Sota Voce: Low-Noise Sampling of Sparse Fixed-Weight Vectors

Décio Luiz Gazzoni Filho[1,2], Gora Adj[3], Slim Bettaieb[3], Alessandro Budroni[3], Jorge Chávez-Saab[3] and Francisco Rodríguez-Henríquez[3]

[1] Department of Electrical Engineering, State University of Londrina, Londrina, Brazil
[2] Instituto de Computação, Universidade Estadual de Campinas (UNICAMP), Campinas, Brazil
decio.gazzoni@ic.unicamp.br
[3] Cryptography Research Centre, Technology Innovation Institute, Abu Dhabi, UAE
{gora.adj,slim.bettaieb,alessandro.budroni,jorge.saab,francisco.rodriguez}@tii.ae

Abstract. Many post-quantum cryptosystems require generating an n-bit binary vector with a prescribed Hamming weight ω, a process known as *fixed-weight sampling*. When $\omega = O(n)$, we call this *dense* fixed-weight sampling, which commonly appears in lattice-based cryptosystems, like those in the NTRU family. In contrast, code-based cryptosystems typically use *sparse* fixed-weight sampling with $\omega = o(n)$ (e.g., $O(\sqrt{n})$. Sparse fixed-weight sampling generally involves three constant-time steps to keep the sampled vector secret: 1. sample ω nearly uniform random integers from a series of decreasing intervals; 2. map these integers into a set of ω distinct indices in $[0, n)$, called the *support*; 3. generate a binary n-bit vector with bits set only at the support indices. Remarkably, some of the core algorithms employed in fixed-weight sampling date back to nearly a century, yet developing efficient and secure techniques remains essential for modern post-quantum cryptographic applications.

In this paper, we present novel algorithms for steps two and three of the fixed-weight sampling process. We demonstrate their practical applicability by replacing the current fixed-weight sampling routine in the HQC post-quantum key exchange mechanism, recently selected for NIST standardization. We rigorously prove that our procedures are sound, secure, and introduce little to no bias. Our implementation of the proposed algorithms accelerates step 2 by up to 2.9× and step 3 by up to 5.8× compared to an optimized version of the fixed-weight sampler currently used in HQC. Since fixed-weight sampling constitutes a significant portion of HQC's execution time, these speedups translate into protocol-level improvements of up to 1.37×, 1.28× and 1.21× for key generation, encapsulation and decapsulation, respectively.

Keywords: Fixed-weight sampling · HQC · Sorting networks · Constant-time implementation

1 Introduction

On August 13, 2024, NIST concluded its Post-Quantum Cryptography Standardization project, an eight-year effort that was launched at PQCrypto 2016, by issuing the first three final standards: ML-KEM [FIP24a], ML-DSA [FIP24b], and SLH-DSA [FIP24c]. To promote cryptographic agility and diversify the underlying hardness assumptions, NIST continued evaluating alternate candidates. On March 11, 2025, it selected Hamming Quasi-Cyclic (HQC) [AAB+22] as an additional algorithm for public-key encryption and key encapsulation mechanism (KEM). Based on the hardness of decoding random quasi-cyclic codes, HQC is intended as a substitute or fallback option to ML-KEM in contexts where cryptographic diversity or defense-in-depth is preferred. This designation makes HQC

the fifth algorithm recognized under NIST's post-quantum cryptography standardization, thereby further reinforcing the emerging Post-Quantum Cryptography portfolio.

Sampling vectors of fixed Hamming weight represents a fundamental computational challenge in post-quantum cryptography. Notably, the secret solution to a Syndrome Decoding Problem instance is a vector with low and bounded Hamming weight, making such sampling an essential component in many code-based cryptographic schemes such as the Round 4 finalists of the NIST Post-Quantum Cryptography Standardization project HQC [AAB+22], BIKE [ABB+22], and Classic McEliece [ABC+22]. We distinguish between two regimes: if $\omega = O(n)$, the binary vector is said to be *dense*; if $\omega = o(n)$ (e.g., $O(\sqrt{n})$), it is considered *sparse*. This work focuses on the latter case, which corresponds to the scenario of the parameters in HQC.[1]

Typically, the computational routine to sample random binary vectors of length n and fixed Hamming-weight ω involves the following three steps:

Step 1 (**Sample integers**) Choose ω integers uniformly at random (up to a negligible bias) from a series of decreasing intervals;

Step 2 (**Construct the support**) Map the sampled sequence of integers (which may contain repeated values) to a set of ω distinct indices in $[0, n)$, referred to as the *support*;

Step 3 (**Write the support to a bit vector**) Create an n-bit vector of weight ω by setting the positions indicated in the support to 1 and the rest to 0.

Generally, the sampled fixed-weight vector is part of sensitive data. To ensure side-channel resilience, all three steps described above must be implemented in constant time, i.e., the algorithm's execution time must be independent of the sampled list of integers and of the secret support values. This constraint typically results in a performance penalty. For instance, Step 3 could be implemented naïvely in $O(\omega)$ time;[2] however, a straightforward constant-time implementation increases the time complexity to $O(n\omega)$.

In this work, we introduce a series of theoretical and practical contributions to the algorithms used in Steps 2 and 3 of sparse fixed-weight sampling. Our efforts aim to improve efficiency while preserving side-channel resistance, particularly in the sparse regime relevant to code-based cryptography. We demonstrate the effectiveness of our contributions by showing the substantial performance gains when applied to the HQC protocol.

Related work. Three of the Round 4 key encapsulation mechanism (KEM) finalists in the NIST post-quantum standardization process, namely, Classic McEliece [ABC+22], BIKE [ABB+22], and HQC [AAB+22], require fixed-weight sampling as a subroutine. Both HQC and BIKE relied on an algorithm based on rejection sampling until Round 3. Specifically, the implementations of the aforementioned Steps 1 and 2 consisted of sampling B-bit integers uniformly from the set $\{0, 1, \ldots, 2^B - 1\}$, with $n \leq 2^B - 1$, and rejecting values either outside the desired range or already sampled. Nevertheless, this rejection-based approach renders the entire procedure non-constant time with respect to the different seeds used to generate the randomness. This observation was cleverly exploited to enable a key-recovery timing attack against HQC and BIKE in [GHJ+22]. Motivated by this work, Sendrier proposed a modified sampling algorithm in [Sen21] that thwarts the timing attack of [GHJ+22] without significantly impacting the efficiency of the scheme. The algorithm runs in quadratic time with respect to the support size, i.e., $O(\omega^2)$.

The lattice-based finalist NTRU [CDH+20] and the alternate candidate NTRU Prime [BBC+20], both in Round 3 of the NIST post-quantum standardization process, employ an

[1] More precisely, the three HQC instances for levels 1, 3 and 5 are designed such that $\omega \approx \frac{\sqrt{n}}{2}$.

[2] Here, as in the rest of the paper, we report running times in a RAM model with unit cost for $\log(n)$-bit operations.

algorithm based on sorting networks, with complexity $O(n \log^2 n)$, to sample dense length-n ternary vectors with fixed Hamming weight. More recently, a constant-time variant of the Fisher–Yates shuffle, explicitly designed for the dense fixed-weight sampling setting, was proposed in [FSL24], with $O(n)$ complexity. The running time of these algorithms is a function of n alone and not ω. As such, they do not display improved performance when moving from the dense regime to the more favorable sparse regime.

Contributions. In this paper, we improve the overall routine for sampling sparse fixed-weight binary vectors, focusing on the parameter settings of HQC. Although we chose HQC as our target application due to its selection by NIST for standardization, we believe that our improvements are equally applicable to other schemes making use of sparse fixed-weight sampling, such as BIKE and Classic McEliece.

Our work involves analyzing and designing several core algorithms, leading to significant practical gains. While we adopt Sendrier's sampling algorithm and analysis [Sen21] for Step 1, we contribute both theoretical and practical enhancements to Step 2 and Step 3, as detailed above. In a nutshell our contributions include:

- For Step 2, we conduct a systematic study of state-of-the-art algorithms, along with two original proposals introduced in this work. Specifically:
  - We provide a constant-time AVX2-vectorized implementation of Sendrier's shuffle, which we refer to as SendrierAVX2, whose performance improves upon the HQC optimized implementation [HQC25].
  - We introduce a novel alternative to Sendrier's algorithm based on permutation unranking via Lehmer codes [Leh60] that we name TLShuffle. Despite having the same time complexity of $O(\omega^2)$, it slightly outperforms SendrierAVX2 in practice for some parameter sets of interest in the HQC support sampling procedure.
  - We implement and analyze DJBSample by Daniel J. Bernstein.[3] We prove its correctness, unbiasedness and time complexity of $O(\omega \log^2 \omega)$.
  - We introduce a new, efficient, and simple – albeit slightly biased – algorithm, named Sota, which resembles ideas from [Knu11, §7.2.1.3] and whose time complexity is $O(\omega \log^2 \omega)$. In addition to demonstrating its correctness, we prove that the bias it introduces is small for the parameter sets used in HQC.

  Compared to Sendrier's state-of-art algorithm [Sen21], all of the above present either an analogous or a faster time-complexity for the choices of n and ω used in HQC.
- For Step 3, we significantly improve upon HQC's reference and optimized procedures. In particular, we introduce a new algorithm, MWSupport, based on merging and sorting networks, which reduces the complexity of this step from $O(n\omega) = O(n^{3/2})$ to $O(n \log n)$, and yields significantly lower execution time in practice.

We implement and publish all proposed algorithms under a permissive open-source license, providing both pure C and AVX2-optimized versions—including a vectorized implementation of Sendrier's algorithm, which was not optimized in [HQC25]. We compare these implementations against the optimized implementation of Sendrier's algorithm from the official optimized HQC codebase [HQC25]. Our benchmarks reveal speedups ranging from 5.4× to 15× for the algorithm corresponding to Step 2, and from 2.6× to 5.2× for the algorithm corresponding to Step 3.

