Sotto Maschera

Masking the Sota Voce Fixed-Weight Vector Sampling

Ben Brooksnieder[1], Dina Hesse[1], Markus Krausz[2] and Tim Güneysu[1,3]

[1] Horst Görtz Institute for IT Security, Ruhr University Bochum, Bochum, Germany, firstname.lastname@rub.de
[2] TÜVIT, Essen, Germany, m.krausz@tuvit.de
[3] DFKI GmbH, Bremen, Germany

Abstract. Sparse fixed-weight vector sampling is a highly critical primitive in several post-quantum cryptography schemes. It has also been the target of multiple side-channel attacks, demonstrating the need for robust implementations. While such implementations do exist, they are not necessarily practical in terms of runtime, and some have already been shown to be insecure. At CHES 2026, Filho *et al.* presented a runtime-optimized, but not side-channel protected, sparse fixed-weight vector sampling algorithm named *Sota Voce*. It consists of the *Sota* support sampler and *MWSupport* support-to-vector conversion, where the latter is planned to be included in the next HQC implementation.
This work explores how both parts of *Sota Voce* can be transformed into a power side-channel protected implementation using masking gadgets for embedded devices. At the same time, we aim to preserve its competitive performance advantage over other masked implementations. By bitslicing core components of the algorithm, including sorting, merging, and compaction, we achieve high parallelism in the computation, improving the overall runtime. Moreover, we avoid slow masking conversions by masking exclusively in the Boolean domain. Thereby, our masked implementation is orders of magnitude faster than the state-of-the-art, while showing no power side-channel leakage in TVLA evaluations. For instance, for HQC, we achieve up to a 45× speedup in comparison to previous work.

Keywords: Side-Channel Analysis · Masking · HQC · FWVS

1 Introduction

Back in 2017, NIST started the selection process for Post-Quantum Cryptography (PQC) in preparation for the advent of large quantum computers that could break today's established asymmetric cryptography. In 2026, several PQC schemes were standardized or selected for standardization by NIST, ISO and other agencies. Most of these schemes are code- or lattice-based, as this allows a practical balance between performance and the sizes of keys, signatures and ciphertexts. In particular, NIST standardized the lattice-based ML-KEM [oST24b] and ML-DSA [oST24a], ISO standardized the code-based Classic McEliece [ABC+17][1], and NIST selected the code-based Hamming Quasi-Cyclic (HQC) [GMA+25] for standardization.

Consequently, research on PQC gradually became more focused on specific implementation challenges. Although optimized implementations addressing timing side-channel leakage exist for most relevant schemes, this is not yet the case for stronger attacker

[1] Note that the German Federal Office for Information Security (BSI) has stopped recommending Classic McEliece in September 2026, due to cryptanalytic improvements.

models with access to power side-channels. This is a growing concern as embedded devices featuring cryptographic implementations, such as passports and banking cards, must soon transition to PQC and be resistant against these attacker models. Therefore, the development of practical, power side-channel-secure PQC implementations is a timely issue. For ML-KEM and ML-DSA, the industry already offers certified, side-channel-secure PQC implementations [fSidI24] that rely on masking [ISW03], which is based on the strong formal foundation of secret sharing.

Common implementation aspects of code- and lattice-based PQC that are critical to performance are polynomial arithmetic and random samplers. Fortunately, polynomial multiplication is a linear operation in the arithmetic masking domain, so efficient algorithms (e.g., NTT) can be applied share-wise. However, random sampling algorithms are usually non-linear and require novel solutions to achieve efficient *and* side-channel secure implementations. For example, CDT-based samplers are commonly used for Gaussian sampling, and masked implementations have already been the focus of multiple recent publications [KLS+25, EGRZ26, HEHT26].

Another common type of random samplers generates uniform random binary or ternary polynomials with a fixed number of non-zero coefficients, commonly named Fixed-Weight Vector Sampling (FWVS). The two major parameters are the weight ω, which refers to the number of non-zero coefficients and n, the number of coefficients. Code-based PQC, like HQC, BIKE, and Classic McEliece, typically sample *sparse* fixed-weight vectors where n is on the order of 10^3 to 10^4 whereas $\omega = O(\sqrt{n})$ is significantly smaller. This results in high performance requirements for the sampler, as, for instance, a third of the computation time of HQC's encapsulation in the reference implementation is spent with FWVS [FAB+26a].

Additionally, in many schemes, the sampled vectors are secret values, so the sampler must be protected against side-channel attacks. In some PQC applications, the input randomness for the sampler is the output of a Pseudorandom Number Generator (PRNG) that is seeded by a sensitive value. Therefore, the side-channel leakage of the sampler must also be independent of randomness that does not directly contribute to the final output vector. For instance, Guo *et al.* [GHJ+22] successfully attacked HQC through a seed-dependent timing variation which led to specification changes for HQC [GMA+25], BIKE [ABB+22], and SMAUG-T [CCSS24].

To counter this threat, Krausz *et al.* [KLRG23] developed bitsliced masked implementations of multiple FWVS algorithms. They demonstrated that the most efficient solution varies by scheme and that masking FWVS results in notable overall performance degradation. Later, Filho *et al.* [FAB+26a] presented a novel approach for FWVS named *Sota Voce* that significantly outperforms other samplers for sparse vectors in an unmasked application. They accelerate FWVS at two stages: generating the support vector and converting it to a bit vector. The conversion, in particular, greatly improves the state-of-the-art and is universally applicable, even in combination with other support samplers. Moreover, it is seed-independent and therefore not subject of standardization, allowing simple integration for many schemes. That is why this conversion will become part of the official HQC implementation [FAB+26b].

However, [FAB+26a] does not discuss power analysis resilience nor potential countermeasures for *Sota Voce*. Yet, masking is generally applicable to any algorithm that can be expressed using Boolean operations, therefore it is a logical countermeasure for protecting *Sota Voce*. Still, straightforward replacement with masking gadgets typically incurs significant performance overheads. Moreover, algorithmic improvements do not necessarily reproduce in a masked implementation, as non-linear operations like comparisons lack native hardware acceleration.

1.1 Our Contributions

In this work, we transfer the algorithmic improvements of *Sota Voce* [FAB+26a] into a side-channel resilient implementation using masking gadgets by Bronchain *et al.* [BC22]. We statistically verify the masking effectiveness of our implementation, which is publicly available[2], with exhaustive Test Vector Leakage Assessment (TVLA) analysis with 100,000 traces that show no first-order leakage. Moreover, we reach great performance improvements over current state-of-the-art: By significantly reducing costly non-linear operations and avoiding slow masking conversions, we circumvent typical performance bottlenecks introduced by masking. In addition, we deploy a bitsliced bitonic sorting and merging network and bitslice a compaction algorithm to parallelize computation, which results in better performance. In comparison with other masked sparse FWVS, we achieve a speedup of up to 6.35× on selected parameters for the support sampling. To an even greater extent, the performance increase for the subsequent support-to-vector conversion reaches up to 84.5×. The combination of both results in an overall performance increase by a factor of 45. Moreover, the components on their own are also deployable.

1.2 Related Work

Before *Sota Voce* [FAB+26a], there already existed several algorithmic approaches for FWVS in the scientific literature. [Ber18] proposed FWVS via shuffling of the fixed-weight vector based on sorting. [DG19] presented RepeatedAND, a FWVS algorithm that directly operates on a bitstring and manipulates the weight with bitwise `AND` operations, which [KLRG23] generalized in an algorithm named Comparison Sampling. [Sen21] introduced a constant-time Fisher-Yates for FWVS and [FSL24] developed another method based on shuffling. These FWVS algorithms differ in output representation and some introduce a small bias in the uniform random sampling distributions (which may be acceptable for some applications). Note that FWVS algorithms for binary polynomials can be easily transformed to output ternary polynomials.

[KLRG23] developed bitsliced masked implementations of multiple FWVS algorithms. Instead of masking the low-level operations of FWVS, [CGTZ23] proposed to split the random permutation into shares. However, [HFG+25] showed that this approach is not necessarily secure in practice. Masked FWVS based on Fisher-Yates has been developed for BIKE [DR24] and for HQC [SVP+24], while the latter has been shown to be vulnerable to power side-channel attacks in [HKM+26]. Masked FWVS, CDT-based Gaussian sampler, and other common primitives can be accelerated with hardware support for a small set of common low-level instructions [KLS+25].

2 Preliminaries

Throughout this paper, we use the notation $[n]$ to describe the exclusive interval $[0, n) = \{0, 1, \ldots, n-1\}$. Moreover, for a k-bit variable a we use $a[i]$ to address the i-th bit of a.

2.1 Fixed-Weight Vector Sampling

The process of sampling random vectors of length n with a fixed Hamming weight ω is called FWVS. There is a distinction between *dense* FWVS, where $\omega = O(n)$, and *sparse* FWVS with $\omega = O(\sqrt{n})$. Both come with different operational constraints and there is no universally efficient sampling algorithm. In this work, we focus on sparse FWVS, as required for code-based PQC.

