (sample input size chosen as per the seed size used in HQC) and generating one block of output (where each block size is 1088-bits). There are six options for the `parallel_slices`, which provide different time-area trade-offs. We choose `parallel_slices = 32` as it provides the best time-area product. An interface diagram of the `SHAKE256` module is shown in Figure 7a. For brevity, we represent all the ports interfacing with the `SHAKE256` module with $\Leftrightarrow$ in all further block diagrams in this paper.

Polynomial Multiplication HQC uses polynomial multiplication operation in all the primitives of HQC-KEM. The polynomial multiplication operation is multiplication of two polynomials with n components in $\mathbb{F}_2$. After profiling all the polynomial multiplication operations from the HQC specification document and the reference design [2], we note that in all the polynomial multiplication operations, one of the inputs is a sparse fixed weight vector (with weight w or w_r in Table 14) of width n-bits. Consequently we design a sparse polynomial multiplication technique with an interleaved reduction $X^n - 1$ (values of n can be found in Table 14).

The motivation behind our polynomial multiplication unit is as follows: we represent the non-sparse arbitrary polynomial as `arb_poly` and the sparse fixed-

weight polynomial by `sparse_poly`. For `sparse_poly`, rather than storing the full polynomial we only store the indices for non-zero values. Then, the multiplication is performed by left shifting `arb_poly` with each index of `sparse_poly` and then performing reduction of the resultant vector in an interleaved fashion. Since the value of n is large in all parameter sets of HQC, we take a sequential approach for performing the left shift. We implement a sequential left shift module similar to one in [12]. The shift module described [12] uses a register based approach and is not scalable when the length of the input is as large as the n value for the HQC parameters (due to a larger resource utilization and complex routing). This issue is circumvented in our design by implementing a block RAM based sequential variable shift module with a dual port BRAM and small barrel rotation unit. The barrel rotation unit and the block RAM widths are used as performance parameter (`BW` - Block Width) for the shift module and in turn for the whole polynomial multiplication unit. A similar implementation of sequential variable shift module was previously described in [10], however we could not readily use their implementation because the shift module is tightly embedded with the other modules for a different application and we re-implemented our version.

The hardware design of our polynomial multiplication module (`poly_mult`) is shown in Figure 7b. The `arb_poly` input to the poly_mult module is loaded sequentially and the width is of each chunk of `arb_poly` is equal to `BW` (making total number of chunks in polynomial equal to `RAMDEPTH` = `ceil`$(n/$`BW`$)$). We store the least significant part of the polynomial at the lowest address of the block RAM and the most significant part at the highest address. Since the polynomial length in HQC parameters is equal to n and is not divisble by `BW` (n is a prime) we pad the most significant part of the polynomial with zeros. For `sparse_poly`, one index is loaded at a time. While performing the shift operation we also perform the reduction $(X^n - 1)$ in an interleaved fashion. As the result of multiplying two n-bit polynomials could be a $2n$-bit polynomial and reduction of $2n$-bit polynomial to $(X^n - 1)$ in $\mathbb{F}_2$ is equivalent to slicing of the $2n$-bit polynomial into two parts of n-bit polynomials and then performing a bitwise XOR. As result, when the shift operation is performed on each chunk we also compute the address value (`ADDR_2N`) (signifying the degree of the resultant polynomial). If we notice that this degree of the resultant polynomial is greater that n we perform `XOR` of this chunk to the lower chunk by decoding the address based on the value of `ADDR_2N`. We perform similar operation over all the indices of the `sparse_poly` to achieve the final multiplied resultant value.

The clock cycles taken by our `poly_mult` module for one polynomial multiplication can be computed using the following formula where W_{SPARSE} is weight of the sparse polynomial, n is length of the polynomial, `BW` is the block width, 3 cycles represents the number of pipeline stages and 2 cycles are for the `start` and `done` synchronization with interfacing modules. The clock cycles taken for shift and interleaved reduction for one index is $(3+$`ceil`$(n/$`BW`$))$. Our `poly_mult` module is constant time and we achieve that by fixing the W_{SPARSE} to a specific value (w and w_r) based on the parameter set.

Table 2: Comparison of our area and timing information `poly_mult` module with the other sparse polynomial multiplication units targeting Artix 7 board with `xc7a200t` FPGA chip.

	Resources								
`BW`	Logic			Memory		F	Cycles	Time	T x A
(bits)	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
Our `poly_mult` module, Polynomial Length* = 12,323, $W^*_{SPARSE} = 71$									
32	134	396	0	181	1	270	27,621	0.10	14
64	202	599	0	205	2	277	13,918	0.05	10
128	486	1,438	0	456	4	238	7,102	0.03	14
General Sparse Multiplier, Polynomial Length* = 12,323, $W^*_{SPARSE} = 71$ [19]									
32	132	319	0	127	2	234	27,691	0.12	16
64	197	549	0	190	4	222	13,988	0.06	12
128	378	1,136	0	381	8	185	7,172	0.04	15
Sparse Multiplier, Polynomial Length* = 10,163, $W^*_{SPARSE} = 71$ [15]									
32	100		—	—	2	240	158,614	0.66	66
64	157		—	—	3	220	90,880	0.41	64
128	292		—	—	5	210	51,688	0.24	70

† = Slices (no info on LUTs), + Length of the non-sparse arbitrary polynomial, * = Weight of the sparse polynomial input

$$latency_{\texttt{poly\_mult}} = W_{SPARSE} \times (3 + \texttt{ceil}(n/\texttt{BW})) + 2$$

Table 2 shows the results for our `poly_mult` module compared with the related work. We note that our sparse polynomial multiplication module performs better in terms of time while utilizing half the Block RAM resources when compared to the existing designs. Table 3 shows results for our `poly_mult` module for the parameter sizes used for HQC hardware design.

Polynomial Addition/Subtraction HQC uses polynomial addition/ subtraction in all of its primitives. Since all addition and subtraction operations happen in $\mathbb{F}_2$, the addition and subtraction could be realized as the same operation. We design two variants of constant-time adders namely `xor_based_adder` and `location_based_adder` that could be attached with our polynomial multiplication module described in Section 2.1. We design our adder modules as an extension for polynomial multiplication because the addition/subtraction always appears with the polynomial multiplication as shown in Algorithm 2, Algorithm 3, and Algorithm 4. The adders operate on contents of block RAM since the polynomials are stored inside the block RAM. Both of the adder module designs do not use any additional block RAM resources, they load the polynomial multiplication output, perform the addition, and write the value back to the same block RAM inside the polynomial.

The `xor_based_adder` design performs addition in a regular $\mathbb{F}_2$ fashion by performing bit-wise `exclusive-OR` operation. The module performs addition sequentially by generating one block RAM address per clock cycle to load inputs from two block RAMs and then performs addition and writes them back to one of the specified block RAMs at the same block RAM address.

Table 3: Time and area information of our `poly_mult` module for different HQC parameter sizes with different performance parameter (`BW`) sizes, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Input Length^{+} (bits)	W^{*}_{sparse}	**Resources**					**F**	**Cycles**	**Time**	**T x A**
		Logic			**Memory**					
		(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
Our poly_mult module (`BW = 32`)										
17,669 (hqc128)	66	139	412	0	189	1	287	36,698	0.13	18
35,851 (hqc192)	100	131	387	0	193	2	257	112,402	0.44	57
57,637 (hqc256)	131	134	397	0	199	2	267	236,457	0.89	119
Our poly_mult module (`BW = 64`)										
17,669 (hqc128)	66	209	620	0	245	2	270	18,482	0.07	14
35,851 (hqc192)	100	219	649	0	249	2	286	56,402	0.20	43
57,637 (hqc256)	131	218	644	0	223	2	283	118,426	0.42	91
Our poly_mult module (`BW = 128`)										
17,669 (hqc128)	66	486	1,439	0	496	4	238	9,374	0.04	19
35,851 (hqc192)	100	488	1,445	0	500	4	240	28,402	0.12	58
57,637 (hqc256)	131	489	1,448	0	474	4	245	59,476	0.24	119

$^{+}$ Length of the non-sparse polynomial, * = Weight of the sparse polynomial input

Table 4: Polynomial addition modules (`xor_based_adder` and `loc_based_adder` with datapath width 128-bits) area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Input Length (bits)	**Resources**					**F**	**Cycles**	**Time**	**T x A**
	Logic			**Memory**					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
`xor_based_adder (BW = 128)`									
17,669	74	143	0	159	0	330	142	0.43	31
35,851	73	142	0	161	0	318	284	0.89	65
57,637	73	142	0	161	0	311	455	1.46	106
`loc_based_adder (BW = 128)`									
17,669	86	160	0	174	0	316	69	0.22	18
35,851	88	161	0	174	0	300	103	0.34	30
57,637	92	161	0	175	0	300	134	0.45	41

The `location_based_adder` is an optimized adder designed to perform addition when one of the input is a sparse vector. This module is mainly designed to perform operations $\mathbf{x}+\mathbf{h}\cdot\mathbf{y}$ from Algorithm 2 and $\mathbf{r}_1+\mathbf{h}\cdot\mathbf{r}_2$ and $\mathbf{s}\cdot\mathbf{r}_2+\mathbf{e}$ from Algorithm 3. In these operations the values of $\mathbf{x}$, $\mathbf{r}_1$, and $\mathbf{e}$ are sparse, fixed-weight vectors so the addition is optimized by only flipping the bits of the other input in the position of one. The `location_based_adder` module takes location of ones from the sparse vector as input and computes the address to load out the part of non-sparse polynomial from the block RAM and flips the bit on the appropriate location and writes it back to the same location. The process is repeated until all locations with ones are covered. Since there are a fixed, and known number of ones in the fixed-weight vector, there is a fixed number of operations and timing does not reveal any sensitive information. Results of our polynomial addition `location_based_adder` module for one performance parameter (width = 128) are shown in Table 4.

