LOCS: LOw-Latency and ConStant-Timing Implementation of Fixed-Weight Sampler for HQC

Pengzhou He, Yazheng Tu, and Jiafeng Xie
Department of Electrical and Computer Engineering, Villanova University, Villanova PA 19087, USA
Email: {phe,ytu1,jiafeng.xie}@villanova.edu

2023 IEEE International Symposium on Circuits and Systems (ISCAS) | 978-1-6654-5109-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/ISCAS46773.2023.10181319

Abstract—**Post-quantum cryptography (PQC) has drawn significant attention from various communities recently and one of the recent advances is the hardware acceleration of PQC algorithms. While Hamming Quasi-Cyclic (HQC) is one of the recently announced National Institute of Standards and Technology (NIST) fourth-round PQC standardization candidates, very few related hardware implementation works have been reported, particularly lacking solid works on important components such as the sampler. As a fixed-weight sparse vector sampler with constant-time operation is critical to the hardware HQC accelerator, in this paper, we present a novel hardware-implemented LOw-latency and ConStant-timing fixed-weight sampler (LOCS). In total, we have proposed three stages of efforts. First of all, a new algorithm for efficient realization of the fixed-weight sparse vector generation based on Fisher-Yates shuffle algorithm is proposed. Then, we have innovatively designed the algorithm into a new hardware sampler: LOCS. Finally, we have conducted a thorough comparison to showcase the efficiency of the proposed sampler, e.g., the proposed LOCS involves 66.7% less latency time than the state-of-the-art design ($n = 17,669$) while remaining constant-time operation. To the authors' best knowledge, this is the first hardware-implemented pure constant-time (no failure probability) fixed-weight sampler for HQC.**

I. Introduction

As it has been proven that the traditional cryptosystems such as RSA and elliptic curve cryptography are vulnerable to the attacks launched from mature quantum computers [1], [2], the need for Post-Quantum Cryptography (PQC) is at an all-time high [1], [3]. The National Institute of Standards and Technology (NIST) started the PQC standardization process in 2016 and announced the fourth-round candidates recently. HQC, which stands for the Hamming Quasi-Cyclic (HQC), was selected as one of the fourth-round candidates [4], [5].

HQC is a code-based PQC scheme whose security relies on decoding small weight vectors of random quasi-cyclic codes [5]. Following the NIST PQC standardization, efficient hardware implementations of HQC are seriously needed, e.g., on Field-Programmable Gate Array (FPGA) devices.

Existing Works. Hardware implementation for PQC is one of the recent advances in the field [4], [6], [7], [8], [9], [10], [11], [12], [13]. So far, however, very few hardware implementations of HQC are available. The authors of HQC have released a high-level synthesized hardware implementation [5]. Another recent hardware design for HQC was given in [14].

Existing Challenges. It is noted that the fixed-weight sparse vector generation is one the major operations of HQC (see Algorithm 1 in Section II). Fixed-weight sparse vectors are involved in many steps of the algorithmic operations of HQC (e.g., the sparse polynomial multiplication) and thus the efficiency of the generation of such vectors is critical to the overall success of the implemented scheme. So far, however, efficient implementations of the sparse vector generation (especially hardware designs) are very rare and the major challenges include (as seen from the recent work of [14]): (i) there still exist situations that the generated indices are duplicated (when randomly sampling the indices for non-zero elements); (ii) meanwhile, the designed hardware structure requires extra procedures to check the incidents of the duplication of generated indices, which actually makes the whole sampling process a non-constant-timing operation; (iii) it is still time-consuming to generate the actual sparse vectors along with the indices of the non-zero elements at the same time.

Therefore, in this paper, we propose a novel hardware-implemented **LO**w-latency and **ConSt**ant-timing fixed-weight sampler (LOCS) for HQC. Key contributions are:

- We have presented the proposed constant-time algorithmic operation based on Fisher-Yates shuffling for the generation of the fixed-weight sparse vector (for HQC).
- We have then presented the corresponding hardware architecture with thorough internal structural descriptions.
- We have given the final comparison to demonstrate the superior performance of the proposed sampler.

Note that this is the first constant-time sampler for HQC (no failure probability), and the proposed sampler generates the fixed-weight sparse vectors into two forms, i.e., indices and the actual random binary vectors, for further use/calculation.