[2] https://anonymous.4open.science/r/masked_sota_voce-7E24/

The *support* of a fixed-weight vector is the set of the non-zero bit positions. Thus, it contains exactly ω distinct entries. Sparse FWVS is typically organized into three steps [FAB+26a]:

Step 1: Sample integers. Sample ω random integers X to initialize the vector support.

Step 2: Construct the support. Map the sampled integers to ω distinct entries in $[n]$. If necessary, reject integers and resample.

Step 3: Convert to bit vector. Create the final n-bit vector by setting the support positions to 1 and all remaining positions to 0 in a process called support-to-vector conversion.

Typically, only Steps 1 and 2 are part of specifications, as these are seed-dependent. In contrast, Step 3 is deterministic and requires no dedicated specification. For instance, the current HQC specification includes a reference implementation of Steps 1 and 2 [GMA+25]. It evolved from an earlier version that used rejection in Step 2 to verify a uniformly distributed support. However, this rejection introduced a timing side-channel that is exploitable [GHJ+22]. To address this vulnerability, Sendrier proposed a rejection-free algorithm [Sen21], which was adopted in the official HQC specification. This countermeasure removes the timing vulnerability but introduces a small distributional bias, which is argued to be negligible for the security of HQC [GMA+25]. Although the attack by Guo *et al.* [GHJ+22] targets timing leakage, the underlying issue is more general: any side channel that allows distinguishing between sampling based on two different seeds, such as memory-access patterns and power consumption, may be exploitable to attack code-based PQC [HKM+26].

Sota Voce. Given the long runtime of FWVS and its repeated use in code-based PQC, Filho *et al.* [FAB+26a] introduced *Sota Voce*, a new approach to sparse FWVS. They improve the support sampling (Step 2) by using a sorting procedure to guarantee that ω distinct entries are generated. Thereby, they avoid rejection and provide a constant-time alternative for Sendrier's proposed sampler. Although this algorithm named *Sota* is faster in comparison, in the case of HQC it is not specification-compliant. In contrast, the proposed algorithm for the support-to-vector conversion (Step 3), named *MWSupport*, is independent and universally applicable, improving the performance up to 6× in comparisons with the reference implementation of HQC.

2.2 HQC

HQC [GMA+25] is a code-based PQC scheme that was selected for standardization by NIST in 2025. Therefore, it is a relevant scheme that requires sparse FWVS and is representative of other code-based PQC with sparse FWVS in this work. It is based on the Quasi-Cyclic Syndrome Decoding problem and is built upon a Public-Key Encryption (PKE) algorithm that can be transformed into a Key Encapsulation Mechanism (KEM) scheme using the Fujisaki-Okamoto (FO) transformation. HQC requires sparse fixed-weight vectors in both key generation and encryption, making FWVS a vital component of its algorithm.

2.3 Masking

Power side-channel analyis, like Differential Power Analysis (DPA) [KJJ99], is a passive hardware security threat where the attacker measures the power consumption of a secret computation and analyzes these measurements to eventually reveal the secret. Masking is a countermeasure against such kinds of attacks and is based on secret sharing, which can be applied in various domains. This means that sensitive intermediate values are

represented by several randomized shares and processed separately, so that leakage from a share does not reveal information about the secret. A d-th-order masking scheme splits secrets into $d+1$ shares and is intended to protect against d-th-order statistical attacks.

The most common masking domains are the Boolean and the arithmetic masking domains. In Boolean masking, a secret bit $x \in \mathbb{F}_2$ is shared into $d+1$ bits $x_i \in \mathbb{F}_2$ such that $x = x_0 \oplus \cdots \oplus x_d$. Boolean masking is efficient for Boolean linear operations: `NOT` and `XOR` can be applied share-wise. In contrast, logical `AND` and `OR` are non-linear and therefore require dedicated masking gadgets to securely combine and then re-share different intermediate recombinations. Arithmetic masking instead represents secrets in an integer residue ring, where addition and multiplication by public constants are linear. Since different parts of an algorithm may be more efficient in different masking domains, secure conversions between domains are often required, but costly to implement.

Among others, Probe-Isolating Non-Interference (PINI) gadgets, introduced by Cassiers *et al.* [CS20], are masking gadgets that allow for the design of masked circuits due to their composability: Any circuit built exclusively from PINI gadgets is itself PINI-secure, meaning it is a provably side-channel-secure design.

As Boolean masking requires many bitwise operations, bitslicing [GR17] is frequently used to optimize the runtime of masked software implementations. Instead of computing operations on one data element at a time, bitslicing stores the same bit position of multiple independent values in one word. Thus, bitwise instructions can process many bit-values in parallel, effectively transforming the processor into a Single-Instruction Multiple-Data (SIMD) device, provided the algorithm allows for word-level parallelism.

2.4 Bitonic Sorting and Merging Networks

Both Sota and MWSupport algorithms from *Sota Voce* require at one point side-channel-secure sorting or merging. However, many sorting algorithms are usually not constant-time or have secret-dependent memory access patterns. In contrast, sorting and merging networks provide a constant-time and data-oblivious alternative. As described by Knuth [Knu08], they consist of a pre-defined sequence of compare-and-swap operations that is independent of the input data. The key primitive is the compare-and-swap, where a pair of two single elements a_i, a_j is compared and conditionally swapped according to the sorting direction, such that $a'_i \leftarrow \min(a_i, a_j)$ and $a'_j \leftarrow \max(a_i, a_j)$.

The bitonic sorting network is a recursive network introduced by Batcher [Bat68] that can sort sequences of arbitrary length and has a complexity of $O(n \log^2 n)$. The network itself is built out of bitonic merging networks that transform a bitonic sequence into a sorted sequence. A sequence is bitonic if it consists of an increasing and a decreasing subsequence, or any cyclic variation of that. The sorting network recursively creates bitonic sequences by concatenating an ascending and descending sorted subsequence, which are then merged to create a fully sorted sequence.

A bitonic merging network that merges a bitonic sequence A of length $n = 2^p$ begins by comparing $a_i, a_{i+n/2}$ for $i \in [n/2]$. This results in $n/2$ parallel, independent compare-and-swap operations with a compare distance of $s = n/2$. At every recursion the compare distance is halved $s \leftarrow s/2$ such that elements a_j, a_{j+s} are compared with

$$j(i, s) = (i \bmod s) + 2s \cdot \lfloor i/s \rfloor \tag{1}$$

for $i \in [n/2]$. At the base case, elements a_{2i} and a_{2i+1} are compared which corresponds to a compare distance of $s = 1$. Figure 1 displays a bitonic sorting network for $n = 8$ elements that highlights the recursive structure of both the sorting and merging networks.

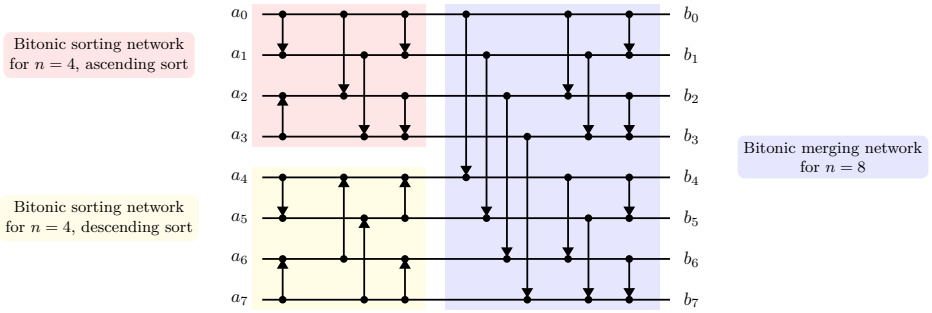


Figure 1: A bitonic sorting network for $n = 8$. Arrow orientation indicates compare direction, horizontal lines are data lines and vertical lines represent compare-and-swap operations.

3 Implementation of Masked Sota Support Sampling

In this section we present our masked implementation of the Sota support sampling algorithm as part of *Sota Voce* by Filho *et al.* [FAB+26a]. Our goal is to design a secure, any-order masked implementation using masking gadgets that minimizes the masking-induced performance overhead. To do so, we briefly explain the gadgets we use [BC22], before explaining the Sota algorithm and discussing which parts are sensitive and need to be side-channel protected. Finally, we show how we bitslice the algorithm, including the sorting network to speed up the implementation.

3.1 Gadgets and Masking Strategy

Our masked implementation is entirely built with the composable PINI gadgets by Bronchain *et al.* [BC22]. These gadgets have a defensive approach and are partially written in assembly to control instruction selection and register usage. This allows for dummy loads and stores that prevent accidental micro-architectural recombinations of shares, but comes with a significant performance overhead between 1.29× and 1.71× compared to the counterparts written in plain C. Yet, exhaustive TVLA analysis by Bronchain *et al.* [BC22] justifies the decision for the assembly gadgets, as no statistical leakage is detectable, whereas the C versions show leakage.

Typically, masked implementations alternate between masking domains to leverage the corresponding linear operations. In particular, arithmetic-to-Boolean conversion and the reverse are common primitives for masked implementations. However, these conversions are non-linear and thus costly with regard to performance and randomness usage [BC22]. Due to the frequent change of bitwise and arithmetic operations in *Sota Voce*, converting to the optimal masking domain for each operation would require many conversions. Instead, we mask our implementation entirely in the Boolean domain to avoid all domain conversions.