To evaluate the impact of our contribution, we include all these algorithms in the HQC scheme's optimized implementation. The results reported in Table 1 present speedups

[3] Made public as a Python script at https://cr.yp.to/2024/gcww-20240512.py.

of 1.28–1.31× in Keygen, 1.21–1.22× in Encapsulation, and 1.14–1.15× in Decapsulation for hqc-128. For hqc-192, the obtained speedups are 1.34–1.37× in Keygen, 1.26–1.27× in Encapsulation, and 1.20–1.21× in Decapsulation. Lastly, we obtain speedups of 1.35–1.37× in Keygen, 1.27–1.28× in Encapsulation, and 1.21× in Decapsulation for hqc-256.

Table 1: Impact of our algorithms on the official optimized implementation of HQC [HQC25]. Performance figures are reported in thousands of clock cycles. The speedups reported are with respect to [HQC25].

		[HQC25]	MWSupport +			
			SendrierAVX2	TLShuffle	DJBSample	Sota
hqc-128	Keygen	64.5	50.2 (1.28×)	49.9 (1.29×)	49.7 (1.30×)	49.3 (1.31×)
	Encaps.	144	119 (1.21×)	119 (1.21×)	118 (1.22×)	118 (1.22×)
	Decaps.	260	228 (1.14×)	227 (1.15×)	227 (1.15×)	226 (1.15×)
hqc-192	Keygen	153	114 (1.34×)	114 (1.34×)	113 (1.35×)	112 (1.37×)
	Encaps.	334	266 (1.26×)	267 (1.26×)	265 (1.26×)	263 (1.27×)
	Decaps.	541	450 (1.20×)	451 (1.20×)	448 (1.21×)	448 (1.21×)
hqc-256	Keygen	306	226 (1.35×)	225 (1.36×)	225 (1.36×)	224 (1.37×)
	Encaps.	652	513 (1.27×)	512 (1.27×)	511 (1.28×)	509 (1.28×)
	Decaps.	1062	881 (1.21×)	880 (1.21×)	879 (1.21×)	877 (1.21×)

Organization. Section 2 provides the notation and necessary background. Sections 3 and 4 introduce the algorithms corresponding to Step 2 and Step 3, respectively. We present the experimental results in Section 5, and conclude in Section 6.

2 Preliminaries

Notation. We use calligraphic capital letters (e.g., $\mathcal{A}$) to denote sets. For a finite set $\mathcal{A}$, the notation $r \leftarrow D(\mathcal{A})$ indicates that the element r has been sampled from the set $\mathcal{A}$ following a probability distribution D. We use $U(\mathcal{A})$ to denote the uniform distribution over $\mathcal{A}$. The notation $|\cdot|$ is used to indicate either the cardinality of a set (e.g., $|\mathcal{A}|$) or the absolute value of a real number (e.g., $|a|$, for a real number a). We denote with $[m, n]$ the set $\{m, m+1, \ldots, n-1\}$, for integers $m \leq n$, and $[n] = [0, n] = \{0, 1, \ldots, n-1\}$.

Vectors (respectively, matrices) are written with lower-case (respectively upper-case) bold letters. For any positive integer n, let $\mathcal{R}$ denote the polynomial ring $\mathcal{R} = \mathbb{F}_2[x]/(x^n - 1)$. Given a positive integer ω, we define the set $\mathcal{R}_\omega$ as the set of vectors in $\mathcal{R}$ having Hamming weight exactly ω.

Our algorithms make use of the $\mathsf{CSwap}(b, x, y)$ function, which swaps the values of the x and y variables if b is *True*, and does nothing otherwise.

2.1 Sparse fixed-weight vector sampling

The computational task of sampling a random vector of length n and Hamming weight ω, with $\omega = O(\sqrt{n})$, can be abstracted to sampling ω elements, corresponding to the support, from the set $[n]$ *without replacement*. A naive algorithm for doing so consists of repeatedly sampling elements and discarding them if they have been sampled already earlier. Then, one must construct the binary vectors whose support corresponds to the obtained indices. A more efficient approach for performing the same task is based on the *Fisher–Yates shuffle* [FY38] and is reported in Algorithm 1.

Algorithm 1: FisherYates(n, ω)

Input: parameters: n, ω
Output: A binary vector of length n and weight ω

```
1 pos = [0, 1, ..., n − 1]
2 v = [0, ..., 0] ∈ {0, 1}^n
3 for i = 0 to ω − 1 do
4     j ← U([i, n]);                    // Step 1
5     swap(pos[i], pos[j]);             // Step 2
6     v[pos[i]] = 1;                    // Step 3
7 return v
```

From a cryptographic implementation perspective, although the Fisher–Yates-based method has linear-time complexity $O(\omega)$, it poses three main challenges corresponding to the core algorithmic steps outlined in Section 1:

Chall. 1 Sampling a list of ω integers in constant-time and uniformly at random from an interval (Line 4) whose size is not, in general, a power of two, is non-trivial;

Chall. 2 A naive implementation of the swap operation (Line 5) involves secret-dependent memory accesses, making it vulnerable to side-channel attacks;

Chall. 3 A naive implementation of the routine to construct the binary vector from the support (Line 6) also makes secret-dependent memory accesses.

Despite its apparent simplicity, addressing the above challenges inevitably introduces time-complexity and performance penalties, and identifying the most effective and reliable strategies requires clever algorithmic design. In Section 2.2.1, we briefly explain how the HQC protocol overcomes these challenges.

2.2 HQC

Our work is based on the fourth round update of the HQC specification from February 2025 [AMAB+25]. HQC is built upon an IND-CPA secure public key encryption (PKE) scheme, which is presented in Algorithm 2, Algorithm 3 and Algorithm 4. Note that these algorithms show only the PKE algorithms and not the complete KEM construction, as our focus in this work is on sampling fixed-weight vectors, a component that appears specifically at the PKE level. To achieve an IND-CCA secure KEM, the Fujisaki–Okamoto transformation with implicit rejection ($\mathsf{FO}^{\not\perp}$) [FO99, HHK17] is applied on top of the PKE.

HQC defines three parameter sets of the form hqc-λ, where $\lambda \in \{128, 192, 256\}$, targeting NIST security levels 1, 3, and 5, respectively. The specific parameters for each set are listed in Table 2. Each security level uses a parameter n that defines the polynomial ring $\mathcal{R} = \mathbb{F}_2[x]/(x^n - 1)$ over which operations are performed. In addition, HQC employs an error-correcting code $\mathcal{C}$, defined as the concatenated Reed–Muller and Reed–Solomon code.

2.2.1 Fixed-weight vector sampling in HQC

In the HQC scheme, the generation of the key pairs and ciphertext relies on structured sampling of vectors from $\mathcal{R}$. Two sampling procedures are employed: SampleVect, which samples uniformly random vectors from $\mathcal{R}$, and SampleFW, which also samples vectors in $\mathcal{R}$ but with a fixed Hamming weight. In the key generation algorithm (Algorithm 2), a uniformly random vector $\mathbf{h} \in \mathcal{R}$ is generated using SampleVect, while the secret key vectors $\mathbf{x}$ and $\mathbf{y}$ are sampled in $\mathcal{R}_\omega$ using SampleFW. During encryption (Algorithm 3),

three vectors are sampled using SampleFW: $\mathbf{r}_1$ and $\mathbf{r}_2$ in $\mathcal{R}_{\omega_r}$, and $\mathbf{e}$ in $\mathcal{R}_{\omega_e}$. For each security level, the values of ω_r and ω_e are equal, and all parameter values are specified in Table 2. The goal of SampleFW is to generate a vector in $\mathcal{R}_\omega$, a bit vector of length n with fixed Hamming weight ω.

This structured sampling is carried over to the KEM construction, where the PKE key generation algorithm maps directly to the KEM key generation procedure. The encapsulation procedure, on the other hand, is based on the PKE encryption algorithm but includes slight modifications. Due to the FO transform, the decapsulation procedure in the KEM involves a re-encryption step to verify the validity of the ciphertext. This re-encryption requires reconstructing the randomness used during encapsulation. As a result, SampleFW is used not only in KEM key generation and encapsulation, but also in decapsulation, making it a core building block throughout the entire KEM operation.

We explain the implementation of SampleFW following the same 3-step framework from the Introduction (see also Section 2.1). Steps 1 and 2 are implemented by GenRandSupport (Algorithm 5), initially proposed by [Sen21] and adopted in HQC. Although GenRandSupport does not sample a uniformly random subset of size ω, Section 5.3 of the HQC specification shows that the statistical bias introduced by this method does not impact the security of the scheme, with the analysis being inspired by that of [Sen21]. Importantly, its use prevents rejection-based side-channel attacks like those of [GHJ+22]. The algorithm internally relies on Rand (Algorithm 6), which returns an integer in $[n]$ by drawing 32 pseudorandom bits from a SHAKE256-based seed expander and reducing modulo n. That is, RandBits(32) returns the next 32 bits of the seed expander stream, interpreted as an integer in $[2^{32}]$. As noted in Section 1, computing Step 2 using the GenRandSupport algorithm requires quadratic time with respect to the support size, $O(\omega^2)$. Given HQC's design choice of setting $\omega = O(\sqrt{n})$, this step results in an overall linear time complexity, $O(n)$.

Finally, Step 3, i.e., writing the support positions into a bit vector $\mathbf{v}$ in a side-channel-resistant manner, is handled by Algorithm 8 (SupportToVector). The algorithm computes, for each support index, the corresponding 64-bit word and bit mask, and then sets the bits using bitwise logic that treats all indices uniformly. This constant-time implementation ensures uniform treatment of all indices and incurs a time cost of $O(n\omega) = O(n^{3/2})$.