Fixed-Weight Vector Generator The fixed-weight vector generator function generates a uniformly random n-bit fixed-weight vector of a specified input

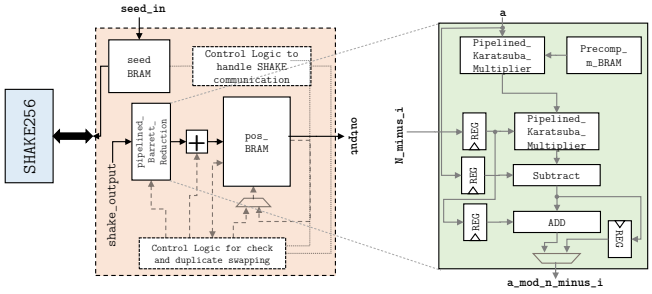


Fig. 1: Hardware design of Constant Weight Word Fixed-Weight vector generator (`fixed_weight_vector_cww`) module.

weight (w). The algorithm for a fixed-weight generation as specified in [1] first generates $24 \times w$ random bits. These random bits are then arranged into w 24-bit integers. These 24-bit integers undergo a threshold check and are rejected if the integer value is beyond the threshold. ($949 \times 17,669$, $467 \times 35,851$, $291 \times 57,637$ for hqc-128, hqc-192 and hqc-256 respectively). After the threshold check, these integers are reduced modulo n. After the threshold check and reduction process, if the weight is not equal to w, then more random bits are drawn from RNG, and the process is repeated until w integers are achieved. After the threshold check and reduction then, a check for duplicates is performed over all the reduced integers. In case any duplicate is found, that integer is discarded, and more random bits are requested drawn from the RNG, which again undergoes threshold check, reduction, and duplicate check. This process is repeated until a uniform fixed-weight vector is generated.

The main pitfall with the fixed-weight vector approach proposed in [1] is that it may show non-constant-time behavior in the rejection sampling process (i.e., the threshold check and duplicate detection as discussed earlier). A timing attack on existing software reference implementation of HQC [1] was performed in [13]. The authors use the information of rejection sampling routine (that is part of fixed-weight generation) being invoked during the deterministic re-encryption process in decapsulation and show that this leaks secret-dependent timing information. The timing of the rejection sampling routine depends upon the given seed. This seed is derived for the encrypt function in encapsulation and decapsulation procedures using the message. The decapsulation operation is dependent on the decoded message and this dependency allows to construct a plaintext distinguisher (described in detail in [13]) which is then used to mount the timing attack.

Although the attack has not been demonstrated on a hardware implementation of HQC yet, we implement two variants of fixed-weight generation to prevent the attack from being possible in hardware. The two variants are Constant Weight Word Fixed-Weight Vector Generation and Fast and Non-Biased Fixed-Weight Vector Generation; discussed in Appendix 1.B due to limited space.

Constant Weight Word (CWW) Fixed Weight Vector Generation: The CWW fixed-weight vector generation variant comes as a fourth-round recommendation

Table 5: Constant Weight Word (CWW) and Fast Non-Biased (FNB) fixed-weight vector module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip. For FNB design (discussed in Appendix 1.B), the w_r parameter is derived from Table 14. The CWW design is fully constant-time so no `ACCEPTABLE_REJECTIONS` parameter is required.

		Resources									
Design	**Weight**	**Logic**			**Memory**		**F**	**Cycles**	**Time**	**T x A**	**Failure**$^+$
	(w_r)	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)		**Prob.**
Constant Weight Word (CWW)											
`hqc128`	75	67	201	4	229	1.0	201	3,062	15.23	1,020	0
`hqc192`	114	71	211	5	245	1.0	200	6,817	34.09	2,420	0
`hqc256`	149	72	216	5	248	1.0	204	11,487	56.31	4,054	0
Fast and Non-Biased Design with `ACCEPTABLE_REJECTIONS` = w_r (discussed in Appendix 1.B)											
`hqc128`	75	106	316	0	124	2.0	223	1,479	6.63	702	2.8×2^{-199}
`hqc192`	114	100	295	0	125	2.0	236	2,226	9.43	1,075	1.1×2^{-280}
`hqc256`	149	107	314	0	192	2.5	242	3,248	13.42	1,435	4.9×2^{-355}

$^+$ = Probability of our design failing to behave constant-time.

Algorithm 1 Constant Weight Word Fixed Weight Vector Generation

Input: N, w, $seed$
Output: w distinct elements in range 0 to $N-1$

1: $rand\_bits \leftarrow prng(input = seed, output\_size = 32 \times w)$
2: **for** $i \leftarrow w-1$ to 0 **do**
3: $\quad pos[i] = i + (rand\_bits[32 + 32 * i - 1 : 32 * i])\%(N - i)$
4: **end for**
5: **for** $j \leftarrow w-1$ to 0 **do**
6: $\quad duplicate\_found \leftarrow 0$
7: $\quad$ **for** $k \leftarrow j+1$ to $w-1$ **do**
8: $\quad\quad$ **if** $pos[j] == pos[k]$ **then**
9: $\quad\quad\quad duplicate\_found \leftarrow 1$
10: $\quad\quad$ **end if**
11: $\quad$ **end for**
12: $\quad$ **if** $duplicate\_found == 1$ **then**
13: $\quad\quad pos[j] = j$
14: $\quad$ **end if**
15: **end for**
16: **return** pos

from the HQC authors ([2]). It was introduced as a fix for the non-constant time behavior of the earlier fixed-weight algorithm (given in [1]) at the cost of small bias. The CWW was originally proposed by Sendrier in (Algorithm 5 of [24]). Shown in Algorithm 1, we rewrite the CWW fixed-weight vector generation algorithm as implemented in our hardware design. The CWW fixed-weight algorithm first generates $32 \times w$ random bits. These random bits are arranged into 32-bit integer array with indices 0 to $w-1$. Each 32-bit integer from the array is then modulo-reduced to `N - ARRAY_INDEX`, and the reduced number is then added with the `ARRAY_INDEX`. After the reduction, a compare and swap is performed, as shown in steps 5-14 of Algorithm 1. This compare and swap step ensures no duplicate elements exist in the final fixed-weight vector.

Our hardware design uses the SHAKE256 module (described in Section 2.1) as the PRNG. The 32-bit interface from our `SHAKE256` module helps us avoid the 32-bit arrangement of random bits (given in steps 2-4 of Algorithm 1). We design

a pipelined Barrett reduction [6] unit to perform the modular reduction (where both the input and modulo value can be changed at runtime, note that in most other design the modulo is fixed at compile time which makes the design of the reduction unit simpler). The operation is shown in step 3 of Algorithm 1. To perform the integer multiplication inside the Barrett reduction, we design karatsuba multiplication [7] unit using the Digital Signal Processing (DSP) units available on the target FPGA. If the DSP resources are unavailable on the target FPGA, we note that our design can naturally be synthesized to use LUTs. We store the reduced values in a dual-port BRAM (`pos_BRAM` shown in Figure 1) of depth w. Once the BRAM is filled, we perform the compare and swap step with the help of the control logic interfaced with the two ports on the BRAM. We note that the pseudorandom number generation, Barrett reduction is performed in constant-time and since the compare and swap procedure is always over a fixed number of memory locations, we achieve a fully constant time hardware implementation for the fixed-weight vector generation process. Although the CWW algorithm ensures the constant time behavior in generating fixed-weight vectors, there is a small bias between the uniform distribution and the algorithm's output. The security analysis performed in [24] for BIKE's parameters [4] shows that this bias has negligible impact on security.

The time and area results for our hardware design are given in Table 5. Because the compare and swap operation requires combinatorial logic between two ports of the dual port BRAM, this becomes the critical path for the design. The compare and scan step takes $w \times (w-1)/2$ clock cycles. Consequently, as shown in the Table 5, as w increases, the number clock cycles also increases.

2.2 Encode and Decode Modules

The encode and decode modules are building blocks of the encrypt and decrypt modules, respectively. We describe the encode and decode modules here, before describing the bigger encrypt and decrypt modules in Section 2.3.

Encode Module As specified in [2], HQC Encode uses concatenation of two codes namely Reed–Muller and Reed–Solomon codes. The hardware design of our `encode` module is shown in Figure 2. The Encode function takes K-bit input and first encodes it with the Reed–Solomon code. The Reed–Solomon encoding process involves systematic encoding using a linear feedback shift register (LFSR) with a feedback connection based on the generator polynomial (shown on page 23, section 2.5.2 of [2]). The Reed–Solomon code generates a n_1-bit output (as given in [2] the value for n_1 is 368, 448, and 720 for `hqc128`, `hqc192`, and `hqc256` respectively). For the Galois field multiplication unit (for the field $\mathbb{F}_2[x]/(x^8+x^4+x^3+x^2+1)$) we design an LFSR-based optimized multiplication unit similar to the one described in [21]. The number of Galois field multiplication units we run in parallel is equal to the degree of the generator polynomial. The outputs from Galois field multipliers are fed in to a LFSR after each cycle. At the end of encoding process the module generates a n_1-bit output.

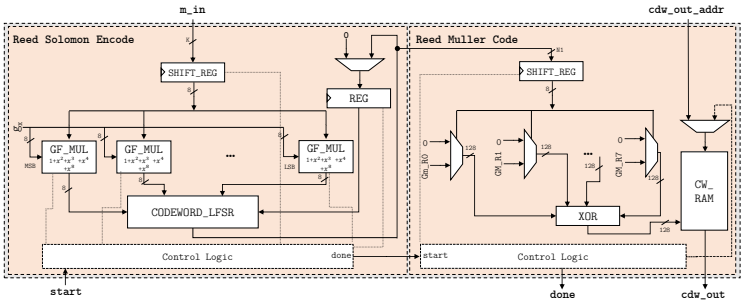


Fig. 2: Hardware design of `encode` module (formed by concatenating two encode functionalities, Reed-Solomon on the left side and Reed-Muller on the right side).