The gadgets by [BC22] provide any-order masked versions of `XOR` and `AND`. The latter is non-linear and requires additional randomness for masking. Hence we minimize the amount of non-linear `AND` gadgets to reduce the randomness usage and improve the runtime performance. For masked addition of k-bit masked numbers in the Boolean domain, we rely on the composed, bitsliced addition gadget by [BC22] that consists of $O(k)$ `AND` gadgets. This is also non-linear but faster than converting to the arithmetic domain, as the conversion itself uses bitsliced addition.

3.2 Sota Support Sampling Algorithm

To sample a support vector S with ω distinct entries in constant-time, the Sota algorithm [FAB+26a], shown in Algorithm 3.1, follows a minimal approach: First, ω random integers $X \in [n-\omega+1]^\omega$ are sorted using a sorting network. Then, the index sequence $(0, 1, \ldots, \omega-1)$ is added to the sorted numbers. The result is a strictly increasing support vector S of ω elements in $[n]$.

Due to strict monotonicity, it is guaranteed that all entries are distinct, resulting in a valid support vector on every run. Thereby, the algorithm refrains from rejection and is constant-time. However, Sota introduces a small measurable bias in the distribution of the sampled support vectors. Whether this bias interferes with the security depends on the concrete scheme; for instance the authors of [FAB+26a] proved that the bias is acceptable for the security of HQC.

Algorithm 3.1: Sota(n, ω, X) [FAB+26a]

Input: A sequence $X \in [n-\omega+1]^\omega$.
Output: A distinct, sorted support $S \in [n]^\omega$.

$(s'_0, \ldots, s'_{\omega-1}) \leftarrow \mathrm{sort}(X)$
for $i = 0$ **to** $\omega - 1$ **do**
  $s_i \leftarrow s'_i + i$
return $(s_0, \ldots, s_{\omega-1})$

3.3 Sensitive Operations

The sampled integers X as input and all derived values determine the final fixed-weight vector and are thereby sensitive. Thus, all operations working with these sensitive values need to be masked, which includes the sorting procedure and the addition of the per-entry indices. For a masked sorting network, it is sufficient to provide a masked compare-and-swap, as it is the only operation that directly interacts with sensitive data. This requires masked integer comparisons and constant-time conditional swaps on masked flags; both operations are discussed below. Krausz *et al.* [KLRG23] provide Boolean terms that minimize the number of non-linear operations for exactly these compare-and-swap operations. This is necessary since non-linear masking gadgets are costly, therefore, minimizing their amount directly yields a performance improvement.

Integer comparison. Comparison of integers is not possible in the arithmetic masking domain, as the underlying integer modulo ring has no partial order relation defined. In contrast, in the Boolean domain, it is possible to determine the order of two numbers. For two k-bit Boolean masked numbers a and b, Algorithm 3.2 calculates the signedness $r \leftarrow a < b$ by selecting the most significant bit t in which a and b differ. Then, r is 1 if t is set in b, else a must be larger.

Similarly, the masked equality check verifies that a and b are equal in all of the k bits. This is achieved by calculating their `XOR` sum and accumulating all resulting k bits into a single bit-flag r using Boolean `OR` gadgets. Then, r is 0 only if a and b are equal. Both algorithms rely only on bitwise, Boolean operations and require $O(k)$ non-linear gadgets.

Constant-time Conditional Set and Swap. To set the masked variable a to a new value z if the masked flag r is set, [KLRG23] utilize (2) that minimizes the amount of non-linear operations. A similar construction is used to swap two Boolean masked values a, b conditionally on the flag r with as few non-linear gadgets as possible.

$$a \leftarrow a \oplus ((a \oplus z) \wedge r) \tag{2}$$

Algorithm 3.2: Optimized Integer Comparison in Boolean Domain [KLRG23].

Input: k bit numbers a and b.
Output: 1-bit signedness $r \leftarrow a < b$.

```
1 r[0] ← 0, t ← a ⊕ b
2 for i in [k] do
3     r[0] ← r[0] ⊕ ((r[0] ⊕ b[i]) ∧ t[i])        // b[i] is the i-th bit of b
4 return r[0]
```

3.4 Bitslicing Sota

All sensitive operations in the masked algorithm work in Boolean values only, some even on a bitwise level. Bitslicing allows for a significant performance increase if the implementation is able to parallelize the application. In the following, we discuss how bitslicing for Sota is implemented and what optimization it enables.

Bitonic Sorting Network. For the sorting of the integers in Sota, we follow [FAB+26a] and use a sorting network. In particular, we adapted the masked bitonic sorting network implementation of [KLRG23], which operates with Boolean shares due to the masked compare-and-swap primitive. The underlying integer comparison requires sequential bitwise manipulations which are inefficient with values in canonical form. Bitslicing allows for a significant speedup by parallelizing 32 compare-and-swaps among 64 masked values (on 32-bit machines). Bitonic sorting networks are particularly suited for bitslicing, as at every recursion the amount of comparisons is constant across all lanes of the network and the compare distance between lanes is equal. Concretely, for a network of n elements, there are, at every level, $n/2$ independent comparisons with a compare distance of s.

Bitsliced Lane Alignment. Comparing bitsliced numbers that lie within the same bitsliced word is non-trivial: In bitsliced representation, a 32-bit memory word contains bits of 32 subsequent numbers, for instance $a_0[0], a_1[0], \ldots, a_{31}[0]$. For compare distances $s = 2^p, p \geq 5$, parallel comparison is trivial, as numbers are naturally aligned in bitsliced form. For example, in the case of $s = 32$, two consecutive memory words can be directly compared, since this would resolve to bitsliced comparisons of $(a_0, a_{32}), (a_1, a_{33}), \cdots, (a_{31}, a_{63})$. However, for smaller compare distances, that is $s \in \{16, 8, 4, 2, 1\}$, the compared numbers are not distributed among two bitsliced words. Comparison of numbers that lie within a single memory word is not efficiently possible. Instead, the numbers of two bitsliced memory words are rearranged by rotating the bits to align them according to the compare distance. Then, they are exchanged using constant swap-masks, such that for $s = 16$ the first word holds numbers $a_0, a_1, \ldots, a_{15}, a_{32}, \ldots, a_{47}$ and the second the numbers $a_{16}, a_{17}, \ldots, a_{31}, a_{48}, \ldots, a_{63}$. Subsequent rearrangements are necessary to align for even smaller compare distances s. For $s = 1$, the numbers are effectively divided into even-indexed and odd-indexed to allow compare-and-swap of a_{2i}, a_{2i+1}. Figure 2 gives an example for the realignment of two 4-bit bitsliced words for the bitonic sorting network. The realignment is a linear operation that allows to always process two 32 bitsliced numbers in parallel, fully utilizing the processor's bit width.

Bit Width Optimization. Bitslicing enables optimizing to the exact bit widths of the input. The number of expensive `AND` gadgets in the masked compare-and-swap operation grows linearly with the number of bits k of the comparands. We know that the support entries require at most $k = \lceil \log_2(n) \rceil$ bits, which is almost always less than a normal 32-bit word. Thus, limiting the bitsliced comparisons to k bits guarantees that no unnecessary comparisons are performed.

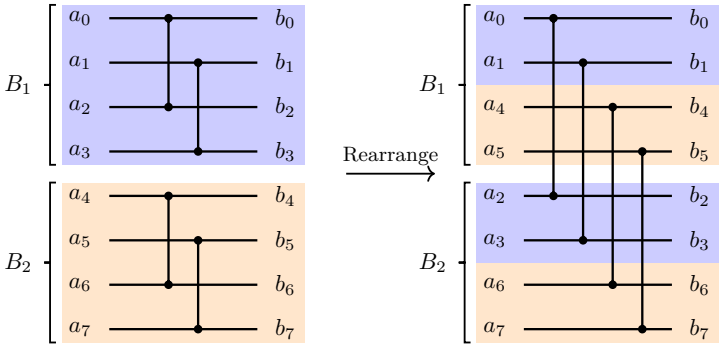


Figure 2: Example realignment of two bitsliced words B_1, B_2 with 4 numbers each and compare-distance $s = 2$ for a bitsliced comparison on a 4-bit machine.

Padding to 64. Two bitsliced words are required for bitsliced comparisons. Thus, the number of processed elements is a multiple of 64. However, the number of elements to sort for the Sota algorithm is determined by ω, which is not necessarily a multiple of 64. Therefore, we pad the input sequence for bitslicing to reach the desired input lengths. For the value of the padded elements, we choose the largest possible value for the bit width, $2^{\lceil \log_2(n) \rceil} - 1$. This guarantees that the padded elements do not interfere with the Sota algorithm, since these elements will naturally stay at the end of the sequence through the sorting process.

Addition of Index. For the addition of the public sequence $(0, 1, \ldots, \omega - 1)$ after the sorting, we use the bitsliced addition gadget. It expects two bitsliced, masked numbers as operands, so we generate the per-entry index directly bitsliced for $i \in [\omega]$ in chunks of 32. As this value is public, it is sufficient to set only the first share of the masked number and clear remaining shares to zero, saving random bytes for the masking. Conversion to arithmetic masking is inefficient, as the support is Boolean-masked due to the masked sorting network and the following support-to-vector conversion algorithm also expects Boolean-masked values. Hence, implementing the addition directly in the Boolean domain saves two costly conversions.