Algorithm 2: HQC-PKE.Keygen

Input: param
Output: sk, pk

1 $\sigma \leftarrow U(\{0,1\}^\lambda)$
2 $\mathbf{y} \leftarrow \mathsf{SampleFW}(\mathcal{R}_\omega)$
3 $\mathbf{x} \leftarrow \mathsf{SampleFW}(\mathcal{R}_\omega)$
4 $\mathbf{h} \leftarrow \mathsf{SampleVect}(\mathcal{R})$
5 $\mathsf{sk} \leftarrow (\mathbf{x}, \mathbf{y}, \sigma)$
6 $\mathsf{pk} \leftarrow (\mathbf{h}, \mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y})$

Algorithm 3: HQC-PKE.Encrypt

Input: pk, $\mathbf{m}$, θ
Output: $\mathbf{c} = (\mathbf{u}, \mathbf{v})$

1 InitSampleFW(θ)
2 $\mathbf{r}_2 \leftarrow \mathsf{SampleFW}(\mathcal{R}_{\omega_r})$
3 $\mathbf{e} \leftarrow \mathsf{SampleFW}(\mathcal{R}_{\omega_e})$
4 $\mathbf{r}_1 \leftarrow \mathsf{SampleFW}(\mathcal{R}_{\omega_r})$
5 $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
6 $\mathbf{v} \leftarrow \mathcal{C}.\mathsf{Encode}(\mathbf{m}) + \mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$

Algorithm 4: HQC-PKE.Decrypt

Input: $\mathsf{sk} = (\mathbf{x}, \mathbf{y}, \sigma)$, $\mathbf{c} = (\mathbf{u}, \mathbf{v})$
Output: $\mathbf{m}$

1 $\mathbf{m} \leftarrow \mathcal{C}.\mathsf{Decode}(\mathbf{v} - \mathbf{u} \cdot \mathbf{y})$

Table 2: Parameter sets for HQC.

Instance	Security	n	ω	$\omega_r = \omega_e$	DFR
hqc-128	NIST-L1	17 669	66	75	$< 2^{-128}$
hqc-192	NIST-L3	35 851	100	114	$< 2^{-192}$
hqc-256	NIST-L5	57 637	131	149	$< 2^{-256}$

Algorithm 5: GenRandSupport

Input: $\mathcal{R}_\omega$
Output: Positions pos[0], ..., pos[$\omega - 1$]

for $i \leftarrow \omega - 1$ **to** 0 **do**
  $l \leftarrow i + \mathsf{Rand}(n - i)$;
  if $l \in \{\, \mathrm{pos}[j] \mid i < j < \omega \,\}$ **then**
    $\mathrm{pos}[i] \leftarrow i$;
  else
    $\mathrm{pos}[i] \leftarrow l$;
return pos[$0, \ldots, \omega - 1$]

Algorithm 6: Rand

Input: n
Output: An integer in $[n]$

$x \leftarrow \mathsf{RandBits}(32)$
return $x \bmod n$

Algorithm 7: SampleFW

Input: $\mathcal{R}_\omega$
Output: $\mathbf{v} \in \mathcal{R}_\omega$

pos[$0, \ldots, \omega - 1$] $\leftarrow \mathsf{GenRandSupport}(\mathcal{R}_\omega)$
$\mathbf{v} \leftarrow \mathsf{SupportToVector}(\mathrm{pos}[0, \ldots, \omega - 1])$
return $\mathbf{v}$

Algorithm 8: SupportToVector

Input: Positions pos[$0, \ldots,$ pos[$\omega - 1$]
Output: Bit vector $\mathbf{v} \in \mathcal{R}_\omega$

`// Precompute index and bit masks`
for $i \leftarrow 0$ **to** $\omega - 1$ **do**
  index_tab[i] $\leftarrow \mathrm{pos}[i] \div 64$
  bit_tab[i] $\leftarrow 1 \ll (\mathrm{pos}[i] \bmod 64)$

`// Write the support in v`
for $i \leftarrow 0$ **to** $\left\lceil \frac{n}{64} \right\rceil - 1$ **do**
  `val` $\leftarrow 0$
  for $j \leftarrow 0$ **to** $\omega - 1$ **do**
    `// Constant-time check:`
    `does` i `==index_tab[j]?`
    same $\leftarrow \mathsf{IsEqual}(i, \text{index_tab}[j])$
    val $\leftarrow$ val $\mid$ (bit_tab[j] $\cdot$ same)
  $\mathbf{v}[i] \leftarrow \mathbf{v}[i] \mid$ val
return $\mathbf{v}$

2.3 Merging and sorting networks

Some fixed-weight or permutation sampling algorithms require constant-time sorting of integers. Classical algorithms like mergesort or quicksort are unsuitable in cryptographic contexts, as their control flow and memory access patterns may leak secret data. A suitable alternative is to use sorting networks.

Knuth [Knu97, Section 5.3.4] defines sorting networks as oblivious comparator networks—fixed sequences of compare-and-swap operations that do not depend on the input. This data-independent structure makes them well-suited for parallel execution, hardware implementation, and constant-time behavior, which are critical properties for side-channel-resistant cryptographic algorithms. In post-quantum cryptography, sorting networks have been used to ensure constant-time sorting [BCS13, BCLvV17], and a public-domain library based on these ideas was released in [Ber19].

Sorting networks operate using a fixed layout of *compare-exchange* steps: given a pair (a, b), swap the elements if $a > b$. This operator can be implemented efficiently in constant time using `min`/`max` instructions, conditional moves, or standard bitwise techniques with logical operations and shifts.

Sorting networks may be either *regular* or *irregular*: the former type allows algorithmic construction of arbitrarily-sized networks, whereas the latter type are specific to a particular size, without a clear pattern to generalize to different sizes. Among the former, the best known constructions are due to Batcher, and consist of the application of the mergesort algorithm, replacing the textbook merging algorithm by a *merging network*. Batcher proposed two different constructions: *bitonic* and *odd-even* merging networks [Bat68]. Both constructions require $O(n \log n)$ compare-exchange operations to merge two sequences of n values each; as mergesort has $O(\log n)$ stages, $O(n \log^2 n)$ compare-exchange operations are required to sort n values. The bitonic sorter seeks to increase parallelism by reducing

the *depth* of the resulting network, whereas the odd-even network seeks to reduce the number of compare-exchange operations required. They can be freely interchanged at each level of the mergesort algorithm. We find the bitonic network to be more amenable to vectorization, compensating for the increased number of compare-exchange blocks, especially when sorting small (e.g. 16-bit) elements.

We also survey other regular merging and sorting networks of note. Van Voorhis [VV71a, VV71b] departs from the standard recursive bisection approach, designing networks with a low number of compare-exchange blocks that remain competitive decades later. Hong and Sedgewick [HS82] introduce the *even-odd* merging network, which matches the depth of Batcher's bitonic network but uses slightly more blocks than his odd-even variant. They also offer an alternative, more intuitive construction of Batcher's networks. Ajtai, Komlós, and Szemerédi [AKS83] prove the existence of sorting networks with optimal $O(n \log n)$ complexity, though the impractically large constants limit their real-world use. Parberry [Par92] presents the *pairwise* sorting network, which matches the depth and block count of mergesort with Batcher's odd-even network, via a distinct construction. Interesting also is Blacher et al. [BGK21] strategy of reshaping the input as a matrix, and then applying irregular networks in parallel to the columns of the matrix, rendering them vectorization-friendly. Finally, Dobbelaere [Dob25] maintains a database of (mostly irregular) low depth and low comparison count sorting networks for sizes up to 64.

3 Efficient algorithms for support sampling

As outlined in the introduction, sampling support vectors with minimal bias is a key step in generating n-bit vectors with fixed Hamming-weight ω. In this section, we present three alternative approaches to generate a secure support sampling, described in Algorithm 9, Algorithm 10, and Algorithm 11. Each of these methods, offers noticeable computational advantages over the official HQC procedure (cf. Section 2.2.1), runs in constant time, and produces support vectors that are either unbiased or with negligible bias. However, each of these proposals produces a different output, even if supplied with the same randomness, and thus are incompatible among each other (as well as with Sendrier's shuffle). While this renders it inconvenient to replace step 2 of the HQC sampler so close to standardization, we believe there is value in offering alternatives for future cryptosystems, which are not burdened by such backward compatibility issues.

For a fixed weight ω, we define a support v as a sequence of ω distinct elements from $[n]$, which fully determines a weight-ω vector by enumerating the positions of its non-zero bits. Throughout this section, we define the following sets

$$\mathcal{X}_{n,\omega} = \prod_{i=0}^{\omega-1} [n-i], \qquad \mathcal{P}_{n,\omega} = \{v \in [n]^\omega : \ v \text{ is non-repeating}\},$$
$$\mathcal{S}_{n,\omega} = \{v \in [n]^\omega : \ v \text{ is sorted and non-repeating}\}.$$

Observe that $|\mathcal{P}_{n,\omega}| = |\mathcal{X}_{n,\omega}| = n \cdot (n-1) \cdots (n-w+1)$. All of the methods presented in this section assume that we can perform independent sampling from various ranges of integers, and map these possibly repeating samplings into non-repeating sequences that can be used as supports. These supports are unsorted in the approach described in Section 3.1, but naturally sorted in the strategies presented in Section 3.2 and Section 3.3.