Table 6: `encode` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Design	Resources					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
`hqc128`	280	858	0	922	2	270.34	97	0.36	100
`hqc192`	358	1,011	0	1,088	2	298.32	131	0.44	157
`hqc256`	514	1,503	0	1,689	2	293.51	189	0.64	331
HLS design - {Reed–Solomon Encode + Reed–Muller Encode} [3]									
`hqc128`	—	2,019	0	603	0	—	7,244	47.18	—

The n_1-bit output from Reed–Solomon code is then encoded by Reed–Muller code. The Reed–Muller encoding is achieved by performing vector-matrix multiplication where each byte from input is the vector and the matrix is the generator matrix (**G**) shown in Appendix 1.A. In our design we store the generator matrix rows (each row is of length 128-bits) in ROM and we select the matrix rows based on each input byte. We store the output after multiplying input byte into a block RAM in chunks of 128-bits. Based on the security parameter set the code word output from Reed–Muller code has a multiplicity value (i.e., number of times a code word or in our case number of times each block RAM location is repeated). As per the specification [2], `hqc128` has multiplicity value of 3 and `hqc192` and `hqc256` have multiplicity value of 5. To optimize the storage, we only store one copy of code word, and while accessing the code word we compute the block RAM address in a way that the multiplicity is achieved. The time and area results for our `encode` module targeting Artix 7 board with `xc7a200t-3` FPGA are shown in Table 6.

Decode Module As given in the specification [2], the ciphertext is first decoded with duplicated Reed-Muller code and then with shortened Reed-Solomon code. To decode duplicated Reed-Muller code, the `transformation` module expands and adds multiple code words into expanded code word, and then the Hadamard transformation is applied to the expanded code word. Finally, `Find_Peak` module finds the location of the highest absolute value of the `Hadamard_Transformation` output. Figure 6 describes detailed hardware design of Reed-Muller Decoder.

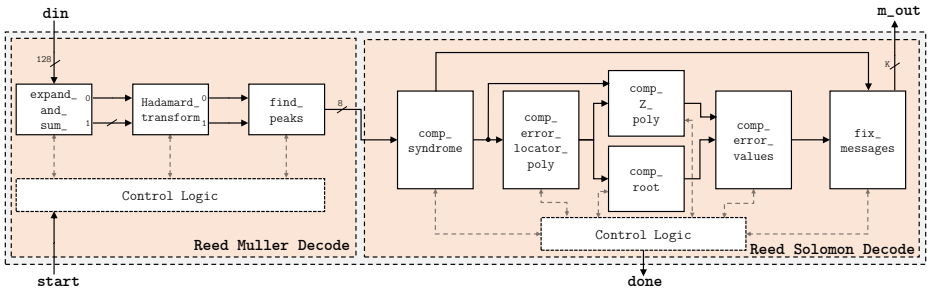


Fig. 3: Hardware design of `decode` module (formed by concatenating two decode functionalities, Reed-Muller on the the left side and Reed-Solomon on the right side).

`expand_and_sum` module collects data inputs into m x 128-bit shift register, then add and shift the last 2-bit lsb of each shift register to produce a pair of data outputs. The data pair is then processed in `hadamard_transformation` module which consist of 7 layers of similar blocks of radix-2 butterfly structure. With the outputs from `hadamard_transformation` coming in pairs, finding peak can be done in parallel inside the `Find_Peak` module and compare the peaks of each to be the final peak. The whole processes then repeated n_1 times to produce n_1 data output to Reed-Solomon Decoder.

To decode Reed-Solomon code, we need to sequentially compute syndromes S_i, coefficients σ_i of error location polynomial $\sigma(x)$, roots of error location polynomial $(\alpha^i)^{-1}$, pre-defined helper polynomial $Z((\alpha^i)^{-1})$, errors e_i, and finally correct the output of decode of Reed-Muller code based on the errors.

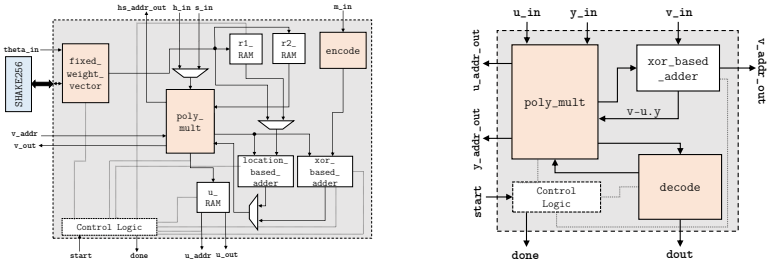
Evaluation Table 6 and Table 7 show time and area results for our `decode` module. Out of the existing other hardware designs [2,3,22] (i.e., HLS and hardware-software codesign), only [3] provides a breakdown of the performance for encode and decode modules. As shown in Table 7 and Table 6 our hardware design outperforms the other designs by a significant margin in all aspects. To the best of our knowledge our implementation of encode and decode is the first hand-optimized hardware implementation of concatenated encode (shortened Reed–Solomon encode + duplicated Reed–Muller encode) and decode (duplicated Reed–Muller decode and shortened Reed–Solomon decode) modules. There are other hand-optimized hardware designs in the literature for Reed–Solomon and Reed–Muller encode and decode (given in [23,5,14]), but their target was not a cryptographic application. Hence, the implementation strategy is highly focused on timing, throughput, and area performance rather than a secure implementation (e.g., constant-time). Consequently, although our design is very efficient, it will not be fair to compare our hardware implementations with them.

2.3 Encrypt and Decrypt Modules

The encrypt and decrypt modules are building blocks of the encapsulation and decapsulation modules, respectively. We describe the encrypt and decrypt modules here, before describing the bigger encapsulation and decapsulation modules later.

Table 7: `decode` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Design	Resources					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(ms)	
`hqc128`	952	2,817	0	3,779	2.5	205	4,611	0.02	19
`hqc192`	1,100	3,257	0	4,727	2.5	212	5,485	0.03	33
`hqc256`	1,243	3,679	0	5,574	2.5	206	9,199	0.04	50
HLS design - {Reed–Muller Decode + Reed–Solomon Decode} [3]									
`hqc128`	—	10,154	0	2,569	3	—	68,619	592.00	—



(a) `encrypt` module. (b) `decrypt` module.

Fig. 4: Hardware design of `encrypt` and `decrypt` modules.

Encrypt Module The `encrypt` module (shown in Algorithm 3) takes public key (**h, s**), message **m**, and seed (θ) and generates a ciphertext (**u,v**) as the output. The hardware design for the `encrypt` module is shown in Figure 4a. We use our CWW fixed-weight vector module (`fixed_weight_vector_cww`) described in Section 2.1.A to generate $\mathbf{r}_1$, $\mathbf{r}_2$, and **e** fixed-weight vectors of weight w_r by expanding `theta_in` and in parallel we run `encode` module (described in Section 2.2). After the generation of $\mathbf{r}_2$ we start the polynomial multiplication of $\mathbf{h}.\mathbf{r}_2$ in parallel to the **e** generation. For polynomial multiplication, we use the `poly_mult` module with `BW = 128` described in Section 2.1. The addition of $\mathbf{r}_1$ in **u** computation and **e** in **v** computation is performed by our `location_based_adder` and addition with **t** is performed by `xor_based_adder` (described in Section 2.1).

From Algorithm 3, we observe that both $\mathbf{h}.\mathbf{r}_2$ and $\mathbf{s}.\mathbf{r}_2$ multiplications can be performed in parallel, consequently we design a `parallel_encrypt` module targeting higher performance where we use two polynomial multiplications in parallel (shown in Figure 8a). We provide a choice of using either `encrypt` or `parallel_encrypt` module as a parameter. Table 8 shows results for both the encrypt hardware implementations targeting Xilinx Artix 7 `xc7a200t` FPGA. We note that the major contributor to the overall time in encrypt operation is due to polynomial multiplication and using two `poly_mult` modules in parallel reduces the overall time by 40-60% across different parameter sets. The area results do not include the `SHAKE256` module as the SHAKE256 is shared among

Table 8: `encrypt` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Design	Resources†					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
`encrypt` module – uses one `poly_mult` module with with `BW = 128`									
`hqc128`	1,230	3,642	4	1,773	10	179	28,217	0.16	194
`hqc192`	1,283	3,797	5	1,966	10	182	79,889	0.44	563
`hqc256`	1,438	4,256	5	2,542	10	192	160,489	0.84	1,202
`parallel_encrypt` module – two `poly_mult` modules with `BW = 128` running in parallel									
`hqc128`	1,734	5,132	4	2,179	12	179	17,202	0.10	173
`hqc192`	1,793	5,308	5	2,376	12	196	46,857	0.24	429
`hqc256`	1,934	5,725	5	2,931	12	196	91,862	0.47	908

† = Given resources does not include the area for `SHAKE256` module.

Table 9: `decrypt` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Design	Resources†					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(us)	
`hqc128`	2,146	6,352	0	5,730	10.5	194	14,198	0.07	150
`hqc192`	2,378	7,038	0	6,787	10.5	187	34,313	0.18	428
`hqc256`	2,886	8,544	0	8,740	13	186	69,356	0.37	1,067

all primitives. Figure 3 shows the hardware block design on our module and Table 7 shows the time and area results for our module.

Decrypt Module The `decrypt` module (shown in Algorithm 4) takes secret key (**x, y**), ciphertext (**u,v**), and generates the message (**m'**). Figure 4b shows our hardware design for `decrypt` module. The module accepts part of the secret key (**y**) as locations with ones (since it is a sparse fixed weight vector). We use our `poly_mult` module with `BW = 128` described in Section 2.1 to compute **u.y** and use `xor_based_adder` module (described in Section 2.1) to compute $\mathbf{v} - \mathbf{u.y}$. We then use the `decode` module (described in Section 2.2) to decode $\mathbf{v} - \mathbf{u.y}$ and retrieve the message. Table 9 shows our hardware implementation results for `decrypt` module targeting Xilinx Artix 7 `xc7a200t` FPGA.