In summary, our masked Sota implementation samples a sorted vector support S that consists of ω distinct entries in constant time. The result is a bitsliced and padded, Boolean masked array of $64 \cdot \lceil \omega/64 \rceil$ elements, which can be consumed by the masked support-to-vector conversion algorithm of Section 4.

4 Masking of MWSupport Conversion Algorithm

To the best of our knowledge, masking the support-to-vector conversion (Step 3 of FWVS) has not been the subject of intensive research. Many masked implementations use a suboptimal, naive approach [KLRG23, SVP+24]. As our benchmarks in Section 5 show, it is the dominant operation of the masked FWVS. Many code-based schemes like HQC require the sampled vector to be written as a n-bit vector. Thus, after the sampling of the support S, a subsequent support-to-vector conversion is required.

In this section, we introduce a masked and bitsliced implementation of the novel MWSupport algorithm from *Sota Voce* by [FAB+26a]. We follow the same masking guidelines for this implementation as for the masked Sota implementation, specified in Section 3. We start by explaining what makes the naive approach unsuitable for masking.

Naive Algorithm. For the support-to-vector conversion, most masked implementations rely on a straightforward approach. As the total vector bit width n exceeds the word size B of the machine, the bit vector is written as $N = \lceil n/B \rceil$ consecutive memory words. To do so, the naive algorithm first transforms the support entries s_i into pairs consisting of word value v_i and the corresponding word index k_i for the word size B. Then, it iterates over all $j \in [N]$ memory indices and checks for all $i \in [\omega]$ pairs, whether $k_i = j$ and conditionally assigns v_i . This naive approach has a total of $O(n\omega) = O(n^{3/2})$ comparisons. While comparisons are cheap in an unmasked implementation, masked comparisons, on the other hand, are costly, as they require bitwise, non-linear Boolean operations. Consequently, this approach is unsuited for masking.

MWSupport. In *Sota Voce*, Filho *et al.* [FAB+26a] propose a more sophisticated conversion algorithm called MWSupport, which has a superior complexity of $O(n \log n)$ with less comparisons over the naive approach. As a result, MWSupport is set to become part of the official HQC reference implementation [FAB+26b]. We base our masked conversion on this algorithm, and thus benefit from its complexity loss and reduced number of comparisons. MWSupport leverages the sorted property of the support vector S sampled by Sota, as discussed in Section 3, to greatly increase the performance. However, it can also be combined with other support sampling procedures if an additional sort is performed beforehand. In that case, we can reuse the masked bitonic sorting network from Section 3.4.

MWSupport converts the final bit vector in batches of word size B, in four steps. It first converts all s_i into their corresponding index-value-pairs (k_i, v_i) in a process called bitgrouping. Second, to limit the amount of total comparisons, pairs that target the same output word, that is $k_i = k_j$, are combined into a single entry. As the exact amount of redundant entries is secret-dependent, a compaction algorithm is used afterwards for a side-channel secure removal. Finally, the B-bit word values v_i are moved to the correct position within the Fixed-Weight Vector (FWV) and interleaved with zero words to achieve the full n-bit fixed-weight vector with a weight of ω. Due to the sorted property of the word indices, which they inherit from the sorted support, the interleaving is done with an efficient merging network. The merging network dominates the computation, leading to an overall complexity of $O(n \log n)$.

We mask all steps, as the support and the bit vector are sensitive values. By parallelizing the bitgrouping, compaction, and the merging with bitslicing, we gain a significant performance increase for the masked implementation. In addition, we save masking conversion gadgets by only using Boolean masking.

4.1 Bitgrouping

Similar to the naive algorithm, the first step of MWSupport is to group the support according to a word size B. The authors of *Sota Voce* [FAB+26a] designed MWSupport with 64-bit systems in mind and selected a word size of $B = 48$ to store the whole (k_i, v_i) pair within the same memory word. This allows faster swapping of the pairs, as it only requires a single move instruction. However, this approach does not translate well for 32-bit systems, as for large n it is not possible do store both, word value v_i and word index k_i in 32 bits. This is, for instance, the case for all parameters of HQC. Additionally, for a masked implementation we face further challenges: First, for the computation of the word index k_i in line 2 of Algorithm 4.1, the support entry s_i is divided by B but division is a non-linear and complex operation in masking. Hence, we restrict B to powers of two, such that division degenerates to a share-wise right-shift in Boolean masking.

Second, the word value computation in line 3 of Algorithm 4.1 is a left shift by the secret variable s_i. Left-shifting a public value by a secret amount is a non-linear operation in the Boolean domain, as well as in the arithmetic domain. Yet, the modulo reduction of s_i by B limits the possible outcomes to B potential word values. In our masked implementation, we

Algorithm 4.1: Bitgrouping(S, B) [FAB+26a]

Input: Sorted support vector S of length ω and a base B.
Output: Word indices K, uncombined word values V of length ω for base B

```
1 for i ∈ [ω] do
2     k_i ← ⌊s_i / B⌋
3     v_i ← 2^(s_i mod B)
4 return K, V
```

use this property to emulate the left shift. We iterate over all possible values $x \in [B]$ and perform a masked comparison with $s_i \bmod B$, as shown in Algorithm 4.2. Depending on the outcome, the word value v_i is set conditionally to $1 << x$. To assert the constant-time property, it is necessary to iterate over all possible $x \in [B]$.

Note that the bitgrouping process is performed independently for all elements in the support S and consists only of Boolean bitwise operations and shift instructions. Thus, it is suitable for bitslicing which allows us to parallelize the process and further improve the performance. Additionally, bitslicing allows us to tailor the bit width of the word values v_i to the FWVS parameters. Moreover, the index-value pair (k_i, v_i) must no longer fit within a single 32-bit register so we set $B = 32$. For the v_i and k_i, B bits and $\lceil \log_2(N) \rceil$ bits are required, respectively. As the masked comparison's complexity grows linearly with the number of bits of the operands, a compact data-representation avoids unnecessary bit-comparisons.

Algorithm 4.2: MaskedBitgrouping(S, B)

Input: Boolean masked, sorted support vector S of length ω and a base B, B is a power of two.
Output: Boolean masked word indices K, uncombined word values V of length ω for base B.

```
1 for i ∈ [ω] do
2     k_i ← s_i >> log2(B) // Share-wise right shift
3     v_i ← 0
4     for x ∈ [B] do
5         b ← s_i mod B == x // Masked comparison
6         c ←mask(1 << x) // Mask constant
7         CSet(b, v_i, c)
```

4.2 Combining Duplicate Pairs

To ensure that each word index k_i occurs at most once for the final merge, the word values v_i, v_j that have the same word index $k_i = k_j$ must be combined. Since the word indices K inherit the sorted property of the support S, it is sufficient to check if $k_i = k_{i+1}$. For two equal neighboring entries k_i, k_{i+1}, the word values are combined in $v_{i+1} \leftarrow v_i + v_{i+1}$ and k_i is set to a large sentinel value ∞_B, as shown in Algorithm 4.3. In the following compaction step, the sentinel values are pushed to the end of the list. If we select $\infty_B = 2^{\lceil \log_2(N) \rceil} - 1$ to be larger than all k_i, the compaction will result in a sorted sequence. For masking, we can safely substitute the addition in line 4 in Algorithm 4.3 with a cheaper `XOR`, as it is guaranteed that v_i and v_{i+1} have distinct bits set if $k_i = k_{i+1}$.

The information that two neighboring pairs have the same index $k_i = k_{i+1}$ is later reused inverted for the compaction as the so-called marks array M. To save redundant

computations, we calculate M using a bitsliced comparison: We copy K and shift its numbers by one such that elements k_i and k_{i+1} align in the bitsliced representation. Then, a masked comparison results in the bitsliced marks M, where $M_i = [k_i \neq k_{i+1}]$.

The actual combination of the word values V is a sequential operation that is not efficiently parallelizable, thus we cannot apply bitslicing here. Instead, the word values V are first transformed into canonical form, combined sequentially depending on M and finally transformed back to bitsliced form, marking it the only component of the whole algorithm that is not bitsliced. This also limits the word size to $B = 32$, as this is the maximum register width on 32-bit systems.

Algorithm 4.3: Combining(K, V) [FAB+26a]

Input: Word indices K, uncombined word values V of length ω.
Output: Uncompacted word indices K, combined word values V of length ω and marks array M.

1 **for** $i \in [\omega - 1]$ **do**
2 $\quad M_i \leftarrow (k_i \neq k_{i+1})$
3 $\quad k' \leftarrow \infty$
4 $\quad v' \leftarrow v_i + v_{i+1}$
5 $\quad$ CSwap$(\neg M_i, v', v_{i+1})$
6 $\quad$ CSwap$(\neg M_i, k', k_i)$
7 **return** K, V, M

4.3 Compaction of Redundant Pairs

In this section, we mask and bitslice the recursive, oblivious compaction algorithm by Sasy *et al.* [SJG22]. Compaction is the process of pushing a marked subset of elements to the end of a list whilst retaining the order of the unmarked elements. The authors of *Sota Voce* [FAB+26a] use this algorithm to push the redundant pairs (k_i, v_i) marked with $k_i = \infty_B$ to the end of the array to restore the sorted property. As a result of the pair combining, the former sorted word indices K contain marked entries that impede the monotonicity. Still, the subset of non-redundant entries is sorted. Thus, the monotonicity of K can be restored with a compaction which is more efficient than a full sort.