The rest of the paper is organized as follows. Section II gives the preliminary. Section III presents the proposed algorithm. Section IV introduces the proposed structure. The comparison and conclusion are given in Sections V and VI, respectively.

II. Preliminary Knowledge

Notations. We define $\mathbb{F}_2$ as the binary finite field. Vectors/polynomials in $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$ are represented by lower-case bold letters. $\omega(\cdot)$ denotes the Hamming weight of a vector, i.e., the number of its nonzero coordinates. $\mathcal{C}[n, k]$ denotes a linear code with length n and dimension k. Elements of $\mathcal{C}$ are referred to as codewords [15]. δ is the minimum number of errors that the decoding algorithm can correct. Also, all computations in HQC schemes are made in the ambient space $\mathbb{F}_2^n$. Interested readers may refer the details of these notations to [5]. Also, in the following of the paper, we use h

to represented the generated fixed-weight vectors and $P[i]$ to denote the index of the ith non-zero element in the vector.

HQC. HQC is a Chosen Ciphertext Attack (IND-CCA) secure encryption scheme built on the hardness of a decision version of the Syndrome Decoding on structured codes [5]. As HQC uses a decodable code $\mathcal{C}[n, k]$ and a random double-circulant $[2n, n]$ code, it features a detailed and precise upper bound for the decryption failure probability analysis [5]. Interested readers can refer to the document of [5] for details.

Algorithms. Let $\mathcal{G}(\cdot), \mathcal{H}(\cdot), \mathcal{K}(\cdot)$ denote SHAKE256-512 ($\cdot||$`G_FCT_DOMAIN`), SHAKE256-512 ($\cdot||$`H_FCT_DOMAIN`) and SHAKE256-512 ($\cdot||$`K_FCT_DOMAIN`), respectively. Algorithm 1 represents the key encapsulation mechanism (KEM) version of HQC. For the public key encryption (PKE) version, interested ones may read [5] for the detailed information.

Algorithm 1: HQC.KEM [5]

```
   Setup($1^\lambda$):
1  generate and output the global parameters param =
     $(n, k, \delta, \omega, \omega_\mathbf{r}, \omega_\mathbf{e})$, $k$ will be the length of the
     symmetric key being exchanged, typically k = 256;
   KeyGen(param):
2  samples $\mathbf{h} \leftarrow \mathcal{R}$, the generator matrix $\mathbf{G} \in \mathbb{F}_2^{k \times n}$ of $\mathcal{C}$;
3  $sk$ = ($\mathbf{x}$, $\mathbf{y}$) $\leftarrow \mathcal{R}^2$ such that $\omega = \omega(\mathbf{x}) = \omega(\mathbf{y})$;
4  $pk$ = ($\mathbf{h}$, $\mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$);
5  return ($pk$, $sk$);
   Encapsulate($pk$):
6  generate $\mathbf{m} \leftarrow \mathbb{F}_2^k$ ;
7  derive the randomness $\theta = \leftarrow \mathcal{G}(\mathbf{m})$;
8  generate the cyphertext
     $\mathbf{c} \leftarrow (\mathbf{u}, \mathbf{v}) = \mathcal{E}.\mathbf{Encrypt}(pk, \mathbf{m}, \theta)$;
9  derive the symmetric key $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$;
10 $d \leftarrow \mathcal{H}(\mathbf{m})$;
11 send ($\mathbf{c}$, $\mathbf{d}$);
   Decapsulate($pk, \mathbf{c}, \mathbf{d}$):
12 decrypt $\mathbf{m}' = \mathcal{E}.\mathbf{Decrypt}(sk, \mathbf{c})$;
13 compute $\theta' = \mathcal{G}(\mathbf{m}')$;
14 (re-)encrypt $\mathbf{m}'$ to get $\mathbf{c}' \leftarrow \mathcal{E}.\mathbf{Encrypt}(pk, \mathbf{m}', \theta')$;
15 if $\mathbf{c} \neq \mathbf{c}'$, or $\mathbf{d} \neq \mathcal{H}(\mathbf{m}')$ then
16     abort;
17 else
18     derive the shared key $K \leftarrow \mathcal{K}(\mathbf{m}, \mathbf{c})$;
19 end
```

Security. HQC has three security levels, namely hqc-128, hqc-192, hqc-256, respectively [5], each with different parameter sets. Note the proposed LOCS is applicable to all of them.