2.4 Key Generation

We now begin to describe the top-level modules, starting with the key generation, followed in later sections with encapsulation and decapsulation.

The key generation (shown in Algorithm 2) takes the secret key seed and public key seed as an input and generates secret key (**x, y**) and public key (**h, s**) respectively as output. Figure 8b shows the hardware design of our `keygen` module. Our `keygen` module assumes that the public key seed and the secret key seed are generated by some other hardware module implementing a true random number generator. We use our CWW fixed-weight vector module (`fixed_weight_vector_cww`) module described in Section 2.1.A to generate (**x, y**) from the secret key seed. **x** and **y** are fixed weight vectors of weight w

Table 10: keygen module area and timing information, data based on synthesis results for Artix 7 board with xc7a200t-3 FPGA chip.

Design	Resources†					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(ms)	
hqc128	809	2,396	4	901	10	179	15,759	0.09	72
hqc192	791	2,342	5	926	10	189	42,106	0.22	177
hqc256	791	2,342	5	942	10	188	82,331	0.44	347
hqc128-perf HLS*[2]	3,900	12,000	0	9,000	3	150	40,000	0.27	1,053
hqc128-comp. HLS*[2]	1,500	4,700	0	2,700	3	129	630,000	4.80	7,200
hqc128-optimized. HLS*[3]	3,921	11,484	0	8,798	6	150	40,427	0.27	1,058
hqc128-pure HLS*[3]	8,359	24,746	0	21,746	7	153	40,427	0.27	2,256

† = Given resources does not include the area for SHAKE256 module, * = Target FGPA is Artix-7 xc7a100t-1

and length n-bits. To optimize the storage, rather than storing full n-bit sparse vector we only output locations of ones. There is also an optional provision to output the full vector as described in Section 2.1. The vector_set_random uses the SHAKE256 module to expand the public key seed and generates **h**. We then use poly_mult module with BW = 128 (described in Section 2.1) to compute (**h.y** and finally use location_based_adder module (described in Section 2.1) to compute **s**. We note that in the Figure 8b only a block RAM for **x** storage (X_RAM) is visible because the $\mathbf{y}, \mathbf{h}, \mathbf{s}$ are stored in the block RAMs which are inside fixed_weight_vector, poly_mult, and location_based_adder modules respectively.

Table 10 shows the results for the keygen module. The area results do not include the SHAKE256 module because it is shared among all other primitives. When we compare our hqc128 keygen design with existing designs from literature, we note that our design runs at least 3× faster than existing hardware designs while utilizing 80% lesser FPGA footprint. We highlight that this improvement is due to our optimized fixed_weight_vector_cww and poly_mult modules discussed in Section 2.1 and Section 2.1 respectively.

2.5 Encapsulation Module

The encapsulate operation (shown in Algorithm 5) takes public key (**h, s**) and message **m** as an input and generates shared secret (K) and ciphertext (**c** = (**u,v**)) and **d**. The hardware design of the encap module is shown in Figure 9a. Our encap module assumes that **m** is generated by some other hardware module implementing a true random number generator and provided as an input to our module. Since the SHAKE256 module is extensively used in encapsulate operation we design a HASH_processor module which handles all the communication with the SHAKE256 module. HASH_processor modules reduces the multiplexing logic of inputs to the SHAKE256 module significantly.

The Hash_processor modules helps in expanding **m** to generate θ. We then use our encrypt module (described in Section 2.3) to encrypt **m** using θ and the public key as inputs and generates ciphertext. After the generation of $\mathbf{r}_1$, $\mathbf{r}_2$, and **e** inside the encrypt module (described in Section 2.3) we then run

Table 11: encap module area and timing information, data based on synthesis results for Artix 7 board with xc7a200t-3 FPGA chip.

Design	Resources†					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(ms)	
our encap module with encrypt									
hqc128	1,400	4,145	4	2,128	13	179	33,438	0.19	262
hqc192	1,445	4,278	5	2,412	15	182	90,346	0.50	716
hqc256	1,625	4,809	5	3,041	15	182	177,154	0.97	1,582
our encap module with parallel_encrypt									
hqc128	1,969	5,828	4	2,531	15	179	22,423	0.13	247
hqc192	2,174	6,434	5	2,821	17	196	57,314	0.29	636
hqc256	2,330	6,898	5	3,417	17	196	108,527	0.55	1,292
hqc128 perf HLS* [2]	5,500	16,000	0	13,000	5	151	89,000	0.59	3,245
hqc128 comp. HLS* [2]	2,100	6,400	0	4,100	5	127	1,500,000	12.00	25,200
hqc128 optimized HLS* [3]	5,575	16,487	0	13,390	10	152	89,110	0.59	3,289
hqc128 pure HLS* [3]	9,955	29,496	0	26,333	11	148	89,131	0.59	5,873

† = Given resources does not include the area for SHAKE256 module, * = Target FGPA is Artix-7 xc7a100t-1

HASH_processor module in parallel to encrypt module to generate $\mathbf{d}$. After the encryption of $\mathbf{m}$ we then use the HASH_processor to compute $\mathcal{K}(\mathbf{m}, \mathbf{c})$ to generate the shared secret K. Our design is constant-time since all the underlying modules are constant-time and the control logic from the encap module does not depend on any secret input. Table 11 shows the results for the encap module with our encrypt and parallel_encrypt. The area results do not include the SHAKE256 module because it is shared among all other primitives. We note that our hqc128 encap with parallel_encrypt design runs at least 4.5× faster than existing hardware designs from the literature while using 64% lesser FPGA footprint. Hence, achieving the best Time-Area product. We highlight that this improvement comes mainly from our optimized encode, and poly_mult hardware designs discussed in Section 2.2, and Section 2.1 respectively.

2.6 Decapsulation Module

The decapsulate operation (shown in Algorithm 6) takes secret key ($\mathbf{x}$, $\mathbf{y}$), public key ($\mathbf{h}$, $\mathbf{s}$), ciphertext ($\mathbf{c} = (\mathbf{u}, \mathbf{v})$), $\mathbf{d}$ as an input and generates shared secret (K). Figure 9b shows hardware design the decap module. We use our decrypt module (described in Section 2.3) to decrypt the input ciphertext using secret key ($\mathbf{y}$) and generate the $\mathbf{m}'$. We then use encap module to perform re-encryption of $\mathbf{m}'$ and generate $\mathbf{u}'$, $\mathbf{v}'$ and $\mathbf{d}'$. We then pause the encap module to verify the $\mathbf{u}'$, $\mathbf{v}'$ and $\mathbf{d}'$ against $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{d}$. After the verification we set a signal (optional port mprime_fail) if the verification fails. Irrespective of verification result we still continue with the generation of the shared secret K to maintain the constant-time behavior. Table 12 shows the results for the decap module using encrypt and parallel _encrypt (for performing the rencryption). The area results do not include the SHAKE256 module because it is shared among all the primitives. When we compare our hqc128 decap with parallel_encrypt design with the existing designs from literature, we note that our design runs at least 5.7× faster

Table 12: `decap` module area and timing information, data based on synthesis results for Artix 7 board with `xc7a200t-3` FPGA chip.

Design	Resources†					F	Cycles	Time	T x A
	Logic			Memory					
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(cyc.)	(ms)	
our `decap` module with `encrypt`									
`hqc128`	3,035	8,984	4	6,596	20	192	48,212	0.25	758
`hqc192`	3,368	9,969	5	7,911	22	186	125,805	0.68	2,290
`hqc256`	3,693	10,931	5	9,424	22	186	248,338	1.33	4,911
our `decap` module with `parallel_encrypt`									
`hqc128`	3,702	10,959	4	7,003	22	179	37,197	0.21	777
`hqc192`	4,025	11,915	5	8,320	24	186	92,773	0.50	2,012
`hqc256`	4,347	12,868	5	9,794	24	186	179,711	0.97	4,216
`hqc128 perf HLS*`[2]	6,200	19,000	0	15,000	9.0	152	190,000	1.20	7,440
`hqc128 comp. HLS*`[2]	2,700	7,700	0	5,600	10.5	130	2,100,000	16.00	43,200
`hqc128 optimized HLS*`[3]	6,223	18,739	0	15,243	18.0	152	193,082	1.27	7,903
`hqc128 pure HLS*`[3]	8,434	24,898	0	21,680	18.0	150	193,004	1.27	10,711

† = Given resources does not include the area for `SHAKE256` module, * = Target FGPA is Artix-7 `xc7a100t-1`

while using 40% lower FPGA footprint. Hence, achieving the best Time Area product. We highlight that this improvement comes mainly from our optimized `decode`, `encode`, and `poly_mult` designs discussed in Section 2.2, Section 2.2, and Section 2.1 repectively.

3 HQC Joint Design and Related Work

In this section, we present our joint hardware design of a HQC combining our `keygen`, `encap`, and `decap` modules (described in Section 2.4, Section 2.5, and Section 2.6 respectively) into one overall design. Following that we compare our joint design with other HQC combined designs from the literature in Section 3.2. In addition to that, we also conduct a comprehensive literature survey focusing on full hardware designs of the other three fourth-round public-key encryption and key-establishment algorithms in NIST's standardization process: BIKE, Classic McEliece, and SIKE. We also include the CRYSTALS-Kyber, a public-key encryption and key-establishment algorithm selected for standardization at the end of the prior third round. Due to limited space we discuss this part in Appendix 1.C.