3.1 An alternative to Sendrier's shuffle

Our first proposed algorithm is TLShuffle (Truncated Lehmer Shuffle) in Algorithm 9. It samples a uniformly random Lehmer code, also known as a factorial-base representation, for a length-n permutation [Leh60], and then applies a permutation unranking procedure

Algorithm 9: TLShuffle(n, ω)

Input: parameters n, ω
Output: List of ω unsorted distinct integers in $[n]$

```
1 Initialize an empty list pos of length ω
2 for i = 0 to ω − 1 do
3     pos[i] ← U[n − i]
4 for i = ω − 2 to 0 do
5     for j = i + 1 to ω − 1 do
6         if pos[j] ≥ pos[i] then
7             pos[j] += 1;
8 return pos[0, 1 . . . , ω − 1]
```

to obtain its one-line representation $(\pi(0), \pi(1), \ldots, \pi(n-1))$, where π is the sampled permutation. The algorithm produces a truncated output containing only the first ω elements, as required. An alternative interpretation is that, in order to pick ω distinct values from $[n]$, the algorithm first draws indices pos$[i]$ from $U([n-i])$ for $0 \leq i < \omega$. Conceptually, these are interpreted as indices that pick elements from a shrinking list of numbers in $[n]$ that remain unpicked. In practice, this just means that for each pos$[i]$, all subsequent elements pos$[j]$ that are greater than pos$[i]$ need to be increased by 1.

This algorithm offers an alternative to Sendrier's constant-time shuffle. While it shares the same time complexity and the same statistical bias derived only from the underlying integer-range sampling, it may be preferable on certain architectures due to its simpler implementation.

Theorem 1. *For $1 \leq \omega \leq n$, Algorithm 9 returns an element of $\mathcal{P}_{n,\omega}$ uniformly at random in time $O(\omega^2)$.*

Proof. Let us start by showing that the returned vector belongs to the set $\mathcal{P}_{n,\omega}$, i.e., there are no repetitions in pos$[0, 1 \ldots, \omega - 1]$. This can be seen by noticing that the operation in Line 6 ensures that each pos$[i]$ does not have collisions with pos$[0, 1 \ldots, \omega - 1]$, and that the inner `for`-loop in Line 5 does not generate collisions among the elements that it updates, i.e., in pos$[i + 1, \ldots, \omega - 1]$, since it increases by one all the elements that are greater or equal to pos$[i]$ and leaves unchanged all the others.

Let us prove now that the returned element is distributed uniformly at random over $\mathcal{P}_{n,\omega}$. One can interpret the operation of the `for`-loop in Line 2 as sampling pos$[0, \ldots, \omega - 1]$ uniformly at random in $\mathcal{X}_{n,\omega}$. Recall that $|\mathcal{P}_{n,\omega}| = |\mathcal{X}_{n,\omega}|$. In order to prove that Algorithm 9 returns an element of $\mathcal{P}_{n,\omega}$ with probability $1/|\mathcal{P}_{n,\omega}|$, we need to show that the `for`-loop in Line 4 represents a bijection $\varphi : \mathcal{X}_{n,\omega} \to \mathcal{P}_{n,\omega}$. Since the two sets have the same size, it is enough to show that φ is injective. Assume two distinct lists $c \neq c' \in \mathcal{X}_{n,\omega}$ give the same output under φ. Let k be the largest index with $c[k] \neq c'[k]$. All indices strictly larger than k are processed before index $k + 1$ in the outer loop, so they evolve identically in the two executions with respect to Line 6. Without loss of generality, let us assume that $c[k] < c'[k]$. At index-$k + 1$ of the outer loop the executions will diverge. In the following iterations, it is impossible that the following iteration of the `for`-loop can readjust the position that diverged since any time that $c[k]$ is incremented by 1 in Line 6, the corresponding position in $c'[k]$ gets incremented as well. Hence the final vectors cannot be equal and φ is injective.

Finally, the time complexity of the algorithm is ruled by the subroutine from Line 4 to Line 6 and is $O(\omega^2)$. □

Remark 1 (Implementation note). The structure and time complexity of Algorithm 9 are analogous to those of Algorithm 5, and the core operation inside the inner loop of our implementations use AVX2 compare and bitwise AND instructions for both. However, Algorithm 5 requires a conditional selection operation, which is best implemented using AVX2 variable blend instructions. Unfortunately, these are known to be inefficient on Intel CPUs, requiring 3 μops on the latest microarchitectures. In contrast, Algorithm 9 is naturally implemented using a subtraction, since the output of a 16-bit AVX2 compare instructions is either 0 or `0xFFFF` (i.e. -1 in two's complement arithmetic); this is a simple instruction requiring a single μop. As a result, the execution time of an AVX2 optimized implementation of Algorithm 9 is slightly faster in practice on Intel CPUs. Notice also that the subroutine in Line 2 can be implemented analogously to Sendrier's approach reported in Algorithm 6 as the integers are drawn from intervals of the same lengths as those in Algorithm 5. Hence, the bias estimation is identical to the one by Sendrier [Sen21].

3.2 Bernstein's fixed-weight sampler

In a publicly available Python script[4], Bernstein introduced Algorithm 10, which we call DJBSample, to sample the support of a fixed-weight vector. Like the previous approaches, this method introduces no bias under the assumption that sampling from an integer range is perfectly uniform. Additionally, it achieves lower computational complexity than the method introduced by Sendrier [Sen21], which is currently used in the HQC specification.

DJBSample employs a divide-and-conquer approach, recursively exploiting a factorization of $|\mathcal{S}_{n,\omega}|$ as $|\mathcal{S}_{n,s}| \cdot |\mathcal{S}_{n-s,\omega-s}|$, for $0 < s < \omega$ (which we fix as $s = \lfloor \omega/2 \rfloor$ in Algorithm 10):

$$\binom{n}{\omega} = \binom{n}{s}\binom{n-s}{\omega-s} \tag{1}$$

Conceptually, it places ω weights into n slots by first choosing uniformly s weights to place among all n slots (the list L of Line 4 of Algorithm 10), and choosing uniformly where the remaining $\omega - s$ are placed among the remaining $n - s$ empty slots (the list R of Line 5). The recursive calls output both lists in sorted order, and uniformity is guaranteed by the factorization of Equation (1).

The combination step interleaves these samples into one sorted, collision-free list. It starts by constructing new lists L^+ and R^+ from L and R by tagging them with bits 0 and 1, respectively (Lines 6 and 7), and also subtracting i from the i-th element of L when inserting it into L^+. Thus, we work with pairs

$$L_i^+ = (L_i - i,\, 0) \quad \text{and} \quad R_j^+ = (R_j,\, 1),$$

then we merge the two sorted lists in Line 8 to obtain (y_i, b_i). Let $c_i = \#\{t < i : b_t = 0\}$ be the number of left items seen so far and output

$$M_i \;=\; y_i + c_i.$$

Each right item is incremented for each left item that was placed before it (enforced by the $+ c_i$ in Line 12), while each left item's earlier $- i$ is exactly canceled by having seen i prior left items. Thus M is sorted, collision-free and uniform in $[n]$.

We let merge be a subroutine that takes two sorted arrays as inputs, and outputs a sorted array containing all elements from both input arrays, as employed in the mergesort sorting algorithm. It can be implemented in constant time using the merging networks discussed in Section 2.3.

We present proofs that Algorithm 10 is correct, unbiased, and has running time $O(\omega \log^2 \omega)$, which to the best of our knowledge are not publicly available elsewhere.

[4] https://cr.yp.to/2024/gcww-20240512.py. We found it in a footnote of [FSL24], but were unable to find a matching published paper or preprint about it.

Algorithm 10: DJBSample(n, ω, x)

Input: Integers n, ω with $n \geq \omega > 0$ and a sequence $x \in \mathcal{X}_{n,\omega}$
Output: A sequence in $\mathcal{S}_{n,\omega}$.

if $\omega == 1$ **then**
  return x
$s \leftarrow \lfloor \omega/2 \rfloor, \quad x^{(L)} \leftarrow [x_0, \ldots, x_{s-1}], \quad x^{(R)} \leftarrow [x_s, \ldots, x_{\omega-1}]$
$L \leftarrow$ DJBSample$(n, s, x^{(L)})$
$R \leftarrow$ DJBSample$(n - s, \omega - s, x^{(R)})$
$L_i^+ \leftarrow (L_i - i, 0)$ for $i \in [s]$
$R_i^+ \leftarrow (R_i, 1)$ for $i \in [s, \omega]$
$M^+ \leftarrow$ merge(L^+, R^+)
$c_i \leftarrow |\{(x, b) \in [M_0^+, \ldots, M_{i-1}^+] : \ b = 0\}|$ for $i \in [\omega]$
for $i = 0$ **to** $\omega - 1$ **do**
  $(y_i, b_i) \leftarrow M_i^+$
  $M_i \leftarrow y_i + c_i$
return M

Theorem 2. *For any input $x \in \mathcal{X}_{n,\omega}$, Algorithm 10 returns an element of $\mathcal{S}_{n,\omega}$, and when using a routine that merges two arrays of length s in time $O(s \log s)$ (e.g., a bitonic or odd-even merging network), its complexity is $O(\omega \log^2 \omega)$.*

Proof. We prove the theorem by induction on ω. For the base case $\omega = 1$, the statement trivially holds. Next we assume there exists an ω_0 such that it holds whenever $\omega < \omega_0 \leq n$, and show that it also holds for $\omega = \omega_0 \leq n$. The inductive assumption holds for the recursive calls in Algorithm 10, so they return sorted and non-repeating sequences $L \in [n]^s$, $R \in [n - s]^{\omega - s}$. We now show that M is sorted and non-repeating, and lies in $[n]^\omega$.

First, note that M is a sorted sequence, since M^+ is sorted and addition by the non-decreasing sequence c_i preserves sorting.