Sampling of the fixed-wight vectors As shown in Step 3 of Algorithm 1, two fixed-weight vectors are sampled/generated for further computation. These two vectors have the length of n and Hamming weight of ω, which means only ω elements in each vector are '1's while all other elements are '0's. Besides that, the positions/indices of the non-zero elements are random. Overall, the process of generating the indices and the vectors is defined as the "sampling of fixed-weight vectors".

Fisher-Yates Shuffle. The Fisher–Yates shuffle is an algorithm for generating a random permutation of a finite sequence. It was firstly described in [16] and further developed into a modern version in [17]. This algorithm has been proven to be able to produce an unbiased permutation (Algorithm 2).

Algorithm 2: The modern version of the Fisher–Yates shuffle

```
  Input  : Array $A$ with $n$ elements;
  Output: Shuffled array $A$;
1 for $i = 0$ to $n - 2$ do
2     $j \leftarrow$ random integer such that $i \leq j \leq n$;
3     exchange $A[i]$ and $A[j]$;
4 end
```

III. The Proposed Algorithmic Operation for Fixed-weight Sparse Vector Generation

Following the Fisher–Yates shuffling algorithm, we propose the constant-time fixed-weight sparse vector generating algorithm, as described in Algorithm 3.

Algorithm 3: Proposed algorithm for the generating of the fixed-weight sparse vector of HQC

```
   Input  : $n$, $\omega$;
   Output: $h$, $P$;
   Setup Step:
1  for $i = 0$ to $\omega - 1$ do
2      $h[i] = 1$;
3  end
4  for $i = \omega$ to $n - 1$ do
5      $h[i] = 0$;
6  end
   Swap Step:
7  for $i = 0$ to $\omega - 1$ do
8      $j \leftarrow$ random integer such that $0 \leq j \leq n$;
9      exchange $h[i]$ and $h[j]$;
10 end
   Index Marking Step:
11 $j = 0$
12 for $i = 0$ to $n - 1$ do
13     if $h[i] = 1$ then
14         $P[j] = i$;
15         $j = j + 1$;
16     else
17 end
```

In the existing strategy for generating the fixed-weight sparse vector [14], the possible duplication of the generated indices for non-zero elements will result in extra time and resource usage for re-generating a new index to make sure all the indices are distinct (as there will be one non-zero element less in the vector if two generated indices are the same). However, by applying the Fisher–Yates shuffling, our proposed algorithm is able to avoid this situation since ω non-zero elements are already set in the vector in advance. Also, the random numbers generated will make sure those elements are placed in random positions and the exchanging operation doesn't have an impact on the number of the non-zero elements (even if two identical indices are generated).

Example. To better illustrate how our proposed algorithm prevents re-generating extra indices, an example is given here.

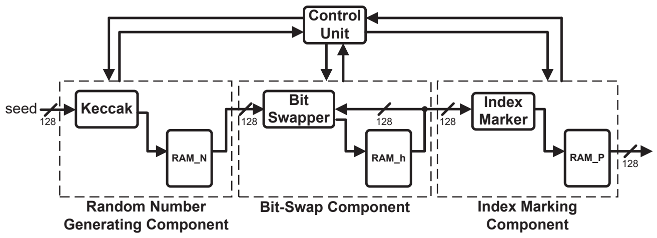

Fig. 1: The proposed fixed-weight sparse vector sampler.

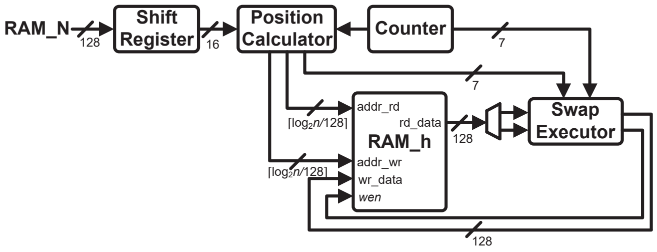

Fig. 2: Structure of BS component (interacting with RAM_h).

Suppose index 100 is generated at the second iteration during the swap step, then after executing the exchanging operation, $h[2]$='0' and $h[100]$= '1'. After two iterations, index 100 is generated again, the two elements, $h[4]$ and $h[100]$ will still both be '1' after the swap operation and this will not reduce the number of '1's in the vector.