3.1 HQC Joint Design

In this work, we present two designs, *Balanced* and *HighSpeed*. The main difference between our *Balanced* and *HighSpeed* designs is that our *Balanced* design uses the regular `encrypt` module (shown in Figure 4a), and our *HighSpeed* design uses the `parallel_encrypt` module (shown in Figure 8a) for performing the encryption and re-encryption operations in encapsulation and decapsulation. The In order to build a resource-efficient yet performant joint design, we start by identifying the common sub-modules among the three `keygen`, `encap`, and `decap` by using `hqc128` parameter set as an example:

SHAKE256: The `SHAKE256` module is used in all the primitives (`keygen`, `encap`, and `decap`) in HQC. As shown in Table 1, the `SHAKE256` with `parallel_slices = 32` has a high area utilization. Consequently, we share one `SHAKE256` module among all the primitives.

Polynomial Multiplication: The `poly_mult` module is also common among all the primitives. Table 3 shows the area utilization of the `poly_mult` module. In our *Balanced* design, we use only one `poly_mult` module which takes up 60%, 35% and 16% of area resources in overall `keygen`, `encap`, and `decap` modules respectively. And in our *HighSpeed* design, we use two `poly_mult` modules for faster Encrypt and Re-encrypt operations as described in Section 2.5 and Section 2.6 this takes up 50% and 26% of overall area resources in `encap`, and `decap` modules respectively.

Encapsulation: As specified in Section 2.6, we use the `encap` module (described in Section 2.5) inside `decap` module to perform re-encryption and hash computation. This `encap` module takes up 46% of the overall `decap` resources in the *Balanced* design and 53% in the *HighSpeed* design. Consequently, sharing one `encap` module to perform both encapsulation and decapsulation would save a significant amount of area.

In order to save the resource overhead due to the duplication of modules, we decide to share the aforementioned modules between the three primitives in our joint design. To differentiate between different operations, we provide a 2-bit port in the interface, which helps in choosing the operation between Key Generation, Encapsulation, and Decapsulation. The results for our combined *Balanced* and *HighSpeed* implementations are shown in Table 13 in comparison with the most recent related work. Our results are generated targeting the Artix 7 (`xc7a100t-csg324-3`) FPGA. This is the same target FPGA family type as used in the related works [22,3,2]. Our data is from synthesis and implementation reports, while data for the other related works are from the cited papers.

3.2 Evaluation and Comparison to Existing HQC Hardware Designs

Previously, a hardware design for HQC has been generated using high-level synthesis (HLS) [2], and code targeting Artix-7 is available online.[4] The generated code can obtain the performance numbers: 0.3ms for key generation, 0.6ms for encapsulation, and 1.2ms for decapsulation, the times correspond to the *HighSpeed* implementation of the lowest security level. Authors also provide *LightWeight* version for the lowest security level, but did not provide hardware designs for other security levels. A different HLS-based design with better results has been presented in [3]. This HLS design can achieve the performance of: 0.27 ms for key generation, 0.59 ms for encapsulation, and 1.27 ms for decapsulation with their *HighSpeed* version. Apart from the HLS designs, a recent hardware design [22] presented a hardware-software codesign approach and reports better performance numbers than that of *LightWeight* versions of both the HLS designs. Note, however, that there is area overhead of the CPU core. The HLS

[4] https://pqc-hqc.org/implementation.html

Table 13: Comparison of our HQC hardware design with the related work.

Design	Resources					F	Encap		Decap		KeyGen	
	Logic			Memory								
	(Slices)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(Mcyc.)	(ms)	(Mcyc.)	(ms)	(Mcyc.)	(ms)
Security Level 1 — Classical 128-bit Security												
HQC – Our Work, HDL design, Artix 7 (xc7a100t)												
Balanced	4,684	13,865	8	6,897	22	164	0.03	0.20	0.05	0.29	0.02	0.10
HighSpeed	5,246	15,214	8	7,293	24	178	0.02	0.13	0.04	0.21	0.02	0.09
HQC – [22], HW/SW codesign, Artix 7 (xc7a100t)												
HW/SW	—	8,000	0	2,400	3	100	0.13	1.3	0.56	5.6	0.06	0.56
HQC – [3], HLS design, Artix 7 (xc7a100t)												
LightWeight	—	8,876	0	6,405	28.0	132	1.48	11.85	2.15	17.21	0.62	5.01
HighSpeed	—	20,169	0	16,374	25	148	0.09	0.59	0.19	1.27	0.04	0.27
HQC – [2], HLS design, Artix 7 (xc7a100t)												
LightWeight	3,100	8,900	0	6,400	14.0	132	1.50	12.00	2.10	16.00	0.63	4.80
HighSpeed	6,600	20,000	0	16,000	12.5	148	0.09	0.60	0.19	1.20	0.04	0.30

HW/SW = Hardware-Software CoDesign, FF = flip-flop, F = $F_{\max}$, BR = BRAM

designs and hardware-software codesign only provide the lowest security level version. Meanwhile, both *Balanced* and *HighSpeed* variants of our design are faster for all three operations when compared to all existing designs. We also achieve the best time area product, and cover all three security levels.

4 Conclusion

This work presented two performance-targeted and constant-time hardware designs of the HQC KEM. This work presented first, hand-optimized design of HQC key generation, encapsulation, and decapsulation written in Verilog targeting FPGAs and provides compile-time parameters to switch between all security levels and performances. This work also presented a memory-optimized Polynomial Multiplication module and a SHAKE256 module, which runs two times faster when compared to the existing work. This work also presented the first hardware implementation of two variants of constant-time solutions for the fixed-weight vector generation process. Our HQC design currently outperforms the other existing hardware designs for HQC. As this work showed, code-based designs such as HQC can be realized very efficiently in optimized hardware.

Acknowledgement

We would like to thank the reviewers for the valuable feedback and Dr. Cuauhtemoc Mancillas López for constructive comments and shepherding our article. We would like to thank Dr. Victor Mateu and Dr. Carlos Aguilar Melchor for helpful discussions. The work was supported in part by research grant from Technology Innovation Institute.

References

1. Aguilar Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zémor, G., Bos, J.: HQC. Tech. rep., National Institute of Standards and Technology (2020), available at https://pqc-hqc.org/doc/hqc-specification_2021-06-06.pdf
2. Aguilar Melchor, C., Aragon, N., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Persichetti, E., Zémor, G., Bos, J.: HQC. Tech. rep., National Institute of Standards and Technology (2023), available at http://pqc-hqc.org/doc/hqc-specification_2023-04-30.pdf
3. Aguilar-Melchor, C., Deneuville, J.C., Dion, A., Howe, J., Malmain, R., Migliore, V., Nawan, M., Nawaz, K.: Towards automating cryptographic hardware implementations: a case study of hqc. Cryptology ePrint Archive, Paper 2022/1425 (2022), https://eprint.iacr.org/2022/1425, https://eprint.iacr.org/2022/1425
4. Aragon, N., Barreto, P., Bettaieb, S., Bidoux, L., Blazy, O., Deneuville, J.C., Gaborit, P., Gueron, S., Guneysu, T., Aguilar Melchor, C., Misoczki, R., Persichetti, E., Sendrier, N., Tillich, J.P., Zémor, G., Vasseur, V., Ghosh, S.: BIKE. Tech. rep., National Institute of Standards and Technology (2020), available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-3-submissions
5. Azad, A.A., Shahed, I.: A compact and fast fpga based implementation of encoding and decoding algorithm using reed solomon codes. International Journal of Future Computer and Communication pp. 31–35 (2014)
6. Barrett, P.: Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In: Odlyzko, A.M. (ed.) Advances in Cryptology — CRYPTO' 86. pp. 311–323. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)
7. Bernstein, D.D.: Fast multiplication and its applications (2008)
8. Chen, P., Chou, T., Deshpande, S., Lahr, N., Niederhagen, R., Szefer, J., Wang, W.: Complete and improved FPGA implementation of Classic Mceliece. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(3), 71–113 (2022). https://doi.org/10.46586/tches.v2022.i3.71-113, https://doi.org/10.46586/tches.v2022.i3.71-113
9. Dang, V.B., Mohajerani, K., Gaj, K.: High-speed hardware architectures and fpga benchmarking of crystals-kyber, ntru, and saber. Cryptology ePrint Archive, Paper 2021/1508 (2021), https://eprint.iacr.org/2021/1508, https://eprint.iacr.org/2021/1508
10. Deshpande, S., del Pozo, S.M., Mateu, V., Manzano, M., Aaraj, N., Szefer, J.: Modular inverse for integers using fast constant time gcd algorithm and its applications. In: Proceedings of the International Conference on Field Programmable Logic and Applications. FPL (August 2021)
11. Galimberti, A., Galli, D., Montanaro, G., Fornaciari, W., Zoni, D.: On the use of hardware accelerators in qc-mdpc code-based cryptography. In: Proceedings of the 19th ACM International Conference on Computing Frontiers. p. 193–194. CF '22, Association for Computing Machinery, New York, NY, USA (2022). https://doi.org/10.1145/3528416.3530243, https://doi.org/10.1145/3528416.3530243
12. Gigliotti, P.: Implementing barrel shifters using multipliers. Tech. Rep. XAPP195, Xilinx (2004), https://www.xilinx.com/support/documentation/application_notes/xapp195.pdf