The algorithm, as shown in Algorithm 4.4, compacts the pairs recursively in a divide-and-conquer fashion: First, the input array is halved (line 7-8) and the algorithm calls itself recursively on both slices (line 10-11). In each invocation, an offset z is calculated based on the mark count variable m. The mark count m contains the number of marked entries in the first slice (line 9). The base case $p = 1$ is reached when only two elements are left in the slice. The elements are swapped according to the offset z and the marks in M (line 2-4). After the recursion, the compacted slices are recombined according to a swap-flag that is derived from z, m and the swap distance s (line 12-16).

By bitslicing this algorithm, we can parallelize recursions among multiple slices, similar to the bitonic sorting network in Section 3.4. This allows us to significantly improve the performance of the masked algorithm. Yet, multiple challenges and inefficiencies arise while masking the algorithm, which we describe in the following.

Counting of Marked Elements. Given an array of length 2^p, we observe that the marks count m is recomputed in every recursion over slices of decreasing length: Let $m_{i,j} = \sum_{l=i}^{j} M_l$ denote the amount of marks over the interval $[i, j)$, as $M_i = [k_i \neq k_{i+1}]$. First, the algorithm computes $m_{0,2^{p-1}}$ and in the following recursion $m_{0,2^{p-2}}$. This is inefficient for masking, because counting is a non-linear operation in the Boolean domain

Algorithm 4.4: OffCompact(V, K, z, p) (from [FAB+26a] based on [SJG22]).

Input: Uncompacted word indices K, combined word values V, marks array M of length 2^p, with cyclic offset of z.
Output: Compacted K, V.

if $p = 1$ **then**
  $b \leftarrow (z = 1) \oplus (k_0 = \infty_B \text{ and } k_1 \neq \infty_B)$ CSwap(b, v_0, v_1)
  CSwap(b, k_0, k_1)
else if $p > 1$ **then**
  $s \leftarrow 2^{p-1}$
  $V^{(L)} \leftarrow (v_0, v_1, \ldots, v_{s-1}),\ V^{(R)} \leftarrow (v_s, v_{s+1}, \ldots, v_{2^p-1})$
  $K^{(L)} \leftarrow (k_0, k_1, \ldots, k_{s-1}),\ K^{(R)} \leftarrow (k_s, k_{s+1}, \ldots, k_{2^p-1})$
  $m \leftarrow |\{i \in [s] : k_i \neq \infty_B\}|$ `// Count entries` $k_i \neq \infty_B$`.`
  OffCompact$(V^{(L)}, K^{(L)}, z \bmod s, p-1)$
  OffCompact$(V^{(R)}, K^{(R)}, (z+m) \bmod s, p-1)$
  $b_1 \leftarrow [((z \bmod s) + m) \geq s] \oplus [z \geq s]$
  for $i \in [s]$ **do**
    $b_2 \leftarrow b_1 \oplus [i \geq ((z+m) \bmod s)]$
    CSwap(b_2, v_i, v_{i+s})
    CSwap(b_2, k_i, k_{i+s})

and we already computed $m_{0,2^{p-2}}$ as part of $m_{0,2^{p-1}}$. Moreover, we detect that the actual value of m is first used for the offset of the second recursion. In other words, the total count of marks of the first slice is used for the second slice in the recursion. In our masked implementation, the algorithm instead executes the first recursion on the first half-slice and then counts $m_{0,2^{p-1}}$ from the bottom-up to avoid unnecessary additions. By adding $m_{2^{p-2},2^{p-1}}$ to the already computed $m_{0,2^{p-2}}$ we receive $m_{0,2^{p-1}}$ with fewer additions.

Intermediate Values and Comparisons. In every non-leaf invocation of the algorithm with $p > 1$, $s = 2^{p-1}$, and offset z, three intermediate values are computed: $z \bmod s$, $(z + m) \bmod s$, and $(z \bmod s) + m$. These values are used as an offset for the next recursion and drive the recombination where they are compared to public constants, as recalled in the following equations:

$$(z \bmod s) + m \geq s \tag{3}$$

$$z \geq s \tag{4}$$

$$(z + m) \bmod s \leq i, \qquad \forall i \in [s] \tag{5}$$

To make the comparisons and modulo reductions more efficient, we leverage the bitsliced data representation: Since the modulus s is always a power of two, we emulate the reduction by reading only the first $p-1$ bits of the number, which is possible as the masked numbers are stored bitwise. Similarly, comparisons (3) and (4) are reduced by accumulating all bits $k > p$ with an `AND` and negating the result. For the comparison of $[s]$ elements in (5), we reuse the masked comparison from the sorting networks to parallelize the operation, as all comparisons are mutually independent.

Bitslicing Special Case of $2^p \leq 64$. Bitslicing allows the processing of 32 values in parallel. However, to leverage the full potential, each number must operate independently of the other numbers within a single bitsliced word. Analogously to the bitsliced bitonic

sorting network from Section 3.4, compaction with swap distance $s < 32$ constitutes an edge case in the bitsliced format, as it involves comparing elements that reside within the same bitsliced word. Therefore, we dedicate recursions with $2^p \leq 64$ to a special bitsliced procedure. At the top level, the swap distance $s = 32$ matches the bitslicing size of 32, so two adjacent bitsliced words naturally align for parallel compaction, without any realignment. At deeper levels, the swap distance is halved and falls below 32, so the bitsliced pairs (k_i, v_i) must be rearranged before each level such that elements at distance s align across two bitsliced words. For that, the realignment from Section 3.4 is reused.

Moreover, we transform the recursion into an iteration from bottom-up: As the first swap occurs in the leaf invocation of $p = 1$ for pairs (k_0, v_0) and (k_1, v_1), we also compute all other 31 leaf invocations for pairs (k_{2i}, v_{2i}) and (k_{2i+1}, v_{2i+1}) in parallel. Yet, this requires knowing the offsets z for all 32 compare-and-swaps beforehand. Usually, the offsets are calculated through the recursion and passed down, so for deeper recursions they are unknown. The same problem applies for all iterations with $p > 1$. We solve this by computing all required m and z values for recursion levels $p \in \{1, 2, \ldots, 6\}$ from the bottom-up and caching the intermediate results. We further optimize the bitsliced alignment of lanes to reduce the number of redundant computations. In the following, we describe the bitsliced computation of m and z and illustrate an example for a 4-bit machine.

Step 1: Count marks `m[p]`. As a byproduct of the duplicate-pair combining, we have the marks array M in bitsliced form with $M_i = [k_i = k_{i+1}]$. In the leaf invocation ($p = 1$) at each of the 32 lanes, we compare M_{2i} and M_{2i+1}. To correctly align M, we apply the realignment for two bitsliced blocks to receive `M0` and `M1`, where `M0` contains all even-indexed marks of M and `M1` the odd-indexed marks. By adding those in a pairwise manner, we compute auxiliary `H[p]` values. Through realignment and addition with themselves, we construct `m[p]` values for higher levels $p > 1$ that are already aligned for bitsliced compare-and-swap. Figure 3 demonstrates the bitsliced computation which compacts $2^3 = 8$ elements. Note that `m[1]` is not required and can be omitted.

Step 2: Derive offsets `z[p]`. Given an initial offset z_0, the algorithm passes z_0 mod s for the first half-slice and applies $(z_0 + m)$ mod s to the second half-slice as new offset. Thus, at every compare-and-swap, the computed offset consists of a combination of m values and the initial offset z_0, reduced to a decreasing number of bits. Concretely, given a bitsliced and aligned `z[p+1]` we can compute `z[p]` by adding `m[p]` only to the second half and decrease the bit width by 1. We call this half-empty variable `T[p]` and it is constructed alongside `m` and `H`, as shown in Figure 3. Given z_0 we can also revert the computation of `z[p]` to:

$$\texttt{z[p]} = z_0 + \sum_{i=1}^{p} \texttt{T[i]} \pmod{2^p}. \tag{6}$$

Figure 4 illustrates the computation and alignment of `z` and shows how parallel, bitsliced compare-and-swap of `OffCompact` is performed. Table 1 gives an overview over all bitsliced variables and their bit widths.

Step 3: Conditional swap. When all intermediate values are calculated and aligned for every level, they are recombined via a masked compare-and-swap. To do so, the bitsliced index-value pairs are aligned according to the swap distance s, similar to the bitonic sorting network. The comparison itself is done with the index i and the calculated $z + m$ mod s. We achieve this in bitslicing, by splatting increasing intervals of i among the lanes, from $8 \times (0, 1, 2, 3)$ at $p = 2$ to $1 \times (0, 1, \ldots, 31)$ at $p = 5$ for a 32-bit device. Finally, the aligned pairs are swapped accordingly, as seen in Algorithm 4.4.