Therefore, by keeping the number of non-zero elements in the vector always equal to ω and thus obviating the probability of generating extra indices, the proposed algorithm remains time-constancy even though identical indices are generated. Also, by first generating the vector itself and then marking all the indices of non-zero elements rather than generating the indices directly, the proposed algorithm is able to present the generated vectors in both two formats for future use, e.g., polynomial multiplication and outputting the secret key.

IV. LOCS: Proposed Sampler Architecture

Following the proposed algorithm, we further present the hardware-implemented LOCS. The proposed sampler contains four major components, namely Random Number Generating (RNG) Component, Bit Swapping (BS) Component, Index Marking (IM) Component, and Control Unit (CU), respectively, as shown in Fig. 1. Note that the data flow is 128-bit and can be extended to other bit-lengths (such as 32 or 64-bit).

RNG component. The RNG component is responsible for generating random numbers used for deriving the indices for non-zero elements, according to Step 8 of the proposed Algorithm 3. The generation of random numbers is realized by executing SHAKE-256 using a Keccak wrapper (including the Keccak core and other sub-components such as output buffer and control unit) to achieve the desired output length and format. When executing the SHAKE-256 operation, a 128-bit long message segment m is used as the seed to generate the random numbers and then sent into the component along with some other parameters like n and ω corresponding to different security levels of HQC. The output will be delivered in the length of 128-bit to the memory RAM_N, where all the generated random numbers are stored for further usage.

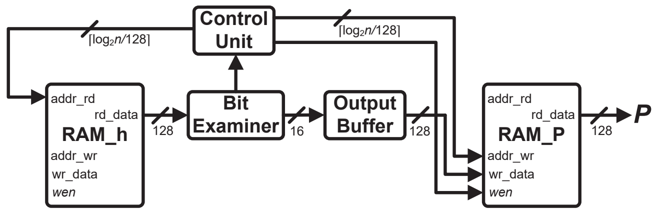

Fig. 3: Structure of IM (interacting with RAM_h and RAM_P).

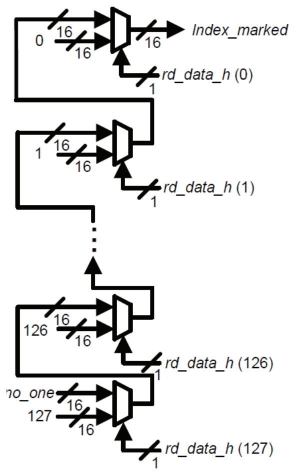

Fig. 4: The internal structure of the bit examiner.

BS component. The internal structure of the BS component is shown in Fig. 2, which consists of a shift register, a position calculator, a swap executor, and a counter. The BS component is used for shuffling the vector by swapping the bits at certain iterations during the swapping step, as stated in Step 9 of Algorithm 2. When executing the swapping operation, it firstly sets up the original vector where the first ω bits are '1's and writes it to RAM_h. Then, the BS component reads the generated random numbers from RAM_N and calculates the corresponding indices by executing modulo n operation. The calculated indices are then sent to the position calculator to determine the position in RAM_h of the corresponding bit, as well as the position of the bit corresponds to the current iteration (which is determined by the counter). After that, the swap executor reads two data segments containing the two bits mentioned above from RAM_h and swaps those two bits, and then writes them back to RAM_h. The vectors are shuffled and stored in RAM_h after the swapping step is done and then are delivered to the IM component to record the indices of non-zero elements that can be accessed for other usage. Note here the random numbers are set as 16-bit long since they cover the range from '0' to the biggest n (all three security levels of HQC) so that all the possible indices can be obtained.

IM component. The internal structure of the IM Component is shown in Fig. 3, which consists of a Bit Examiner, a Control Unit, and an Output Buffer. After it starts to work, the bit examiner reads a memory chunk from RAM_h and determines if there exists '1' or not. If there exists one or more '1's, the examiner will output the corresponding index (indices) to the output buffer and then set the bit(s) to 0. If there is no '1' in the chunk, the bit examiner will output a signal to the control unit to calculate the next address to read. When the output buffer