13. Guo, Q., Hlauschek, C., Johansson, T., Lahr, N., Nilsson, A., Schröder, R.L.: Don't reject this: Key-recovery timing attacks due to rejection-sampling in hqc and bike. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022, Issue 3**, 223–263 (2022). https://doi.org/10.46586/tches.v2022.i3.223-263, https://tches.iacr.org/index.php/TCHES/article/view/9700
14. Hashemipour-Nazari, M., Goossens, K., Balatsoukas-Stimming, A.: Hardware implementation of iterative projection-aggregation decoding of reed-muller codes. In: ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). pp. 8293–8297 (2021). https://doi.org/10.1109/ICASSP39728.2021.9414655
15. Hu, J., Wang, W., Cheung, R.C., Wang, H.: Optimized polynomial multiplier over commutative rings on fpgas: A case study on bike. In: 2019 International Conference on Field-Programmable Technology (ICFPT). pp. 231–234 (2019). https://doi.org/10.1109/ICFPT47387.2019.00035
16. Jati, A., Gupta, N., Chattopadhyay, A., Sanadhya, S.K.: A configurable CRYSTALS-Kyber hardware implementation with side-channel protection. Cryptology ePrint Archive (2021)
17. Massolino, P.M.C., Longa, P., Renes, J., Batina, L.: A compact and scalable hardware/software co-design of SIKE. IACR Transactions on Cryptographic Hardware and Embedded Systems **2020**(2), 245–271 (Mar 2020). https://doi.org/10.13154/tches.v2020.i2.245-271, https://tches.iacr.org/index.php/TCHES/article/view/8551
18. Montanaro, G., Galimberti, A., Colizzi, E., Zoni, D.: Hardware-software co-design of bike with hls-generated accelerators (2022). https://doi.org/10.48550/ARXIV.2209.03830, https://arxiv.org/abs/2209.03830
19. Richter-Brockmann, J., Chen, M.S., Ghosh, S., Güneysu, T.: Racing bike: Improved polynomial multiplication and inversion in hardware. IACR Transactions on Cryptographic Hardware and Embedded Systems **2022**(1), 557–588 (Nov 2021). https://doi.org/10.46586/tches.v2022.i1.557-588, https://tches.iacr.org/index.php/TCHES/article/view/9307
20. Richter-Brockmann, J., Mono, J., Guneysu, T.: Folding bike: Scalable hardware implementation for reconfigurable devices. IEEE Transactions on Computers **71**(5), 1204–1215 (2022). https://doi.org/10.1109/TC.2021.3078294
21. Sandoval-Ruiz, C.: Vhdl optimized model of a multiplier in finite fields. Ingenieria y Universidad **21**(2), 195–212 (Jun 2017). https://doi.org/10.11144/Javeriana.iyu21-2.vhdl, https://revistas.javeriana.edu.co/index.php/iyu/article/view/195
22. Schöffel, M., Feldmann, J., Wehn, N.: Code-based cryptography in iot: A HW/SW co-design of HQC. CoRR **abs/2301.04888** (2023). https://doi.org/10.48550/arXiv.2301.04888, https://doi.org/10.48550/arXiv.2301.04888
23. Scholl, S., Wehn, N.: Hardware implementation of a reed-solomon soft decoder based on information set decoding. In: 2014 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1–6 (2014). https://doi.org/10.7873/DATE.2014.222
24. Sendrier, N.: Secure sampling of constant-weight words – application to bike. Cryptology ePrint Archive, Paper 2021/1631 (2021), https://eprint.iacr.org/2021/1631, https://eprint.iacr.org/2021/1631
25. Wang, W., Tian, S., Jungk, B., Bindel, N., Longa, P., Szefer, J.: Parameterized hardware accelerators for lattice-based cryptography and their application to the hw/sw co-design of qTESLA. IACR Transactions on

Cryptographic Hardware and Embedded Systems **2020**(3), 269–306 (Jun 2020). https://doi.org/10.13154/tches.v2020.i3.269-306, https://tches.iacr.org/index.php/TCHES/article/view/8591
26. Xing, Y., Li, S.: A compact hardware implementation of cca-secure key exchange mechanism crystals-kyber on fpga. IACR Transactions on Cryptographic Hardware and Embedded Systems **2021**(2), 328–356 (Feb 2021). https://doi.org/10.46586/tches.v2021.i2.328-356, https://tches.iacr.org/index.php/TCHES/article/view/8797
27. Zhu, Y., Zhu, W., Chen, C., Zhu, M., Li, Z., Wei, S., Liu, L.: Compact gf(2) systemizer and optimized constant-time hardware sorters for key generation in classic mceliece. Cryptology ePrint Archive, Paper 2022/1277 (2022), https://eprint.iacr.org/2022/1277, https://eprint.iacr.org/2022/1277

Appendix 1.A Preliminaries

In this section, we briefly introduce HQC. We first introduce notations used in this paper. Then HQC public key encryption (PKE) and key encapsulation mechanism (KEM) algorithms are described. We refer to the specification of HQC [2] for more detailed information.

1.A.1 Notation

In this paper, we denote $\mathbb{F}_2$ the binary finite field, and $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ the quotient ring on which vectors and operations of HQC are defined. For any field or ring, $\mathbb{F}_2^l$ or $\mathcal{R}^l$ denotes the field or ring of l dimensional vectors over $\mathbb{F}_2$ or $\mathcal{R}$. An element $\mathbf{x} \in \mathcal{R}$ can be represented as either a vector $\mathbf{x} \coloneqq (x_0, x_1, \ldots, x_{n-1})$ or a polynomial $\mathbf{x} \coloneqq \sum_{i=0}^{n-1} x_i X^i$. The Hamming weight of $\mathbf{x}$ is defined as $\sum_{i=0}^{n-1} x_i$, i.e., the number of non-zero coefficients in $\mathbf{x}$. $\mathbf{x} \leftarrow \mathcal{R}$ denotes $\mathbf{x}$ is chosen uniformly random from $\mathcal{R}$, while $\mathbf{x} \xleftarrow{w} \mathcal{R}$ denotes $\mathbf{x}$ is randomly chosen from $\mathcal{R}$ and the Hamming weight of $\mathbf{x}$ is w. Optionally, a random seed can be specified for the sampling process, and the sampling process with random seed θ is denoted as $\mathbf{x} \xleftarrow{w,\theta} \mathcal{R}$. For $\mathbf{x}, \mathbf{y} \in \mathcal{R}$, the addition $\mathbf{a} = \mathbf{x} + \mathbf{y} \in \mathcal{R}$ and is defined as $a_i = x_i + y_i \mod 2$ or $a_i = x_i \text{ xor } y_i$ for $i = 0, \ldots, n-1$. The multiplication $\mathbf{a} = \mathbf{x} \cdot \mathbf{y} \in \mathcal{R}$ and is defined as $a_k = \sum_{i+j \equiv k \mod n} x_i \cdot y_j \mod 2$ for $k = 0, \ldots, n-1$. Encode$(\cdot)$ and Decode$(\cdot)$ are the encode and decode function of concatenated Reed–Muller and Reed–Solomon codes [2]. $\mathcal{G}(\cdot), \mathcal{H}(\cdot), \mathcal{K}(\cdot)$ are hash functions with domain separation bytes 3, 4, 5 respectively. Parameters n, w, w_r depend on the security level, which can be found on Table 14.

1.A.2 HQC PKE and KEM Schemes

Encode of duplicated Reed–Muller code. Encode of duplicated Reed–Muller code is to directly perform a matrix vector multiplication. The generator

matrix is shown below (note that numbers are big endian and in hexadecimal):

$$\mathbf{G} = \begin{pmatrix} \texttt{aaaaaaaa aaaaaaaa aaaaaaaa aaaaaaaa} \\ \texttt{cccccccc cccccccc cccccccc cccccccc} \\ \texttt{f0f0f0f0 f0f0f0f0 f0f0f0f0 f0f0f0f0} \\ \texttt{ff00ff00 ff00ff00 ff00ff00 ff00ff00} \\ \texttt{ffff0000 ffff0000 ffff0000 ffff0000} \\ \texttt{00000000 ffffffff 00000000 ffffffff} \\ \texttt{00000000 00000000 ffffffff ffffffff} \\ \texttt{ffffffff ffffffff ffffffff ffffffff} \end{pmatrix}$$

If the message is $\mathbf{m} = (m_0, \ldots, m_7) \in \mathbb{F}_{2^8}$, then $\mathbf{c} = \mathbf{mG}$, and the codeword is given by duplicating $\mathbf{c}$ 3 or 5 times, depending on the security level.

Algorithm 2 HQC.PKE.KeyGen() and HQC.KEM.KeyGen()

$\mathbf{h} \leftarrow \mathcal{R}$
$(\mathbf{x}, \mathbf{y}) \xleftarrow{w} \mathcal{R}^2$
$\mathbf{s} \coloneqq \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$
return (pk $\coloneqq (\mathbf{h}, \mathbf{s})$, sk $\coloneqq (\mathbf{x}, \mathbf{y})$)

Algorithm 3 HQC.PKE.Encrypt(pk $= (\mathbf{h}, \mathbf{s})$, $\mathbf{m}$, θ)

$(\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}) \xleftarrow{\omega_r, \theta} \mathcal{R}^3$
$\mathbf{u} \coloneqq \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
$\mathbf{t} \coloneqq \text{Encode}(\mathbf{m})$
$\mathbf{v} \coloneqq \mathbf{t} + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$
return $\mathbf{c} \coloneqq (\mathbf{u}, \mathbf{v})$

Algorithm 4 HQC.PKE.Decrypt(sk $= (\mathbf{x}, \mathbf{y})$, $\mathbf{c} = (\mathbf{u}, \mathbf{v})$)

$\mathbf{m}' \coloneqq \text{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$
return $\mathbf{m}'$

Algorithm 5 HQC.KEM.Encapsulate(pk $= (\mathbf{h}, \mathbf{s})$)

$\mathbf{m} \leftarrow \mathbb{F}_2^k$
salt $\leftarrow \mathbb{F}_2^{128}$
$\theta \coloneqq \mathcal{G}(\mathbf{m}||pk||salt)$
$\mathbf{c} \coloneqq (\mathbf{u}, \mathbf{v}) = \text{HQC.PKE.Encrypt}(\text{pk}, \mathbf{m}, \theta)$
$K \coloneqq \mathcal{K}(\mathbf{m}, \mathbf{c})$
$\mathbf{d} \coloneqq \mathcal{H}(\mathbf{m})$
return $(K,$ $(\mathbf{c}, \mathbf{d},$ salt$))$