Next, we show that the sequence lies in $[n]^\omega$. Note that c_i counts how many of the first i elements in M^+ come from L^+. Hence, if $M_i^+ = (y_i, b_i)$ comes from L^+, then $y_i + c_i$ is the same as the original value in L and must lie in $[n]$. On the other hand, since $c_i \leq |L^+| = s$, and $R_i \in [n - s]$, the values in M coming from R^+ are guaranteed to be in $[n]$. This shows that all of M is in $[n]^\omega$.

To show that M is non-repeating, we assume for the sake of contradiction that $M_i = M_{i+1}$ for some i. Note that if M_i comes from L then $c_{i+1} > c_i$, so we would have $M_i^+ > M_{i+1}^+$, which contradicts M^+ being sorted. On the other hand, if M_i comes from R then $c_{i+1} = c_i$ and M_i^+, M_{i+1}^+ are also equal in their first entry. Therefore, M_{i+1} must also come from R, since otherwise M_{i+1}^+ would have been sorted ahead of M_i^+, but this is still a contradiction since then M_i^+ and M_{i+1}^+ come from repeated elements in R.

Regarding the running time, the algorithm makes two recursive calls, followed by one merging and other linear-time operations, hence the execution time can be written as $[T(w) = T(\lfloor \omega/2 \rfloor) + T(\lceil \omega/2 \rceil) + O(\omega \log \omega)$, and the result follows from an application of the Master theorem [CLRS09].

□

Theorem 3. *The output of Algorithm 10 defines a surjective map $\phi : \mathcal{X}_{n,\omega} \to \mathcal{S}_{n,\omega}$ such that each $y \in \mathcal{S}_{n,\omega}$ has the same number of preimages under ϕ. Thus, if $x \in \mathcal{X}_{n,\omega}$ is uniformly sampled, then $\phi(x)$ is distributed uniformly over $\mathcal{S}_{n,\omega}$.*

Proof. We proceed again by induction on ω, with trivial base case for $\omega = 1$ and $\omega = 2$. Let $\mathcal{V}_{\omega,s}$ consist of all the subsets of $[\omega]$ of size s. We prove that the output of Algorithm 10

is uniform by showing that, for each $y \in \mathcal{S}_{n,\omega}$, there is a bijection between $\mathcal{V}_{\omega,s}$ and the pairs $(L, R) \in \mathcal{S}_{n,s} \times \mathcal{S}_{n-s,\omega-s}$ that produce y as an output of lines 6-13 of the algorithm. This shows that every y has the same number of preimages (L, R) under a single step of the recursive algorithm, which in turn implies, by the induction hypothesis, that every y has the same number of preimages under the entire algorithm.

Fix $y \in \mathcal{S}_{n,\omega}$ and let $V \in \mathcal{V}_{\omega,s}$ be any subset of $[\omega]$ of length s. We partition y into subsequences $L_V \coloneqq \{y_i : i \in V\}$ and $R_V^+ \coloneqq \{y_i : i \notin V\}$. Let $c_i \coloneqq |\{\ell \in L_V : \ell < (R_V^+)_i\}|$ be the number of elements in L_V that are smaller than $(R_V^+)_i$, and define a new sequence R_V given by $(R_V)_i \coloneqq (R_V^+)_i - c_i$. It is easy to see that (L_V, R_V) is mapped to y by lines 6-13 of the algorithm, but it remains to be shown that $(L_V, R_V) \in \mathcal{S}_{n,s} \times \mathcal{S}_{n-s,\omega-s}$ and that $V \mapsto (L_V, R_V)$ is a bijection between $\mathcal{V}_{\omega,s}$ and the preimages of y.

It is clear that $L_V \in \mathcal{S}_{n,s}$, since it is a subsequence of $y \in \mathcal{S}_{n,\omega}$, so we move on to show that $R_V \in \mathcal{S}_{n-s,\omega-s}$. First, since elements in L_V are distinct we have $c_{i+1} - c_i < (R_V^+)_{i+1} - (R_V^+)_i$, which gives $(R_V)_{i+1} - (R_V)_i > 0$, hence R_V is strictly increasing. Next, since $c_i < R_i^+$, we also have that $(R_V)_i = (R_V^+)_i - c_i > 0$. Finally, note that $s - c_i$ is the number of elements from L_V that are greater than $(R_V^+)_i$, so $s - c_i < n - (R_V^+)_i$, which gives $(R_V)_i < n - s$.

We now show that the map $V \mapsto (L_V, R_V)$ is injective. This is straightforward, since L_V is a subset of y, which has no repetitions, so if $V \neq V'$ then L_V and $L_{V'}$ are different subsets. We lastly show that $V \mapsto (L_V, R_V)$ is surjective over all the preimages of y. Indeed, the injectivity of the map shows that each $y \in \mathcal{S}_{n,\omega}$ has at least $\binom{\omega}{s}$ preimages in $\mathcal{S}_{n,s} \times \mathcal{S}_{n-s,\omega-s}$, but since $|\mathcal{S}_{n,s}| \cdot |\mathcal{S}_{n-s,\omega-s}| = \binom{\omega}{s}|\mathcal{S}_{n,\omega}|$, it follows that each y has in fact *exactly* $\binom{\omega}{s}$ preimages, and the map is surjective.

□

3.3 Sota: A Minimalist Approach to fixed-weight Sampling

We have thus far studied three approaches for constructing the support: the existing approach of Sendrier in Algorithm 5, its new alternative TLShuffle that we introduce in Section 3.1, and DJBSample discussed in Section 3.2. A fundamental characteristic shared by all these approaches, is that they work on a sequence of integers sampled from a series of nested decreasing intervals. In return, each produces a collision-free sequence of integers that can serve as the support ω for the output binary vector. Notably, these strategies generate this support without relying on rejection sampling, which, as discussed before, is essential for protecting HQC and BIKE against timing attacks.

In this subsection, we present a novel algorithm that arguably offers *the* simplest constant-time method for constructing the support ω without using rejection sampling. Unlike previous methods, our algorithm takes as input sequences with elements sampled from a single uniform interval, namely $[n - \omega + 1]$. We have dubbed our remarkably compact solution Sota (Algorithm 11), short for "Sort Then Add".

To see that Algorithm 11 outputs a sequence in $\mathcal{S}_{n,\omega}$, we note that adding an increasing sequence bounded by $\omega - 1$ to a sorted sequence of integers not greater than $n - \omega$ produces a sorted sequence of integers not greater than $n - 1$, and since the added sequence is strictly increasing, the output sequence has no repetitions.

Regarding the complexity of Sota, one can easily see that its running time reduces to that of sorting ω elements, i.e., $O(\omega \log^2 \omega)$ when a sorting network such as bitonic sorting is employed. Although this complexity is asymptotically the same as that of DJBSample, the simplicity of the algorithm provides considerable savings in concrete costs.

In [Knu11, §7.2.1.3], Knuth discusses a correspondence between elements of $\mathcal{S}_{n,\omega}$ and integer sequences in $[n-\omega+1]^\omega$ with repetitions, which, in essence, is what Sota implements. Historically, this result was established by Scherk in 1828 [Sch28] and re-discovered eight years later by Förstemann [Fö35].

Algorithm 11: Sota(n, ω, x)

Input: A sequence $x \in [n - \omega + 1]^\omega$.
Output: A sequence in $\mathcal{S}_{n,\omega}$

$[s_0, \ldots, s_{\omega-1}] \leftarrow \mathsf{sort}(x)$
$z_i \leftarrow s_i + i$ for $i = 0, 1, \ldots, \omega - 1$
return $[z_0, \ldots, z_{\omega-1}]$

In exchange for its simplicity and unlike the previously described methods, Sota introduces a measurable bias in the generated supports, even when the integer-range sampling is perfectly uniform. However, as shown by Theorem 4, the bias remains acceptable for secure HQC implementations. Furthermore, Sota's natural simplicity allows us to fully exploit oblivious sorting networks (e.g., `djbsort`) and, crucially, makes it inherently more resistant to fault and side-channel attacks.

Theorem 4. *Let O_1 be a random oracle that returns a uniformly sampled element of $\mathcal{S}_{n,\omega} \coloneqq \{v \in [n]^\omega : v$ is sorted and non-repeating$\}$, and let O_2 be an oracle that samples $x \in [n-\omega+q]^\omega$ uniformly and returns Sota(x). The IND-CCA2 security of the HQC.KEM scheme is smaller when modeling SampleFW with O_2 rather than with O_1, by a factor no greater than*

$$\tau_{max} = \frac{n!}{(n-\omega+1)^\omega (n-\omega)!}.$$

Proof. This follows from [AMAB+25, Section 5.3.1], which shows that an attacker's IND-CPA advantage against HQC.PKE when using a biased distribution D is increased by a factor of

$$\tau_{max} \coloneqq \max_{e \in \mathcal{S}_{n,\omega}} \left(\frac{\Pr[e : e \leftarrow D(\mathcal{S}_{n,\omega})]}{\Pr[e : e \leftarrow U(\mathcal{S}_{n,\omega})]} \right)$$

at most, and the same holds for the IND-CCA2 advantage against HQC.KEM due to the reduction in [AMAB+25, Section 5.2.2]. In the case of Algorithm 11, each output has at most $\omega!$ preimages, since a sorted sequences of length ω has at most $\omega!$ unsorted preimages, and addition by $[0, 1, \ldots, \omega - 1]$ is one-to-one. Therefore, $\Pr[e : e \leftarrow D(\mathcal{S}_{n,\omega})] \leq \omega!/(n-\omega+1)^\omega$ for all e, and

$$\Pr[e : e \leftarrow U(\mathcal{S}_{n,\omega})] = \frac{1}{|\mathcal{S}_{n,\omega}|} = \frac{\omega!(n-\omega)!}{n!}.$$

Therefore, $\tau_{max} = n!/((n-\omega+1)^\omega (n-\omega)!)$ □

In Table 3, we gives concrete values for the bias introduced by both Sota and Rand (Algorithm 6) for the three HQC parameter sets. The involved operations are samplings of weight-ω vectors, used during the key generation procedure, and vectors of weight $\omega_r = \omega_e$. The bias of Sota is computed from the τ_{max} ratio in Theorem 4, whereas the bias from Rand, similarly to [Sen21], is given by $\tau_{max} = (1 + \frac{n-\omega+1-r}{2^B})^\omega$, with $r = 2^B \bmod (n - \omega + 1)$. These two sources of bias are independent, and so the τ_{max} ratios from both distributions combine multiplicatively, meaning that the loss in security bits, computed as $\log_2 \tau_{\max}$, is additive.