TABLE I: Comparison of the Implementation Results

Design	n	ω	LUT	FF	Slice	BRAM	Fmax	Latency	Delay[1]	Failure Probability.*	Constant-Time
Existing Sampler [14]	17,669	75	240	111	-	2	226	709	3.14	1.1×2^{-11}	No
Existing Sampler [14]	35,581	114	229	112	-	2	220	1,840	8.36	1.1×2^{-9}	No
Existing Sampler [14]	57,673	149	234	117	-	2	228	2,106	9.24	1.1×2^{-12}	No
Existing Sampler [14]	17,669	75	316	124	-	2	223	3,649	16.36	2.8×2^{-199}	Not Strictly
Existing Sampler [14]	35,581	114	295	125	-	2	246	4,200	17.80	1.1×2^{-280}	Not Strictly
Existing Sampler [14]	57,673	149	314	192	-	2.5	242	5,935	24.52	4.9×2^{-355}	Not Strictly
Proposed Sampler	17,669	75	1,560	766	490	2	170	976	5.45	N/A	Strictly Yes
Proposed Sampler	35,581	114	1,553	761	444	3	185	1,636	6.17	N/A	Strictly Yes
Proposed Sampler	57,673	149	1,569	779	464	3	181	2,268	8.69	N/A	Strictly Yes
Proposed Sampler[2]	17,669	75	7,004	4,176	2,004	2	170	976	5.45	N/A	Strictly Yes
Proposed Sampler[2]	35,581	114	6,985	4,148	1,979	3	185	1,636	6.17	N/A	Strictly Yes
Proposed Sampler[2]	57,673	149	7,211	4,173	1,985	3	181	2,268	8.69	N/A	Strictly Yes

Unit for delay: ns. Unit for Fmax: MHz.
*: Failure probability: the probability that the Sampler fails to generate the vector in constant time.
[1]: Delay is calculated as latency×(1/Fmax), where the latency refers to the computation cycles (Keccak core operation cycles are also included).
[2]: The performance listed here includes all the components (Keccack is also included). Note the Keccack core of [18] is used in the proposed sampler.

is full, the content in the buffer (ideally eight different indices for non-zero elements) will be written to the memory RAM_P. The structure determining the indices of '1's is depicted in a chain of MUXes, where one of the two inputs is the index while the other is the output from the MUX below, as shown in Fig. 4. When a bit is '1', the corresponding MUX will select its index and outputs it to the MUX above; on the other hand, if a bit is '0', the MUX will just propagate the output delivered from the MUX below. Here we have an indicator no_one (in the actual implementation we set it to a binary string of '1's) to indicate that there is no '1' existing. The control unit will be notified when the output of the Mux chain is equal to no_one and calculates the next memory address to read once the indicating signal is received. Note the IM examines all the memory addresses even if all the indices have been marked in order to achieve a constant-time operation.

V. Implementation and Comparison

In this section, we have implemented the proposed sampler of Fig. 1, corresponding to different security levels of HQC, on the FPGA platform. We have also compared the proposed sampler with the existing design to verify its efficiency.

Experimental Setup. The experimental setup is as follows: (i) the proposed design was described in VHDL and implemented on the Artix-7 xc7a200t-3 FPGA through Vivado 2020.2 (after place & route); (ii) we have obtained its implementation performance under three security level parameter sets of HQC ($n = 17,669$, $\omega = 75$; $n = 35,581$, $\omega = 113$; and $n = 57,637$, $\omega = 149$); (iii) the obtained implementation results, including the number of resource usage (LUTs, FFs, and BRAMs), maximum frequency (MHz), latency, delay time, and related failure probability are listed in Table I along with those of the existing one [14]; (iv) as the design of [14] only reported the resource usage of the sampler core (excluding the Keccak), we similar also listed the related performance for the sake of a fair comparison (nevertheless, the implementation results of our full sampler are also listed).

Implementation Results and Comparison. As shown in Table I, the resource utilization of the proposed sampler remains stable for different security levels as the main data flow in the sampler is fixed and the major difference between different security implementations is the number of iterations. On the other hand, the latency of the proposed design increases proportionally to ω because the higher number of non-zero elements leads to more iterations when generating the indices and swapping the corresponding bits.

Besides that, it is shown that the proposed sampler has a much lower latency than the existing design, i.e., 66.7%, 65.3%, and 64.6% less delay time than the existing design for hqc-128, hqc-192, and hqc-256, respectively. As the proposed sampler is based on the Fisher-Yates shuffle, it requires more resource usage than the existing design of [14]. Nevertheless, the existing one of [14] still suffers the probability that it could not complete the generation in a constant time; while the proposed design eliminates all chances of failure and is a completely time-constant fixed-weight sparse vector sampler. Moreover, the proposed sampler is able to generate two forms of vectors, namely indices and vectors, which are stored in the memory for further steps of usage (while the existing design does not provide that). In conclusion, the proposed sampler is more practical for actual usage than the existing one of [14].