Algorithm 6 HQC.KEM.Decapsulate(sk $= (\mathbf{x}, \mathbf{y})$, $\mathbf{c}$, $\mathbf{d}$, salt)

$\mathbf{m}' \coloneqq \text{HQC.PKE.Decrypt}(\text{sk}, \mathbf{c})$
$\theta' \coloneqq \mathcal{G}(\mathbf{m}'||pk||salt)$
$\mathbf{c}' \coloneqq (\mathbf{u}', \mathbf{v}') = \text{HQC.PKE.Encrypt}(\text{pk}, \mathbf{m}', \theta')$
$\mathbf{d}' \coloneqq \mathcal{H}(\mathbf{m}')$
$K' \coloneqq \mathcal{K}(\mathbf{m}', \mathbf{c})$
if $\mathbf{c} \neq \mathbf{c}'$ or $\mathbf{d} \neq \mathbf{d}'$ **then**
    return $(K',$ $0)$
else
    return $(K',$ $1)$
end if

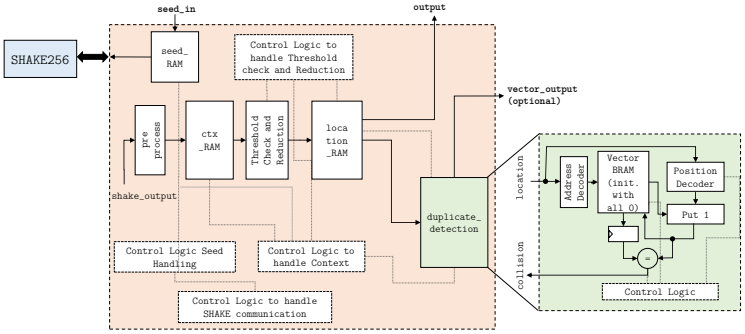


Fig. 5: Hardware design of Fast and Non-Biased (FNB) fixed-weight vector generation (`fixed_weight_vector_fnb`) module.

Table 14: Parameter sets for HQC. n is the length of the vector (polynomial). n_1 is the length of the Reed–Solomon code. n_2 is the length of the Reed–Muller code. w is the weight of vectors $\mathbf{x}, \mathbf{y}$. w_r is the weight of vectors $\mathbf{r}_1, \mathbf{r}_2, \mathbf{e}$. $[n, k, d]$ of Reed–Solomon and Reed–Muller codes are shown in the last two columns, and they are the length, the dimension, and the minimum distance of the code. In HQC, shortened Reed–Solomon code and duplicated Reed–Muller code are used. The multiplicity for duplicated Reed–Muller code is 3, 5, 5 for hqc-128, hqc-192, hqc-256.

Instance	n	w	w_r	**security**	$p_{\mathbf{fail}}$	**Reed–Solomon**	**Reed–Muller**
hqc128	17,669	66	75	128	$< 2^{-128}$	[46, 16, 15]	[384, 8, 192]
hqc192	35,851	100	114	192	$< 2^{-192}$	[56, 24, 16]	[640, 8, 320]
hqc256	57,637	131	149	256	$< 2^{-256}$	[90, 32, 29]	[640, 8, 320]

Appendix 1.B Fast and Non-Biased (FNB) Fixed-Weight Vector Generation:

Although the CWW design is constant in time, it does have a small bias. As an alternative, we propose a new FNB fixed-weight vector generation design which is based on fixed-weight vector generation technique given in [1]. Our FNB fixed-weight generation module can be parametrized to create design with an arbitrarily small probability of timing attack being possible. In our hardware module, have a parameter `ACCEPTABLE_REJECTIONS`, which can be used to specify how many indices could be rejected in either rejection sampling or in duplicated detection and still, the design will behave constant time. The parameter (`ACCEPTABLE_REJECTIONS`) can be set based on user's target failure probability. If the actual failures are within the failure probability set by the selected parameter value, then the timing side channel given in [13] is not possible.

The hardware design of our FNB fixed-weight vector generation module fixed_weight_vector_fnb is shown in Figure 5. We use `SHAKE256` module described in Section 2.1 to expand 320-bit seed to a $24 \times w$-bit string. Since the `SHAKE256`

module has 32-bit interface, the seed is loaded in 32-bit chunks, and the seed is stored in `seed_RAM` as shown in the Figure 5. The 32-bit chunk from SHAKE256 is broken into 24-bit integer by `preprocess` unit and stored in the `ctx_RAM` then threshold check and reduction are performed. For the reduction, we use Barrett reduction [6]. Unlike the variable Barrett reduction discussed in Section 2.1.A, this specific Barrett reduction is optimized as we always reduce the inputs to a specific fixed value (n). After the reduction, the integer values are stored in the `locations_RAM`. Once the `locations_RAM` is filled, the `duplicate_detection` module is triggered. The `duplicate_detection` module helps detect potential duplicates values in the `location_RAM` by traversing through all address locations of `location_RAM` and updating the value stored in a dual-ported RAM `VectorBRAM`. While the `duplicate_detection` module checks for duplicates, the `SHAKE256` module generates the next $24 \times w$-bit string to tackle any potential duplicates and stores them in the `ctx_RAM`. This way, we can mask any clock cycles taken for seed expansion.

Our hardware design uses a PRNG to generate the uniformly random bits required for the fixed weight vector generation from an input seed of length 320-bits. Our hardware design includes this PRNG in the form of SHAKE256 and assumes that the seed will be initialized by some other hardware module implementing a true random number generator.

Our FNB fixed-weight generation module can be parametrized to create design with an arbitrarily small probability of timing attack being possible. In our hardware module, have a parameter name is `ACCEPTABLE_REJECTIONS`, which can be used to specify how many indices could be rejected, and still, the design will behave constant time (at the cost of extra area for more storage and extra cycles). The extra area is needed because we generate additional (based on parameter value) uniformly random bits in advance and store them in the `ctx_RAM` (shown in Figure 5). The extra clock cycles are needed because even after we found the required number of indices that are under the threshold value, we still go over all the `ctx_RAM` locations. And for the duplicate detection logic inside `duplicate_detection` module (shown in Figure 5), the control logic is programmed to take the same cycles in both cases of duplicate being detected or not. The parameter (`ACCEPTABLE_REJECTIONS`) can be set based on user's target failure probability. If the actual failures are within the failure probability set by the selected parameter value, then the timing side channel given in [13] is not possible.

Table 5 shows the comparison of our new FNB design to the CCW design. The area results shown in Table 5 exclude `SHAKE256` module as the SHAKE256 is shared among all primitives. The reported frequency in Although the CWW algorithm ensures the constant time behavior in generating fixed-weight vectors, there is a small bias between the uniform distribution and the algorithm's output. Meanwhile, for the new FNB algorithm, there is no bias. Further, FNB is faster than CWW, and the time-area product is better. These benefits come at the cost of extremely small probabilities that the design is not constant time, but only if it happens that there are more rejections than w_r. Table 5 shows that

Table 15: Comparison of the time and area of state-of-the-art hardware implementations of other (NIST PQC competition) round 4 KEM candidates.

Design	Resources					F	Encap		Decap		KeyGen	
	Logic		Memory									
	(SLICES)	(LUT)	(DSP)	(FF)	(BR)	(MHz)	(Mcyc.)	(ms)	(Mcyc.)	(ms)	(Mcyc.)	(ms)
Security Level 1 — Classical 128-bit Security												
HQC – Our Work, HDL design, Artix 7 (xc7a200t)												
BAL	4,560	13,481	8	6,897	22	164	0.03	0.20	0.05	0.29	0.02	0.10
HS	5,133	15,195	8	7,293	24	178	0.02	0.13	0.04	0.21	0.02	0.09
BIKE – [20], HDL design, Artix 7 (xc7a35t)												
LW	4,078	12,868	7	5,354	17.0	121	0.20	1.2	1.62	13.3	2.67	21.9
HS	15,187	52,967	13	7,035	49.0	96	0.01	0.1	0.19	1.9	0.26	2.7
BIKE – [19], HDL design, Artix 7 (xc7a200t)												
LW	3,777	12,319	7	3,896	9.0	121	0.05	0.4	0.84	6.89	0.46	3.8
TO	5,617	19,607	9	5,008	17.0	100	0.03	0.3	0.42	4.2	0.18	1.9
HS	7,332	25,549	13	5,462	34.0	113	0.01	0.1	0.21	1.9	0.19	1.7
Classic McEliece – [8], HDL design, Artix 7 (xc7a200t)												
LW	—	23,890	5	45,658	138.5	112	0.13	1.1	0.17	1.5	8.88	79.2
HS	—	40,018	4	61,881	177.5	113	0.03	0.3	0.10	0.9	0.97	8.6
SIKE – [17], HDL design, Artix 7 (xc7a100t)												
LW	3,415	—	57	7,202	21	145	—	25.6	—	27.2	—	15.1
HS	7,408	—	162	11,661	37	109	—	15.3	—	16.3	—	9.1
Kyber – [16], HDL design, (xc7a35t-2)												
CB	—	5,269	2	2,422	6	—	0.67	2.67	0.73	2.93	0.69	2.75
RB	—	7,151	2	2,422	5	—	0.03	0.10	0.03	0.12	0.04	0.15
Kyber – [9], HDL design, (xc7a200t)												
HS	—	9,457	4	8,543	4.5	220	0.003	0.01	0.004	0.02	0.002	0.01
Kyber – [26], HDL design, (xc7a12t-1)												
BAL	2,126	7,412	2	4,644	3	161	0.005	0.23	0.006	0.04	0.003	0.02

CB = CoProcessorBased, *RB* = RoundBased, *LW*=LightWeight, *HS*=HighSpeed, *TO*=TradeOff, *BAL*=Balanced, FF = flip-flop, F = $F_{\max}$, BR = BRAM

that the probability of non-constant time behavior for FNB can be 2^{-200} or even smaller. To compute the failure probability (given in Table 5) for each parameter set, we take into account both threshold check failure and duplicate detection probabilities for the respective parameter sets.