4 A novel algorithm for writing the support to a vector

We now discuss the last step of the fixed-weight sampling process: writing out the explicit vector given its support. We assume throughout this section that the support vector

Table 3: Bias introduced by Sota and Rand compared to the uniform distribution.

Instance	Security	ω			$\omega_r = \omega_e$		
		$\tau_{max}^{\mathsf{Sota}}$	$\tau_{max}^{\mathsf{Rand}}$	Bit loss	$\tau_{max}^{\mathsf{Sota}}$	$\tau_{max}^{\mathsf{Rand}}$	Bit loss
hqc-128	NIST-L1	1.1294	1.0001	0.1757	1.1706	1.0001	0.2274
hqc-192	NIST-L3	1.1483	1.0006	0.2005	1.1973	1.0007	0.2607
hqc-256	NIST-L5	1.1595	1.0001	0.2135	1.2112	1.0012	0.2783

Algorithm 12: BitGrouping(v, B)

Input: A sorted support vector v for weight ω, and a base B.

Output: Sequences K, V of length ω representing the non-zero B-bit words of the sampled vector: if $K_i \neq \infty$, then V_i holds the value of the K_i-th word, but if $K_i = \infty$, then the entry is to be ignored. The subsequence of non-∞ entries of K is sorted and non-repeating.

$K_i \leftarrow \lfloor v_i/B \rfloor$ for $i \in [\omega]$
$V_i \leftarrow 2^{v[i] \mod B}$ for $i \in [\omega]$
for i in $[\omega - 1]$ **do**
  $k \leftarrow \infty$
  $v \leftarrow V_i + V_{i+1}$
  $b \leftarrow (K_i == K_{i+1})$
  CSwap(b, v, V_{i+1})
  CSwap(b, k, K_i)
return V, K

is sorted, which incurs an additional cost if the support was sampled using the method from Section 3.1, but not if it was sampled using the method in Section 3.2 or Section 3.3.

Since we want to avoid timing variations due to secret-dependent memory indexing, a naïve solution is to iterate over every bit, writing either a 0 or a 1 that is selected in a constant-time manner (cf. Algorithm 8). We improve on this method by writing the output in batches of B-bit words for some base B. Doing so requires modifying the support vector v to compute the non-zero words from the non-zero bits: for each entry v_i, we compute the word index $K_i = \lfloor v_i/B \rfloor$ and the word value $V_i = 2^{v_i - BK_i}$. In the event that two indices v_i, v_j fall in the same word, the two entries can be accumulated by updating $V_j \leftarrow V_j + V_i$ and setting $K_i \leftarrow \infty$ to mark (K_i, V_i) as disposed values, where ∞ represents a sufficiently large value. Note that the support vector v is assumed to be sorted, so we can assume that word collisions as the one described above can only happen for $j = i + 1$ and we can correct them all in constant time with a single pass through the array. We call this the BitGrouping algorithm, presented as Algorithm 12.

The next step would be to filter out the residual entries that have $K_i = \infty$. However, there is a variable number of them which depends on the sampled vector, so rather than taking them out we just push them to the end of the array in constant time. This can be achieved through any sorting algorithm, provided ∞ is understood to be larger than any other value. However, a full sorting is wasteful since we know the list is already partially sorted and it is only some entries that need to be pushed to the end of the list. This describes a process known as *compaction*: based on an array of flags K, an array V is rearranged such that flagged elements appear at the end and unflagged elements appear in their original order. The authors of [SJG22] have proposed an $O(\omega \log \omega)$ algorithm for compaction in an oblivious setting, which we have adapted for our use case in Algorithm 13. We refer to their work for the proofs of correctness and complexity.

Algorithm 13: Compact(V, K)

Input: Sequences V, K of length ω
Output: Rearranges V, K such that elements with $K_i = \infty$ are pushed to the end of the sequence, and elements with $K_i \neq \infty$ appear in the same order as the original.

1 **if** $\omega == 0$: **return** $\varnothing$
2 $s \leftarrow 2^{\lfloor \log_2(\omega) \rfloor}, \quad s' \leftarrow \omega - s$
3 $V^{(L)} \leftarrow [V_0, V_1, \ldots, V_{s'-1}], \quad V^{(R)} \leftarrow [V_{s'}, V_{s'+1}, \ldots, V_{\omega-1}]$
4 $K^{(L)} \leftarrow [K_0, K_1, \ldots, K_{s'-1}], \quad K^{(R)} \leftarrow [K_{s'}, K_{s'+1}, \ldots, K_{\omega-1}]$
5 $m \leftarrow |\{i \in [s'] : \ K_i \neq \infty\}|$
6 Compact$(V^{(L)}, K^{(L)})$
7 OffCompact$(V^{(R)}, K^{(R)}, (s - s' + m) \mod s)$
8 **for** i in $[s']$ **do**
9 CSwap$(i \geq m, V_i, V_{i+s})$
10 CSwap$(i \geq m, K_i, K_{i+s})$

We then need to add the zero words that are missing, which must be intertwined with the non-zero words in the correct order. This is achieved through the MergeWords algorithm (Algorithm 15), whose correctness we prove in Proposition 1. Note that the merge routine is assumed to produce a sorted output, and triplets are sorted by comparing the left-most entries first.

Remark 2. By an abuse of notation, line 2 of Algorithm 15 assumes that if $K_i = \infty$ then $K_i - i = \infty$. In practice, ∞ represents a large but finite value, so line 2 needs to perform the subtraction by i conditionally on $K_i \neq \infty$.

Proposition 1. *Algorithm 15 is correct.*

Proof. Let $x_i = (K_i - i, i, V_i)$ be an element in L for which $K_i \neq \infty$. Since K is sorted, there are i elements from L that are ahead of x_i. On the other hand, $(j, \infty, 0)$ gets sorted ahead of x_i if and only if $j < K_i - i$, so $K_i - i$ elements from R are ahead of x_i. Therefore, there are a total of $i + (K_i - i) = K_i$ elements ahead of x_i in the merged list, and V_i occupies the K_i-th place in the output, as desired. Since all N element of R are sorted ahead of triplets (k, i, v) where $k = \infty$, the first N elements of M contain no triplets with $k = \infty$, so all other elements of the output are zeros. □

Algorithm 15 produces the desired vector split into B-bit words, which still need to be rearranged into words of the appropriate size depending on the architecture used. The MWSupport algorithm (Algorithm 16) puts together all the previous algorithms and performs the entire process of writing the support.

Time complexity. The complexity of BitGrouping is $O(\omega)$, whereas Compact has a complexity of $O(\omega \log \omega)$ (see [SJG22]). The cost of MergeWords is dominated by the cost of the merging routine, which is $O(n \log n)$ when implemented with *e.g.* bitonic or odd-even merging networks, and the final step of converting between word sizes can be done in $O(n)$. Therefore, the entire MWSupport algorithm has an overall complexity of $O(n \log n)$.

The choice of B. In an optimized implementation, the (k, i, v) triplets from Algorithm 15 should be concatenated into a single processor word to facilitate comparisons and rearrangements during merging. We need to allocate at least $\lceil \log_2(n/B + 1) \rceil$ bits for k to

Algorithm 14: $\mathsf{OffCompact}(V, K, z)$
Auxiliary routine for Compact. Performs the compaction for size ω a power of 2 with a cyclic offset by $z < \omega$ in both V and K.

if $\omega == 2$ **then**
  $b \leftarrow (z == 1) \oplus (K_0 == \infty$ **and** $K_1 \neq \infty)$
  $\mathsf{CSwap}(b, V_0, V_1)$
  $\mathsf{CSwap}(b, K_0, K_1)$
else if $\omega > 2$ **then**
  $s \leftarrow \omega/2$
  $V^{(L)} \leftarrow [V_0, V_1, \ldots, V_{s-1}], \quad V^{(R)} \leftarrow [V_s, V_{s+1}, \ldots, V_{\omega-1}]$
  $K^{(L)} \leftarrow [K_0, K_1, \ldots, K_{s-1}], \quad K^{(R)} \leftarrow [K_s, K_{s+1}, \ldots, K_{\omega-1}]$
  $m \leftarrow |\{i \in [s] : \ K_i \neq \infty\}|$
  $\mathsf{OffCompact}(V^{(L)}, K^{(L)}, z \mod s)$
  $\mathsf{OffCompact}(V^{(R)}, K^{(R)}, (z+m) \mod s)$
  $b_1 \leftarrow [((z \mod s) + m) \geq s] \oplus [z \geq s]$
  for i in $[s]$ **do**
    $b_2 \leftarrow b_1 \oplus [i \geq ((z+m) \mod s)]$
    $\mathsf{CSwap}(b_2, V_i, V_{i+s})$
    $\mathsf{CSwap}(b_2, K_i, K_{i+s})$

Algorithm 15: $\mathsf{MergeWords}(V, K, B)$

Input: Sequences V, K of length ω output by Compact, and a base B.
Output: The full list of B-bit words for the vector with the prescribed support.