Future Work and Discussion. The proposed LOCS, to our best knowledge, is the first real hardware-implemented real constant-timing sampler for HQC (also with low-latency). We hope the following works can focus more on the actual hardware acceleration of HQC and related side-channel attacks.

VI. Conclusion

This paper presents a novel hardware-implemented fixed-weight sampler for HQC with real constant-time operation (first work in the literature). We have presented the proposed constant-timing algorithmic process based on the Fisher-Yates shuffling for vector generation. Then, the details of the structure of the hardware sampler based on the proposed algorithm are provided. Finally, the implementation results of the proposed design and the comparison are presented to confirm the efficiency of the proposed design.

VII. Acknowledgement

The work of J. Xie was supported by NIST-60NANB20D20 and NSF SaTC-2020625.

References

[1] P. W. Shor, “Algorithms for quantum computation: discrete logarithms and factoring,” in *Proceedings 35th annual symposium on foundations of computer science*, pp. 124–134, Ieee, 1994.

[2] K. Basu and et al., “NIST post-quantum cryptography-a hardware evaluation study,” *Cryptology ePrint Archive*, 2019.

[3] J. Xie *et al.*, “Special session: The recent advance in hardware implementation of post-quantum cryptography,” in *IEEE VTS*, pp. 1–10, 2020.

[4] “PQC Standardization Process: Announcing Four Candidates to be Standardized, Plus Fourth Round Candidates.” https://csrc.nist.gov/News/2022/pqc-candidates-to-be-standardized-and-round-4.

[5] C. A. Melchor *et al.*, “Hamming quasi-cyclic (HQC) (NIST Round 3 Submission).” https://pqc-hqc.org/index.html.

[6] A. Wang, W. Tan, K. K. Parhi, and Y. Lao, “Integral sampler and polynomial multiplication architecture for lattice-based cryptography,” 2022.

[7] W. Tan, A. Wang, Y. Lao, X. Zhang, and K. Parhi, “Low-latency vlsi architectures for modular polynomial multiplication via fast filtering and applications to lattice-based cryptography,” 10 2021.

[8] J. Xie, P. He, and C.-Y. Lee, “CROP: FPGA implementation of high-performance polynomial multiplication in saber kem based on novel cyclic-row oriented processing strategy,” in *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pp. 130–137, IEEE, 2021.

[9] D.-e.-S. Kundi, Y. Zhang, C. Wang, A. Khalid, M. O’Neill, W. Liu, *et al.*, “Ultra high-speed polynomial multiplications for lattice-based cryptography on FPGAs,” *IEEE Transactions on Emerging Topics in Computing*, no. 01, pp. 1–1, 2022.

[10] S. Bian, M. Hiromoto, and T. Sato, “Filianore: Better multiplier architectures for lwe-based post-quantum key exchange,” in *DAC*, pp. 1–6, 2019.

[11] Y. Zhang, C. Wang, D. E. S. Kundi, A. Khalid, M. O’Neill, and W. Liu, “An efficient and parallel r-lwe cryptoprocessor,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 5, pp. 886–890, 2020.

[12] P. He and et al., “Fpga implementation of compact hardware accelerators for ring-binary-lwe based post-quantum cryptography,” *ACM Transactions on Reconfigurable Technology and Systems*, 2022.

[13] B. J. Lucas and et al., “Lightweight hardware implementation of binary ring-lwe pqc accelerator,” *IEEE Computer Architecture Letters*, vol. 21, no. 1, pp. 17–20, 2022.

[14] S. Deshpande, M. Nawan, K. Nawaz, J. Szefer, and C. Xu, “Towards a fast and efficient hardware implementation of HQC,” *Cryptology ePrint Archive*, 2022.

[15] S. Lin and D. J. Costello, *Error control coding*, vol. 2. Prentice hall New York, 2001.

[16] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company, 1953.

[17] R. Durstenfeld, “Algorithm 235: Random permutation,” *Commun. ACM*, vol. 7, p. 420, 1964.

[18] K. Team. Keccak in VHDL: High-speed core. https://keccak.team/hardware.html November.