Table 1: Intermediate bitsliced variables for special case $2^p \leq 64$. Each variable is represented as a bitsliced word with the given bit width.

Variable	Width	Role
`M0`, `M1`	1 bit	Even- and odd-indexed marks of M, aligned through bitsliced realignment.
`H[p]`	$p+1$ bits	Total mark count at level p, built bottom-up from `M0`, `M1`.
`m[p]`	p bits	First-half mark count at level p, derived from `H[p-1]`
`T[p]`	p bits	Level-p increment of the offset, derived from `H[p]`.
`z[p]`	p bits	Cyclic offset at level p, `z[p]` $= z_0 + \sum_{i=1}^{p}$ `T[i]` $\pmod{2^p}$.

4.4 Merging with Zero Words

The final step of the conversion algorithm is tasked with positioning the ω non-zero pairs (k_i, v_i) at their corresponding positions among the total N B-bit words of the final vector. Since for sparse FWVS, $N \gg \omega$, the majority of the memory words will be zero-words, but the exact number is secret-dependent, as there is an unknown number of unique word indices k_i. Thus, a zero word for every possible index $j \in [N]$ with the pair $(j, 0)$ is provided for the final fixed-weight vector. To interleave these with the non-zero pairs, a merging network is used. After the compaction, the pairs (K, V) are in ascending order with regard to the word index K. To guarantee that the word values v_i are moved to the correct position, the pairs are converted into triples and re-indexed, such that the non-zero word value takes precedence over the zero word:

$$(k_i - i, i, v_i) \leftarrow (k_i, v_i) \text{ for all } i \in [\omega]. \tag{7}$$

To ensure that the non-zero words have higher precedence, the second component of the zero words is set to the sentinel value $\infty_M > \omega$:

$$(j, \infty_M, 0) \text{ for all } j \in [N]. \tag{8}$$

In the compare-step of the merging routine, the first element of the triple takes precedence over the second, while the third is ignored as payload. This will result in a sorted list of triples, where the third components are the word values of the final vector in correct order.

For the masked bitsliced merge, we use the bitonic merging network from Section 3.4. The concatenation of non-zero and zero triples is merged, which yields $N + \omega$ elements in total. As $N + \omega$ is most likely not a power of two, we must use a bitonic merging network for arbitrary input lengths. These have the additional requirement that the input must be an increasing bitonic sequence, that is a concatenation of an increasing sequence followed by a decreasing sequence. To adhere to that, we concatenate the increasing non-zero triples with a decreasing list of zero triples to receive an increasing bitonic sequence which we then sort with the bitonic merging network.

Bitslicing the merging network allows for a final, small optimization: Since the amount of required non-linear gadgets for the masked integer comparison grows linearly with the number of bits of the comparands, minimizing the bit width leads to better performance. While it is necessary to move the whole triple for swapping two entries, it is sufficient to limit the comparison to the bits of the first and second elements, as the third component is the word value, which is only payload. Hence, we move $\lceil \log_2 N \rceil + \lceil \log_2 \omega \rceil + B$ bits on each exchange but save B bits of the v_i in the comparison of the masked compare-and-swap in the merge. This seems minimal; however, the merging is the dominant factor of the whole algorithm, as its complexity is driven by n and not $\omega = O(\sqrt{n})$ unlike the other procedures, so the optimizations add up.

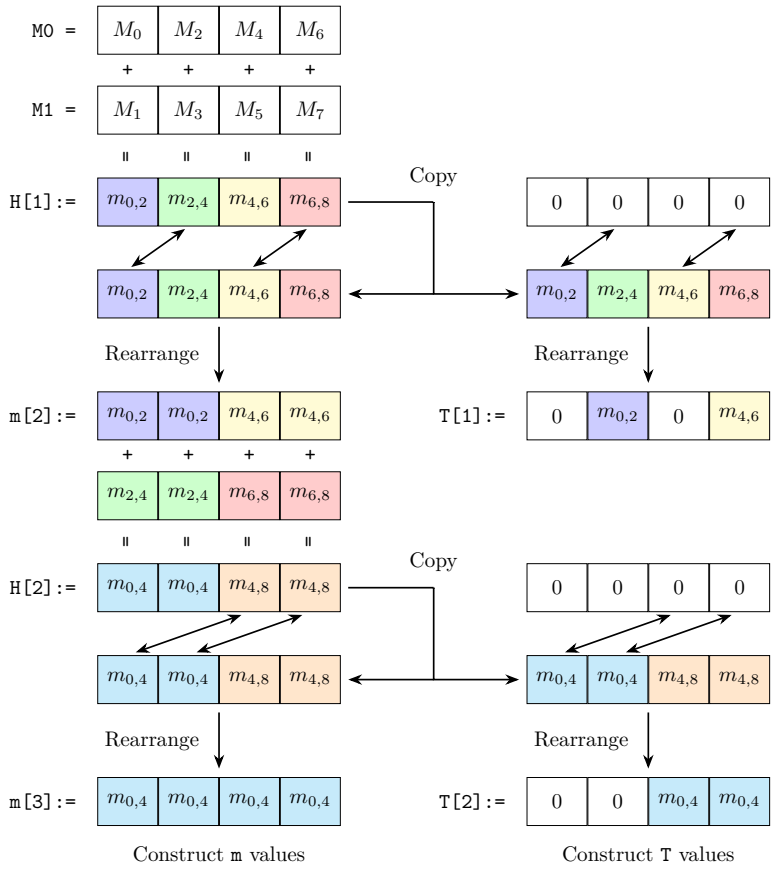


Figure 3: Example construction of intermediate values `T`, `H` and mark count `m` for 8 elements in a bitsliced setting with 4-bit registers.

To receive the masked vector of n bits and weight ω, we copy the first N merged word values. The algorithm guarantees that the word values are in correct order and yield the fixed-weight vector.

5 Evaluation

In the evaluation, we compare the runtime performance of our masked implementations of Sota and MWSupport against current state-of-the-art masked implementations for sparse FWVS. Furthermore, we provide an exhaustive TVLA analysis of our program to give statistical evidence for its masking effectiveness. We claim considerable runtime performance improvements in both support sampling and support-to-vector conversion while detecting no first-order leakage. For the support sampling, we compare our masked Sota against the Fisher-Yates-based sampler from [KLRG23], since it is their fastest masked implementation for constant-time sparse FWVS. Additionally, we compare against the masked HQC FWVS implementation from [SVP+24], which is also based on Fisher-Yates. For the support-to-vector conversion, we compare our masked MWSupport, with and without an additional sorting of the support, against the masked algorithms from [KLRG23] and [SVP+24]. They implement the naive approach of the support-to-vector conversion, as described in the beginning of Section 4.

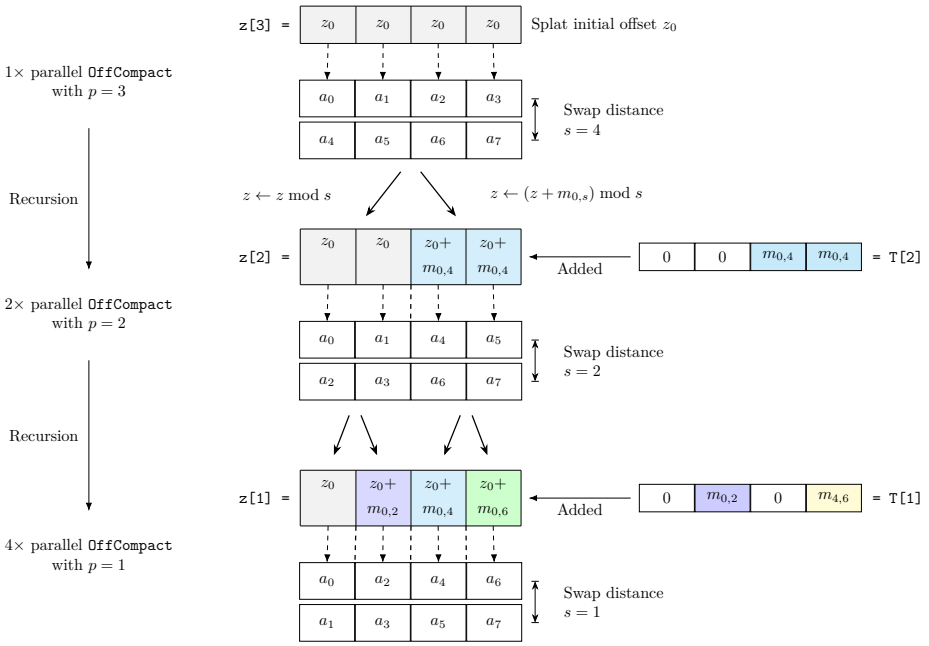


Figure 4: Parallel evaluation of `OffCompact` with alignment and construction of z values based on the initial offset z_0 for 8 elements in a bitsliced setting with 4-bit registers.