$N \leftarrow \lceil n/B \rceil$
$L_i \leftarrow (K_i - i, i, V_i)$ for $i \in [\omega]$
$R_i \leftarrow (i, \infty, 0)$ for $i \in [N]$
$M \leftarrow \mathsf{merge}(L, R)$
for i in $[N]$ **do**
  $(k, i, v) \leftarrow M_i$
  $R_i \leftarrow v$
return R

enumerate all possible words plus the ∞ value, and $\lceil \log_2(\omega+1) \rceil$ bits for i to enumerate the possible choices in $[\omega]$ plus the ∞ value. The last entry contains the B-bit word, so we obtain the bound $\lceil \log_2(n/B+1) \rceil + \lceil \log_2(\omega+1) \rceil + B \leq W$, where W is the processor word length. For $W = 64$, we obtain $B \leq 48, 47, 45$ for HQC security levels $1, 3, 5$, respectively. On the other hand, translating between B-bit words and W-bit words is simpler when $\mathrm{lcm}(B, W)$ is small, so we have found $B = 48, 40, 40$ to be a good tradeoff.

5 Experimental results

We implemented the algorithms of Section 3 and Section 4 in both reference and AVX2-optimized versions. The reference version prioritize clarity, while the AVX2 versions demonstrate the performance achievable by the proposed techniques. Our code is available under a permissive open source license at https://github.com/Crypto-TII/sota-voce.

Recall that the performance of our HQC implementation is dominated by MWSupport, with its $O(n \log n)$ running time. The algorithm of [FSL24], designed for the dense regime,

Algorithm 16: MWSupport (v, B, W)

Input: A support vector consisting of ω sorted distinct indices in $[n]$, a base B, and a natural word size W.
Output: W-bit word representation of the sampled vector.

```
V, K ← BitGrouping(v, B)
Compact(V, K)
R ← MergeWords(V, K, B)
Convert R from B-bit words to W-bit words.
return R
```

claims $O(n)$ performance, leading us to question whether implementing [FSL24] for HQC could yield a net performance win. In the sparse regime with Hamming weight ω, algorithms for step 2, such as Algorithm 7 and those of Section 3, consume only ω pseudorandom integers of at most n bits each, whereas [FSL24] consumes n such integers. For HQC, where $\omega = O(\sqrt{n})$, this is a $O(\sqrt{n})$ increase in PRNG demand relative to ours. The resulting PRNG throughput requirements typically dominate the $\log n$ arithmetic term, making [FSL24] an unlikely contender for performance improvement at HQC sizes.

Concretely, Algorithm 5 of [FSL24] generates randomness for its shuffle by drawing an array of integers reduced modulo $n-1, n-2, \ldots, 2$. This yields $(n-1)\cdot(n-2)\cdots 2 = (n-1)!$ possible arrays, so the information-theoretic minimum is $\lg((n-1)!)$ random bits – in practice, more are needed to mitigate modulo-induced bias. For hqc-128 with $n = 17669$, we have $\lg((n-1)!) \approx 224$ kbits $= 28$ kB. Given the Keccak team's reported SkylakeX performance for SHAKE256 (as employed by HQC's seed expander) of 5.49 cycles/byte [BDH+25], this amounts to a lower bound of

$$28{,}000 \text{ bytes} \times 5.49 \text{ cycles/byte} \approx 154{,}000 \text{ cycles}$$

solely to generate randomness for the shuffle, excluding the shuffling itself. This already exceeds not only the cost of our fixed-weight sampling but also the full key generation / encapsulation costs for hqc-128 (see Table 1). Repeating the calculation at other security levels leads to the same conclusion, which supports our decision not to implement [FSL24] as a baseline for HQC.

5.1 Benchmarking methodology

We benchmark our implementations on an Intel Core i7-13700k CPU (13th generation, codenamed "Raptor Lake"), specifically on its performance cores with microarchitecture codename "Raptor Cove", with nominal core clock of 3.4 GHz. The machine runs Ubuntu Linux 25.04, and all code is compiled with Clang version 18.1.8 using flags `-O3 -march=native -mtune=native -mavx2`. Although we produced and make available both AVX2-optimized and non-optimized code, our benchmarks refer only to the optimized version. This choice is justified by the fact that only the optimized implementation of HQC [HQC25], which serves as our comparison baseline, is constant-time.

We take the usual precautions to reduce noise, errors and biases in our measurements: disable HyperThreading and TurboBoost, set Linux's scaling governor to `performance` mode to avoid downclockings, and ensure adequate cooling to avoid thermal throttling during benchmark runs. We also use Linux's `cgroups` feature to isolate one core from the OS scheduler, and benchmarks are executed exclusively on this core; this prevents context switches during benchmark execution. Benchmark results were obtained by averaging over 1000 randomly generated instances. Each instance was repeated 100 times and the results were averaged. We measure CPU cycle counts with the aid of `libcpucycles`[5].

[5] Available at https://cpucycles.cr.yp.to.

5.2 Benchmarking of standalone algorithms

Tables 4 and 5 present performances comparison between our AVX2 optimized implementations of the algorithms introduced in Sections 3 and 4 respectively and the corresponding subroutines used in the official HQC optimized implementation [HQC25].

Vectorizing Sendrier's shuffle. In addition to the algorithms from Section 3, we include a custom AVX2-optimized implementation of Sendrier's shuffle [Sen21] (see Line 3 of Algorithm 5), developed for this work, which we refer to as SendrierAVX2. Both this implementation, and that of TLShuffle, were inspired by the implementation of permutation unranking found in [BBPF24]. The vectorized version batches multiple candidate positions per iteration and manipulates them in 256-bit registers, reducing the number of inner loop iterations by a factor of up to $256/16 = 16$ relative to the scalar reference code (given that we employ an array of 16-bit elements for the support). We used a very compact instruction sequence to express the core conditional selection operation of Sendrier's shuffle, while retaining constant-time execution, crucial both for vectorization efficiency and side-channel resilience. HQC's optimized implementation does not include AVX2 support for this subroutine, so our vectorized Sendrier implementation is itself a contribution of this work.

Discussion. Looking at Table 4, it appears that TLShuffle is the preferable choice among the unbiased algorithms. However, this timing does not account for the fact that TLShuffle outputs an unsorted version of the support. Therefore, when combined with the faster Algorithm 8 for step 3, an additional sorting is required, which offsets this advantage. For this reason, the last column of Table 5 reports the cost of MWSupport with the addition of the sorting routine to be used when the input is not already sorted.

Overall, our benchmarks demonstrate that all proposed algorithms yield performance improvements ranging from 5.1× to 18× over the original implementation of Sendrier's shuffle, and from 2.7× to 6.3× over the AVX2-optimized support-writing routine from [HQC25]. However, such large speedups cannot be fully attributed to the performance characteristics of Sendrier's algorithm itself, and are instead representative of the specific code found in HQC's official implementation, and the inability of the compiler to autovectorize it.

Table 4: Cycle counts of proposed shuffling algorithms against the corresponding routine in [HQC25]. Values in parentheses indicate relative improvements.

(n, ω)	[HQC25]	SendrierAVX2	TLShuffle	DJBSample	Sota
$(17669, 66)$	3301	544 (6.1×)	455 (7.3×)	564 (5.9×)	365 (9.0×)
$(17669, 75)$	3769	691 (5.5×)	555 (6.8×)	606 (6.2×)	351 (11×)
$(35851, 100)$	5337	868 (6.1×)	972 (5.5×)	861 (6.2×)	355 (15×)
$(35851, 114)$	6155	1016 (6.1×)	1197 (5.1×)	862 (7.1×)	347 (18×)
$(57637, 131)$	7455	1243 (6.0×)	883 (8.4×)	1165 (6.4×)	555 (13×)
$(57637, 149)$	8538	1495 (5.7×)	1083 (7.9×)	1354 (6.3×)	566 (15×)

5.3 Impact on HQC optimized implementations

We implement a variant of HQC for each algorithm presented in Section 3, reporting its performance only using our AVX2-optimized version of MWSupport from Section 4, due to its clear performance improvement over HQC's naïve implementation for the same step, and the fact that its use does not affect KATs. In addition, we include a variant that employs our AVX2-optimized implementation of Sendrier's shuffle and that is KAT-compliant regardless of the algorithm used to write the support.

Table 5: Cycle counts of the write-support algorithm from Section 4 against the corresponding routine in [HQC25]. Values in parentheses indicate relative improvements.

(n, ω)	[HQC25]	MWSupport	MWSupport + sort
$(17669, 66)$	6827	2262 (3.0×)	2543 (2.7×)
$(17669, 75)$	7753	2398 (3.2×)	2584 (3.0×)
$(35851, 100)$	19980	4559 (4.4×)	4950 (4.0×)
$(35851, 114)$	22691	4907 (4.6×)	5095 (4.5×)
$(57637, 131)$	41210	7043 (5.9×)	7512 (5.5×)
$(57637, 149)$	46850	7393 (6.3×)	7873 (6.0×)

The results of the protocol-level benchmarks are reported in Table 1. We observe improvements in the range of 1.28× to 1.37×, 1.21× to 1.28× and 1.15× to 1.21× for HQC key generation, encapsulation and decapsulation, respectively; these improvements generally increase for higher security levels, due to the asymptotic advantage of the proposed algorithms, particularly Algorithm 16. Amdahl's law limits the potential for improvements of a particular step (in our case, fixed-weight sampling) on the overall performance of the scheme; our performance profiling of HQC's original implementation indicates that approximately 30%, 33% and 15% of execution time is spent on fixed-weight sampling for key generation, encapsulation and decapsulation, respectively, in the parameter set hqc-128. In light of this limitation, the improvements that we present are about as good as one could expect for the sampling subroutines of HQC.