Appendix 1.C Comparison to Hardware Designs for Other Round 4 Algorithms

We also provide Table 15 where we tabulate latest hardware implementations of all other post-quantum cryptographic algorithm hardware implementations from the fourth round of NIST's standardization process, plus the to-be standardized Kyber algorithm. We focus on comparison of the hardware designs for lowest level of security, Level 1, as all publications give clear time and area numbers. Majority of related work provides hardware designs for more than the lowest security level, but the timing and area numbers are not clearly broken down in the respective publications, so we focus only on comparing among the lowest security level designs.

Among the other existing designs, a hardware design for BIKE has been presented in [20]. The work investigated different strategies to efficiently implement the BIKE algorithm on FPGAs. The authors improved already existing polynomial multipliers, proposed efficient designs to realize polynomial inversions, and implement the Black-Gray-Flip (BGF) decoder. The authors provided VHDL designs for key generation, encapsulation, and decapsulation. For the fastest design, the authors showed 2.7ms for the key generation, 0.1ms for the encapsulation, and 1.9ms for the decapsulation, the times correspond to the high-speed implementation for the lowest security level. The authors also provide data for light-weight implementation for the lowest security level. Their paper further discusses Level 3 parameters for BIKE, but does not give final hardware data for the that security level. The authors provide free, non-commercial license for the hardware code.[5] Another BIKE hardware implementation in [19] provides similar results but at much smaller area for their high-speed version. Two key arithmetic components from BIKE, polynomial multiplication and inversion were improved by implementing a sparse polynomial multiplier and extended euclidean algorithm based inversion unit due to which substantial amount of improvement was seen in all the primitives. The authors provided verilog designs for key generation, encapsulation, and decapsulation and they are available free under non-commercial license. [6]

Apart from earlier mentioned BIKE hardware implementations, [11] presents a practical approach of client (decapsulation and keygen) - server(Encapsulation) based model for generating the shared secret using BIKE. The presented design outperforms [20] in terms of time for all primitives but has significantly larger area footprint. [18] presents a comparison between pure software implementation, pure HLS design, and HLS based HW/SW codesign for BIKE. However the performance of any of these design do not outperform the performance results tabulated in Table 15.

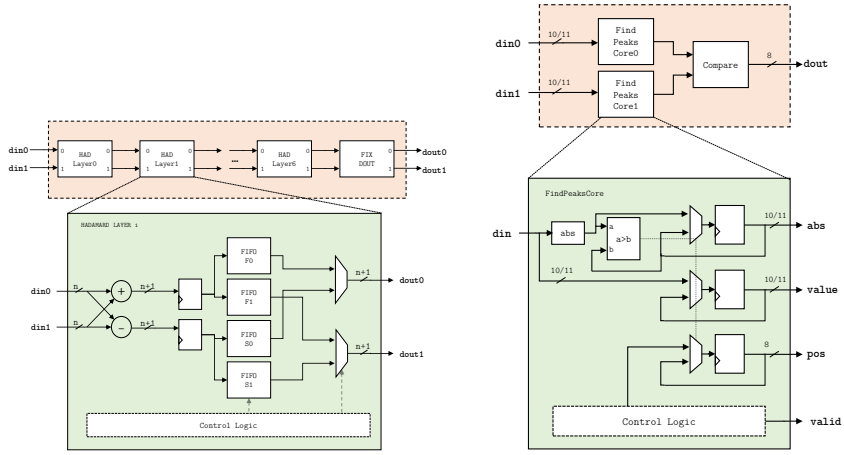
Classic McEliece has been most recently implemented in [8]. This is the first complete implementation of Classic McEliece KEM. The design provided Verilog code for encapsulation and decapsulation modules as well as key generation module with seed expansion. The authors presented three new algorithms that can be used for systemization of the public key matrix during key generation. The authors showed that the complete Classic McEliece design can perform key generation in 8.6ms, encapsulation in 0.3ms, and decapsulation in 0.9ms, the times correspond to the high-speed implementation for the lowest security level. The authors also provide hardware implementation for other security levels, and light-weight and high-speed versions for all the levels. The authors provide open-source code for the hardware.[7] Apart from the earlier implementation, [27] presented a high-throughput and compact key generation module. The authors presented improvements in Gaussian elimination, sorting unit and other hardware optimizations such as algorithm level pipelining. Overall, 11% reduction in

[5] https://github.com/Chair-for-Security-Engineering/BIKE
[6] https://github.com/Chair-for-Security-Engineering/RacingBIKE
[7] https://caslab.csl.yale.edu/code/pqc-classic-mceliece/

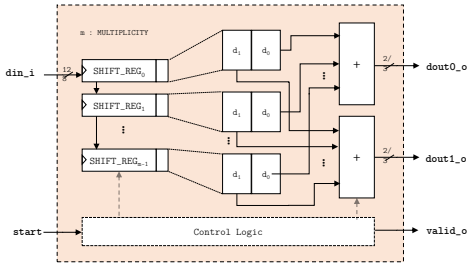
clock cycles, 31% reduction in BRAM utilization was observed with 11% increase in the logic utilization.

Hardware implementation of SIKE has been provided in [17]. The authors created VHDL implementation of SIKE as a hardware co-processor. Their design can realize any of the SIKE security levels. For the high-speed design for the lowest security level, authors report the time for encapsulation, decapsulation, and keygen as 15.3ms, 16.3ms, and 9.1ms respectively. The authors make the code available under Creative Commons public domain license.[8]

Different hardware implementations of CRYSTALS-Kyber are available in [16], [9], and [26]. The authors presented designs configurable for different performance, area requirements, and parameter sets. The high-speed design provided in [9] outperforms all other algorithms in terms of time for key generation, encapsulation, and decapsulation. For the lowest security level, the authors reported 0.02ms for key generation, 0.03ms for encapsulation, and 0.04ms for decapsulation. The authors did not provide access to the code for their hardware design.



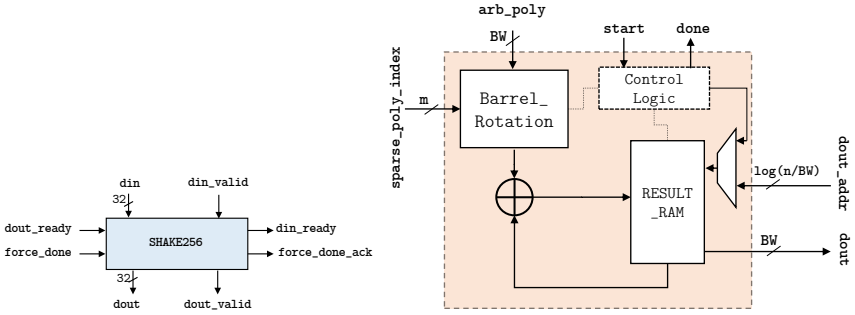
(a) `hadamard_transformation` module. (b) `find_peak` module.



(c) `expand_and_sum` module.

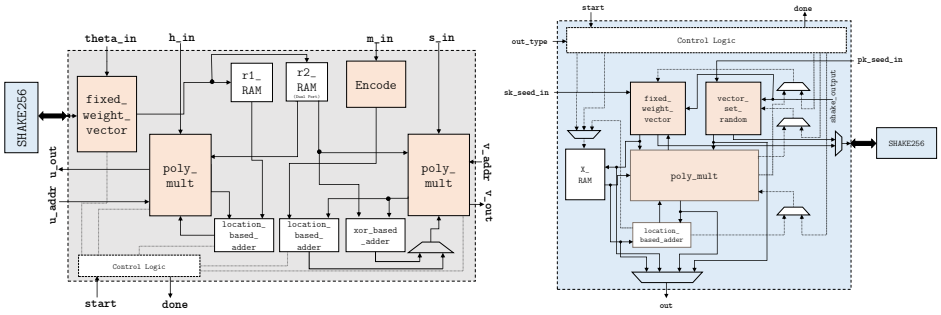
Fig. 6: Hardware design of Reed-Muller Decoder.

[8] https://github.com/pmassolino/hw-sike



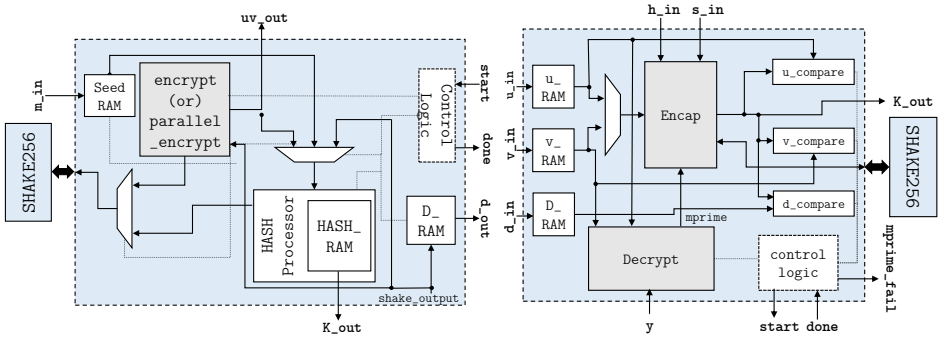
(a) Interface for `SHAKE256` module. (b) Hardware design of `poly_mult` module.

Fig. 7: Block diagram for interface of the `SHAKE256` module and `poly_mult` module.



(a) `parallel_encrypt` module (b) `keygen` module.

Fig. 8: Hardware design of `parallel_encrypt` and `keygen` modules.



(a) `encap` module. (b) `decap` module.

Fig. 9: Hardware design of `encap` and `decap` modules.