5.1 Setup

The algorithms are executed on a ChipWhisperer Lite hosting the STM32F303 target featuring a 32-bit ARM Cortex-M4, which runs at 7.384 MHz. The programs are compiled with first-order masking ($d = 1$), word size $B = 32$ and optimization level `-O3` using `arm-none-eabi-gcc` with version `10.3.1`. The cycles are measured by reading the `DWT_CYCCNT` register at beginning and end of the execution. The randomness required for the masking is generated beforehand and excluded from the cycle counts, although we record how many random bytes the implementation requires. By that, we decouple the performance analysis from the used PRNG and can give an estimate of the performance overhead introduced by an online PRNG. For the TVLA analysis, we use the Keysight 33500B waveform generator as an external clock with a frequency of 7.384 MHz and the PicoScope 6000E series oscilloscope to capture the traces with a sample rate of 78 MS/s.

Gadgets. As described in Subsection 3.1, we use the composable PINI gadgets by Bronchain *et al.* [BC22] for our masked implementations. The masked implementations from Krausz *et al.* [KLRG23] are based on the plain C variants of the same gadgets, so we exchanged them with the assembly gadgets for a fair comparison. The gadgets used by Spyropoulos *et al.* [SVP+24] reuse gadgets from the masked BIKE implementation by Demange *et al.* [DR24] which themselves are based on various previous works [BBE+18, CGTV15, Cor14]. However, none of the works contain practical leakage evaluations and [HKM+26] demonstrate that they contain exploitable leakage. As we could not trivially exchange the gadgets used in [SVP+24], we perform a gadget runtime evaluation to put the comparison into context. Table 2 shows the cycle counts measured on our test setup. The gadgets we deploy are between 2.72× and 2.85× slower than their counterparts in the implementation of [SVP+24] and require the same number of random bytes. Still, our masked support-to-vector conversion implementation achieves a better overall runtime.

Table 2: Cycle counts and random-byte consumption of selected gadgets on our setup.

Gadget	Cycles	Random Bytes	Relative speedup
`XOR` (Assembly) [BC22]	111	0	0.35×
`XOR` (C) [SVP+24]	39	0	2.85×
`AND` (Assembly) [BC22]	250	4	0.37×
`AND` (C) [SVP+24]	92	4	2.72×

5.2 Runtime Analysis

We select four sparse FWVS parameter sets from HQC and Classic McEliece for the benchmarking that are relevant for embedded devices. They cover wide ranges for n and ω to better classify the performance of the algorithms in varying application scenarios. Since the implementation by [SVP+24] is specifically designed for an application in HQC, we exclude it from benchmarking of classic McEliece parameters.

Support Sampling. For a fair comparison of the support sampling procedure, we exclude the masked integer reduction of [SVP+24] and provide all samplers with a sampled sequence of ω integers. Table 3 shows the cycle counts and randomness usage for all tested parameter sets and implementations. Our masked Sota is between 3.36× and 6.35× faster than [KLRG23] while requiring 3.21× to 5.56× fewer random bytes for masking. Compared to the masked FWVS for HQC by [SVP+24], masked Sota is 2.39× (HQC-128) and 5.23× (HQC-192) faster although the underlying gadgets we use are slower. Our implementation needs fewer random bytes for masking, further increasing the effective speedup.

The correlation between cycle count and randomness usage is explained by the application of non-linear `AND` gadgets as these are a major consumer of fresh randomness and have a heavy impact on cycle counts when present in the hot path of the code. The padding introduced by bitslicing is clearly observable in the difference between $\omega = 64$ and $\omega = 96$ for the McEliece parameters, as the implementation pads to the next multiple of 64 for the masked compare-and-swap. That also explains the marginal difference between $\omega = 75$ and $\omega = 114$ for the HQC parameters, as both are padded to 128 internally. The $57,000$ cycle difference is solely due to the increased bit width of the support entries for $n = 35\,851$, resulting in a few more cycles per each compare-and-swap. Bitslicing is also the reason why HQC-128 takes $120,000$ cycles longer than McEliece-460896, although the requested Hamming weight is smaller, due to padding and bit-width tailoring.

Table 3: Support sampling performance with respect to cycles and randomness consumption.

Scheme	n	ω	Cycles (in thousands)			Randomness (in bytes)		
			Sota	[KLRG23]	[SVP+24]	Sota	[KLRG23]	[SVP+24]
HQC-128	17 669	75	947	3 186	2 263	7 168	23 004	89 100
HQC-192	35 851	114	1 004	5 478	5 250	7 632	42 448	206 568
McEl.348864	3 488	64	300	1 904	–	2 188	10 920	–
McEl.460896	4 608	96	827	3 418	–	6 240	22 500	–

Support-to-Vector Conversion. For the support-to-vector conversion, both [KLRG23] and [SVP+24] rely on the same naive approach of iterating over the entire vector for each support entry to set the correct vector word. This approach has a complexity of $O(n\omega) = O(n^{3/2})$, whereas the new algorithm MWSupport runs in $O(n \log n)$. This

advantage in complexity is clearly shown in the cycle counts of Table 4. The speedup of our implementation ranges from 20.6× to 92.9× compared with [KLRG23]. In comparison with [SVP+24], the runtime is 35.3× and 49.7× faster, respectively. Considering the fact that [SVP+24] use faster gadgets, the effective speedup of our algorithm is even greater. The random byte requirements behave accordingly as shown in Table 5.

As MWSupport requires a sorted support vector, it works best in combination with Sota. However, it is still a competitive candidate for support-to-vector conversion when used with other support samplers which do not produce a sorted support. In that case, an additional masked sorting network is applied upstream to first sort the support. To do so, we reuse the bitonic sorting network from Section 3.4. Yet, the relative performance degradation compared with the other competitors is only marginal. MWSupport + Sort, even as a standalone algorithm, still reaches a speedup ranging from 18.1× up to 84.5×. This renders the masked implementation valuable even in combination with other support samplers.

We attribute this significant performance improvement over previous work due to the algorithmic superiority, as with increasing n the speedup grows. Moreover, the naive approach is particularly unsuited for masking, as it requires many redundant masked comparisons, which invoke costly `AND` gadgets. Parallelizing the majority of the algorithm with bitslicing further improves the performance.

Table 4: Support-to-vector conversion cycle counts (in thousands).

Scheme	n	ω	MWSupport	MWSupport + Sort	[KLRG23]	[SVP+24]
HQC-128	17 669	75	5 297	6 192	281 923	187 007
HQC-192	35 851	114	9 681	10 631	898 601	480 743
McEl.348864	3 488	64	2 024	2 301	41 705	–
McEl.460896	4 608	96	3 203	3 983	86 676	–

Table 5: Support-to-vector conversion masking randomness usage (in bytes).

Scheme	n	ω	MWSupport	MWSupport + Sort	[KLRG23]	[SVP+24]
HQC-128	17 669	75	59 584	66 528	2 654 400	9 870 116
HQC-192	35 851	114	104 104	111 496	8 689 992	25 234 512
McEl.348864	3 488	64	26 264	28 364	362 752	–
McEl.460896	4 608	96	41 016	47 064	774 144	–

5.3 TVLA Analysis

We perform exhaustive first-order TVLA analysis of our masked implementations of Sota and MWSupport. As for the FWVS parameters, we select HQC-128 with $n = 17\,669$ and $\omega = 75$. For each algorithm, we measure 100,000 traces and perform fixed-vs-random TVLA. Due to the high sample rate of our oscilloscope and the long operations, the length of the traces exceeds 10^7 samples. Therefore, using the standard threshold of $t_h = 4.5$ to declare leakage could lead to false positives, as described by Ding *et al.* [DZD+17]. Thus, we choose $t_h = 6$ as the threshold for our long traces, following their recommendations.

Results. Figure 5 shows the t-values of the TVLA of our masked Sota algorithm (Section 3). The traces have a length of approximately 10 M samples. There is no first-order statistical leakage, as no t-value exceeds the threshold of $t_h = 6$. The TVLA of our masked

support-to-vector conversion algorithm MWSupport (Section 4) is shown in Figure 6. The traces have a length exceeding 56 M samples. Again, no statistical first-order leakage is detectable given the threshold.

The strong theoretical and statistical security guarantees provided by the gadgets from [BC22] translate well into our masked implementations as shown in the TVLAs. Moreover, bitslicing is inherently less noisy than a representation in canonical form, as the logical bits of a secret value are distributed among multiple memory words and masking shares. Even with the highest optimization flag `-O3`, our implementations show no leakage.

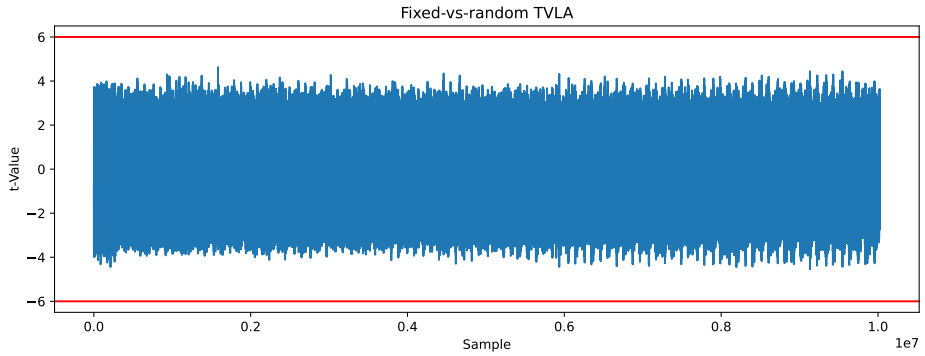


Figure 5: Fixed-vs-random TVLA with 100,000 traces of masked Sota algorithm running HQC-128 parameters.