6 Conclusion

In March 2025, NIST selected HQC for post-quantum cryptographic standardization, confirming its position as a robust, code-based alternative to lattice-based KEMs. As HQC moves toward widespread adoption, enhancing the performance and side-channel resilience of its core components becomes increasingly important. In this work, we presented a set of algorithmic improvements targeting the sampling of sparse fixed-weight binary vectors, a fundamental and performance-critical subroutine in HQC. Our optimizations focus on Steps 2 and 3 of the sampling procedure description specified in the Introduction, namely, support sampling and writing the support to the output binary vector, respectively. These two processes handle sensitive data, either secret key components or ephemeral randomness, and must therefore run in constant time to resist timing side-channel attacks.

For Step 2, which samples the support, we contribute with a constant-time AVX2-vectorized implementation of Sendrier's shuffle, introduce a new Lehmer code–based sampler, incorporate Bernstein's efficient and unbiased algorithm, and propose a lightweight, practical and relatively unbiased alternative dubbed Sota. Furthermore, we report in Table 1 the execution time improvements of these proposed procedures for support sampling.

For Step 3, which writes the support to the binary output vector, we introduced a novel support-writing algorithm that reduces the complexity from $O(n\omega) = O(n^{\frac{3}{2}})$ to $O(n \log n)$. This complexity reduction yields a noticeable performance boost, clearly reflected in the timing results reported in Table 1. Indeed, our experimental results confirm that these improvements deliver significant speedups in isolation (up to 18× for step 2 and 6.3× for step 3) and also in the overall HQC scheme. For hqc-256, our best-performing combinations reduce the runtime of key generation, encapsulation, and decapsulation by approximately 27%, 22%, and 17% respectively, compared to the official optimized implementation.

By providing faster, side-channel-secure implementations of these critical steps, our work

strengthens HQC's practical viability and contributes to the broader effort of deploying efficient, quantum-safe cryptography. Our code is publicly available under a permissive open-source license to support further research, verification, evaluation, and adoption.

6.1 Future work

We believe there might be room for improvements on Step 1 of the sampling procedure as well. Sendrier [Sen21] analyzed the approach of Algorithm 6 and proved that the bias it introduces is negligible. We leave as a future research direction to investigate the existence of more efficient methods in terms of randomness consumption and whose bias is still negligible. Our own profiling indicates that the SHAKE256-based seed expander takes up a significant amount of time of the sampling routine, and by reducing randomness consumption, it might be possible to obtain further relevant speedups.

Another avenue for future work is implementing these techniques for the BIKE [ABB+22] and Classic McEliece [ABC+22] cryptosystems. Due to the similar asymptotics and concrete parameter choices in BIKE compared to HQC, we are confident that similar speedups can be obtained for BIKE. The performance impact of our algorithms is somewhat less clear in the case of Classic McEliece, since it adopts $\omega = O(n/\log n)$, and concrete choices of n are about an order of magnitude smaller than in HQC. As such, we believe a fully optimized implementation of our techniques is required to quantify their performance impact on Classic McEliece.

References

[AAB+22] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jerome Lacan, Jean-Marc Robert, and Pascal Veron. HQC. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[ABB+22] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar-Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zémor, Valentin Vasseur, Santosh Ghosh, and Jan Richter-Brokmann. BIKE. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/Projects/post-quantum-cryptography/round-4-submissions.

[ABC+22] Martin R. Albrecht, Daniel J. Bernstein, Tung Chou, Carlos Cid, Jan Gilcher, Tanja Lange, Varun Maram, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Kenneth G. Paterson, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, Cen Jung Tjhai, Martin Tomlinson, and Wen Wang. Classic McEliece. Technical report, National Institute of Standards and Technology, 2022. available at https://csrc.nist.gov/projects/post-quantum-cryptography/round-4-submissions.

[AKS83] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, page 1–9, New York, NY, USA, 1983. Association for Computing Machinery.

[AMAB+25] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jurjen Bos, Jean-Christophe Deneuville, Arnaud Dion, Philippe Gaborit, Jérôme Lacan, Edoardo Persichetti, Jean-Marc Robert, Pascal Véron, and Gilles Zémor. Hamming Quasi-Cyclic (HQC), 4th round official documentation. https://web.archive.org/web/20250710111615/https://pqc-hqc.org/doc/hqc-specification_2025-02-19.pdf, February 2025.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[BBC+20] Daniel J. Bernstein, Billy Bob Brumley, Ming-Shing Chen, Chitchanok Chuengsatiansup, Tanja Lange, Adrian Marotzke, Bo-Yuan Peng, Nicola Tuveri, Christine van Vredendaal, and Bo-Yin Yang. NTRU Prime. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[BBPF24] Slim Bettaieb, Alessandro Budroni, Marco Palumbi, and Décio Luiz Gazzoni Filho. Quasi-optimal permutation ranking and applications to PERK. In Serge Vaudenay and Christophe Petit, editors, *AFRICACRYPT 24*, volume 14861 of *LNCS*, pages 49–65. Springer, Cham, July 2024.

[BCLvV17] Daniel J. Bernstein, Chitchanok Chuengsatiansup, Tanja Lange, and Christine van Vredendaal. NTRU prime: Reducing attack surface at low cost. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017*, volume 10719 of *LNCS*, pages 235–260. Springer, Cham, August 2017.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 250–272. Springer, Berlin, Heidelberg, August 2013.

[BDH+25] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Software performance figures. https://keccak.team/sw_performance.html, 2025.

[Ber19] Daniel J. Bernstein. djbsort software library. https://sorting.cr.yp.to/, 2019.

[BGK21] Mark Blacher, Joachim Giesen, and Lars Kühne. Fast and robust vectorized in-place sorting of primitive types. In David Coudert and Emanuele Natale, editors, *19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[CDH+20] Cong Chen, Oussama Danba, Jeffrey Hoffstein, Andreas Hülsing, Joost Rijneveld, John M. Schanck, Peter Schwabe, William Whyte, Zhenfei Zhang, Tsunekazu Saito, Takashi Yamakawa, and Keita Xagawa. NTRU. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 3rd edition, 2009.

[Dob25] Bert Dobbelaere. Smallest and fastest sorting networks for a given number of inputs. https://bertdobbelaere.github.io/sorting_networks_extended.html, 2025.

[FIP24a] Module-Lattice-Based Key-Encapsulation Mechanism Standard. National Institute of Standards and Technology, NIST FIPS PUB 203, U.S. Department of Commerce, August 2024. https://doi.org/10.6028/NIST.FIPS.203.

[FIP24b] Module-Lattice-Based Digital Signature Standard. National Institute of Standards and Technology, NIST FIPS PUB 204, U.S. Department of Commerce, August 2024. https://doi.org/10.6028/NIST.FIPS.204.

[FIP24c] Stateless Hash-Based Digital Signature Standard. National Institute of Standards and Technology, NIST FIPS PUB 205, U.S. Department of Commerce, August 2024. https://doi.org/10.6028/NIST.FIPS.205.

[FO99] Eiichiro Fujisaki and Tatsuaki Okamoto. Secure integration of asymmetric and symmetric encryption schemes. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 537–554. Springer, Berlin, Heidelberg, August 1999.

[FSL24] Décio Luiz Gazzoni Filho, Tomás S. R. Silva, and Julio Cesar López-Hernández. Efficient isochronous fixed-weight sampling with applications to NTRU. *CiC*, 1(2):14, 2024.

[FY38] Ronald A. Fisher and Frank Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. Oliver & Boyd, London, 6th ed. edition, 1938.

[Fö35] W.A. Förstemann. Einfacher beweis eines satzes der combinationslehre. *Journal für die reine und angewandte Mathematik*, 13:237–239, 1835.

[GHJ+22] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder. Don't reject this: Key-recovery timing attacks due to rejection-sampling in HQC and BIKE. *IACR TCHES*, 2022(3):223–263, 2022.

[HHK17] Dennis Hofheinz, Kathrin Hövelmanns, and Eike Kiltz. A modular analysis of the Fujisaki-Okamoto transformation. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part I*, volume 10677 of *LNCS*, pages 341–371. Springer, Cham, November 2017.

[HQC25] HQC Team. Optimized Implementation of HQC. PQC-HQC website, February 2025. Version dated 19-02-2025; accessed 15-07-2025. Available at https://web.archive.org/web/20250710111449/https://pqc-hqc.org/implementation.html.

[HS82] Zhu Hong and Robert Sedgewick. Notes on merging networks (preliminary version). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, page 296–302, New York, NY, USA, 1982. Association for Computing Machinery.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Boston, 2nd edition, 1997.

[Knu11] Donald E. Knuth. *The Art of Computer Programming, Volume 4A: Combinatorial Algorithms, Part 1*. Addison-Wesley Professional, third edition, 2011.

[Leh60] Derrick H Lehmer. Teaching combinatorial tricks to a computer. *Combinatorial Analysis*, pages 179–193, 1960.

[Par92] Ian Parberry. The pairwise sorting network. *Parallel Processing Letters*, 02(02n03):205–211, 1992.

[Sch28] H.F. Scherk. Aufgaben und lehrsätze, erstere aufzulösen, letztere zu beweisen. *Journal für die reine und angewandte Mathematik*, 3:96–100, 1828.

[Sen21] Nicolas Sendrier. Secure sampling of constant-weight words – application to BIKE. Cryptology ePrint Archive, Report 2021/1631, 2021.

[SJG22] Sajin Sasy, Aaron Johnson, and Ian Goldberg. Fast fully oblivious compaction and shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2565–2579, New York, NY, USA, 2022. Association for Computing Machinery.

[VV71a] David C. Van Voorhis. A generalization of the divide-sort-merge strategy for sorting networks. Technical report, Stanford University, Stanford, CA, USA, 1971.

[VV71b] David C. Van Voorhis. Large [g,d] sorting networks. Technical report, Stanford University, Stanford, CA, USA, 1971.