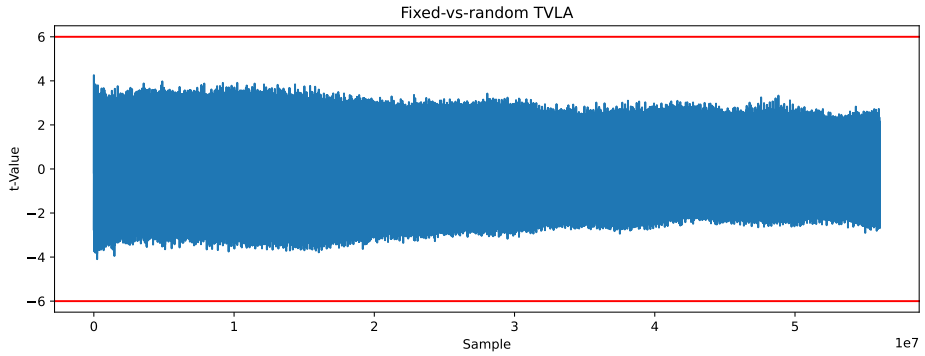


Figure 6: Fixed-vs-random TVLA with 100,000 traces of masked MWSupport algorithm running HQC-128 parameters.

6 Conclusion

In this work, we presented the first masked implementations of the *Sota* and *MWSupport* algorithms from *Sota Voce* [FAB+26a]. By bitslicing many components of the algorithms, including the compaction algorithm by Sasy *et al.* [SJG22], we achieve significant performance improvements over current state-of-the-art masked FWVS implementations. Moreover, we completely avoid costly masking domain conversions and limit the amount of non-linear masking gadgets where possible. Benchmarking demonstrates that our masked support-to-vector conversion algorithm MWSupport, in particular, shows a significant speedup and making it a viable for masked FWVS. Using only Boolean masking allows us to build our implementations on the strong PINI any-order masking gadgets from [BC22].

Their masking effectiveness translates well into our implementation, as we show with an exhaustive TVLA analysis.

Future work should include efficient masking of Step 1 of the sparse FWVS process. Spyropoulos *et al.* [SVP+24] mask the Barrett reduction used in HQC for this step with arithmetic masking, but they still exhibit detectable leakage that can be improved upon [HKM+26]. Since MWSupport will be integrated into the HQC reference implementation [FAB+26b], a fully masked HQC with secure and efficient FWVS is within reach.

References

[ABB+22] Nicolas Aragon, Paulo SLM Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghos, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichet, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE: Bit Flipping Key Encapsulation, 2022. https://bikesuite.org/files/v5.0/BIKE_Spec.2022.10.10.1.pdf.

[ABC+17] Martin 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. https://classic.mceliece.org/, 2017.

[Bat68] Kenneth E. Batcher. Sorting networks and their applications. In *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1968 Spring Joint Computer Conference, Atlantic City, NJ, USA, 30 April - 2 May 1968*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.

[BBE+18] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Benjamin Grégoire, Mélissa Rossi, and Mehdi Tibouchi. Masking the GLP lattice-based signature scheme at any order. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part II*, volume 10821 of *LNCS*, pages 354–384. Springer, Cham, April / May 2018.

[BC22] Olivier Bronchain and Gaëtan Cassiers. Bitslicing arithmetic/boolean masking conversions for fun and profit with application to lattice-based kems. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(4):553–588, 2022.

[Ber18] Daniel J Bernstein. Divergence bounds for random fixed-weight vectors obtained by sorting. https://ntruprime.cr.yp.to/divergence-20180430.pdf, 2018.

[CCSS24] Jung Hee Cheon, Hyeongmin Choe, Jungjoo Seo, and Hyoeun Seong. SMAUG (-t), revisited: Timing-secure, more compact, less failure. *IEEE Access*, 12:188386–188397, 2024.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to Boolean masking with logarithmic complexity. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 130–149. Springer, Berlin, Heidelberg, March 2015.

[CGTZ23] Jean-Sébastien Coron, François Gérard, Matthias Trannoy, and Rina Zeitoun. High-order masking of NTRU. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2023(2):180–211, 2023.

[Cor14] Jean-Sébastien Coron. Higher order masking of look-up tables. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 441–458. Springer, Berlin, Heidelberg, May 2014.

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020.

[DG19] Nir Drucker and Shay Gueron. Generating a random string with a fixed weight. In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning - Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27-28, 2019, Proceedings*, volume 11527 of *Lecture Notes in Computer Science*, pages 141–155. Springer, 2019.

[DR24] Loïc Demange and Mélissa Rossi. A provably masked implementation of BIKE key encapsulation mechanism. *CiC*, 1(1):23, 2024.

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017.

[EGRZ26] Elie Eid, Aurélien Greuet, Nathan Reboud, and Rina Zeitoun. Efficient high-order masking of frodokem's cdt-based gaussian sampler. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):868–892, Apr. 2026.

[FAB+26a] Décio Luiz Gazzoni Filho, Gora Adj, Slim Bettaieb, Alessandro Budroni, Jorge Chávez-Saab, and Francisco Rodríguez-Henríquez. Sota voce: Low-noise sampling of sparse fixed-weight vectors. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2026(1):353–375, 2026.

[FAB+26b] Décio Luiz Gazzoni Filho, Gora Adj, Slim Bettaieb, Alessandro Budroni, Jorge Chávez-Saab, and Francisco Rodríguez-Henríquez. Sota voce: Low-noise sampling of sparse fixed-weight vectors. Presentation at CHES 2026, October 2026. Available at: https://iacr.org/submit/files/slides/2026/tches/tches2026/1_53/1_53_slides.pdf.

[fSidI24] Bundesamt für Sicherheit in der Informationstechnik. Certification report bsi-dsz-cc-1249-2024, 2024.

[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.

[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.

[GMA+25] Philippe Gaborit, Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, Arnaud Dion, Jérôme Lacan, Jean-Marc Robert, Pascal Véron, Paulo L. Barreto, Santosh Ghosh, Shay Gueron, Tim Güneysu, Rafael Misoczki, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, and Valentin Vasseur. Hamming Quasi-Cyclic (HQC), 2025. https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf.

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 567–597. Springer, Cham, April / May 2017.

[HEHT26] Calvin Abou Haida, Thomas Espitau, Clément Hoffmann, and Mehdi Tibouchi. Slicing bits and cutting costs in cdt sampling: High-order masking of frodokem's gaussian sampler, revisited. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(4):549–574, Sep. 2026.

[HFG+25] Dina Hesse, Jakob Feldtkeller, Tim Güneysu, Julius Hermelink, Georg Land, Markus Krausz, and Jan Richter-Brockmann. t-probing (in-)security - pitfalls on noise assumptions. *IACR Cryptol. ePrint Arch.*, 2025:1202, 2025.

[HKM+26] Dina Hesse, Markus Krausz, Raagavan Muruganantha n, Tabea Wollinger, and Tim Güneysu. Power reveals timing conceals - side-channel attacks and hiding countermeasures for HQC's fixed-weight vector sampling. Cryptology ePrint Archive, Paper 2026/1462, 2026.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 463–481. Springer, Berlin, Heidelberg, August 2003.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Berlin, Heidelberg, August 1999.

[KLRG23] Markus Krausz, Georg Land, Jan Richter-Brockmann, and Tim Güneysu. A holistic approach towards side-channel secure fixed-weight polynomial sampling. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *Public-Key Cryptography - PKC 2023 - 26th IACR International Conference on Practice and Theory of Public-Key Cryptography, Atlanta, GA, USA, May 7-10, 2023, Proceedings, Part II*, volume 13941 of *Lecture Notes in Computer Science*, pages 94–124. Springer, 2023.

[KLS+25] Markus Krausz, Georg Land, Florian Stolz, Jan Richter-Brockmann, and Tim Güneysu. To extend or not to extend: Agile masking instructions for PQC. In Yongdae Kim, Atsuko Miyaji, and Mehdi Tibouchi, editors, *Cryptology and Network Security - 24th International Conference, CANS 2025, Osaka, Japan, November 17-20, 2025, Proceedings*, volume 16351 of *Lecture Notes in Computer Science*, pages 70–92. Springer, 2025.

[Knu08] Donald Ervin Knuth. *The art of computer programming 3 Sorting and searching*. Addison-Wesley, 2. ed., 24. print. edition, 2008.

[oST24a] National Institute of Standards and Technology. Module-lattice-based digital signature standard (FIPS 204). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf, 2024.

[oST24b] National Institute of Standards and Technology. Module-lattice-based key-encapsulation mechanism standard (FIPS 203). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf, 2024.

[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 Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 2565–2579. ACM, 2022.

[SVP+24] Maxime Spyropoulos, David Vigilant, Fabrice Perion, Renaud Pacalet, and Laurent Sauvage. Masked vector sampling for HQC. Cryptology ePrint Archive, Report 2024/1106, 2024.