Sparse Secrets, Dense Problems

Efficiently Masking HQC in Hardware

Fabian Buschkowski[1], Niklas Höher[1] and Tim Güneysu[1,2]

[1] Ruhr University Bochum, Bochum, Germany, firstname.lastname@rub.de
[2] DFKI GmbH, Bremen, Germany

Abstract. HQC has recently been selected by NIST as a code-based alternative to ML-KEM and is currently undergoing standardization. While multiple timing and power side-channel attacks against HQC have been presented, application of masking has so far only been studied for software implementations.
As part of this work, we close this gap by first proposing a novel masking technique for the sparse-dense vector multiplication, one of the core components in all HQC operations. We manage to securely exploit the sparsity of the secret operand using a mixed Boolean and arithmetic masking technique and prove its first-order security in the transition- and glitch-robust probing model as well as its PINI-composability. With the same memory width, our configurable and highly parallelizable masked design takes only 3 % more clock cycles than a previous unprotected HQC multiplier. Furthermore, our technique requires only 81 bits of randomness per cycle, irrespective of its parallelism. However, like previous Rotate-and-Add constructions, it is vulnerable to profiled horizontal side-channel attacks, which recover all support indices within ±8 positions from 200 traces of our FPGA implementation despite TVLA showing no signs of leakage after 10 million traces. To prevent such attacks, we integrate efficient hiding countermeasures natively into the architecture of the multiplier, resulting in only 1.7 % additional latency and moderate area overhead.
Together with efficiently masked fixed-weight samplers and a masked decoder, we present the, to our knowledge, first fully masked hardware implementation of HQC, supporting first-order masked key generation, encapsulation, and decapsulation. Notably, our masked key generation is faster than a previous unprotected design from literature on an Artix-7 FPGA. Lastly, all modules of our complete designs show no signs of first-order leakage after 1 million traces.

Keywords: HQC · Hardware Implementations · Masking · Sparse Polynomial Multiplication · Horizontal Attacks · Hiding

1 Introduction

The Hamming Quasi-Cyclic (HQC) Key Encapsulation Mechanism (KEM), has recently been chosen by the National Institute of Standards and Technology (NIST) for standardization as a code-based alternative to the lattice-based ML-KEM. While timing attacks on the HQC decoder [PT19, WBB+22] and the fixed-weight sampling [GHJ+22] have been mitigated through constant time implementations [WBB+22, Sen21, GMA+25], a number of recently published power [SRSW20, RBG+25, HWZY26, HKM+26] and electromagnetic [GLG22, PRJ+25, BSZ26] side-channel attacks require more sophisticated countermeasures. Masking is one of the most well-studied of these countermeasures, yet its application to HQC has so far been limited to software implementations [SVP+25, GSA+25]. Existing hardware implementations are either unprotected [DXN+23, ABP25, THKX23] or rely solely on hiding-based techniques [JLK+25a] without provable security.

Masking HQC in hardware poses several challenges since the fixed-weight vector sampling routine, the sparse-dense polynomial multiplication without the possibility of using the Number-Theoretic Transform (NTT) to speed it up, as well as the complex encoder and decoder are non-trivial to efficiently protect against side-channel adversaries.

Our Contribution. In this work, we solve the previously mentioned challenges and present, to the best of our knowledge, the first fully masked hardware implementation of HQC. In detail, our contributions are as follows:

- **Masked sparse-dense multiplication** (Section 3): We present a novel multiplication architecture that represents the sparse fixed-weight vectors as arithmetically masked indices and implements secure multiplication using a sequence of rotations by individual arithmetic shares. As a result, our architecture is highly parallelizable and achieves latencies only 3 % higher than previous unprotected designs, while only requiring a constant 81 bits of randomness per cycle. We prove the security of our proposed first-order masked construction in the glitch- and transition-robust probing model as well as its Probe-Isolating Non-Interference (PINI) composability.
- **Horizontal attack and hiding countermeasure** (Section 4): We demonstrate a profiled chosen-ciphertext side-channel attack on our FPGA implementation of the multiplication, recovering all 66 support indices of HQC-1 within ± 8 positions from 200 traces, although Test Vector Leakage Assessment (TVLA) using 10 million traces indicated no leakage. We prevent this attack by randomizing the rotation order, requiring 1.7 % additional latency.
- **Masked HQC design** (Section 5): We propose an efficient architecture for masked fixed-weight vector samplers and implement a fully masked Reed-Muller and Reed-Solomon decoder, which we integrate into complete hardware implementations of the key generation, encapsulation, and decapsulation routine within the existing HADES Design Space Exploration (DSE) and masking framework [BLH+25]. The resulting designs cover a wide range of performance trade-offs, with our first-order masked key generation beating a previous unmasked hardware implementation in terms of latency as well as delay. We additionally provide TVLA results of all three operations using 1 million traces each, showing no signs of leakage.
- **Open source**: The resulting hardware designs are available at `[URL REDACTED]`.

A Note on the new HQC Specification. Shortly before the submission of this work, the specification of HQC was updated, modifying some parameters [GMA+26]. While we were not able to adapt our work in the short amount of time, all techniques presented in this work remain applicable. The modified parameters do not influence our security evaluation and change the reported performance by a few percent at most.

2 Background

In this section, we provide background on masking, composability notions, and HQC.

2.1 Masking

Masking [CJRR99] has emerged as the most prominent countermeasure against side-channel attacks due to its provable security. The core idea of masking is to split all secret values into multiple statistically independent shares such that any subset of these shares yields no information about the original secret, which can be reconstructed by combining all shares.

Algorithm 1: HQC decapsulation

Input: $\mathsf{dk_{KEM}} = ((\mathsf{seed_{PKE.ek}}, \mathbf{s}), \mathsf{seed_{PKE.dk}}, \sigma, \mathsf{seed_{KEM}}), \mathsf{c_{KEM}} = ((\mathbf{u}, \mathbf{v}), \mathsf{salt})$

$\mathbf{y} \leftarrow \mathsf{SampleFixedWeightVect\$}(\mathsf{seed_{PKE.dk}})$

$\mathbf{m}' \leftarrow \mathsf{Decode}(\mathbf{v} - \mathsf{Truncate}(\mathbf{u} \cdot \mathbf{y}, l))$ /* end of decryption */

$(K', \theta') \leftarrow \mathsf{G}(\mathsf{H}(\mathsf{ek_{KEM}})||\mathbf{m}'||\mathsf{salt})$

$\mathbf{h} \leftarrow \mathsf{SampleVect}(\mathsf{seed_{PKE.ek}})$ /* start of re-encryption */

$(\mathbf{r'_2}, \mathbf{e}', \mathbf{r'_1}) \leftarrow \mathsf{SampleFixedWeightVect}(\theta')$

$\mathbf{u}' \leftarrow \mathbf{r'_1} + \mathbf{h}' \cdot \mathbf{r'_2}$

$\mathbf{v}' \leftarrow \mathsf{Encode}(\mathbf{m}') + \mathsf{Truncate}(\mathbf{s}' \cdot \mathbf{r'_2} + \mathbf{e}', l)$

$\mathsf{c'_{KEM}} \leftarrow (\mathbf{u}', \mathbf{v}', \mathsf{salt})$ /* end of re-encryption */

$\bar{K} \leftarrow \mathsf{J}(\mathsf{H}(\mathsf{ek_{KEM}})||\sigma||\theta')$

if $\mathsf{c_{KEM}} \neq \mathsf{c'_{KEM}}$ **then** $K' \leftarrow \bar{K}$

return K'

To securely share a secret x against a d-th order attacker, it is split into $d+1$ shares $x_0, ..., x_d$ such that $x_0 \circ x_1 \circ ... \circ x_d = x$ and any d shares are independent of x. For Boolean masking, which is very efficient for securing Boolean circuits such as cryptographic S-Boxes and symmetric ciphers, $\circ$ is instantiated with an XOR operation. For other operations, such as modular additions and multiplications over a field $\mathbb{Z}_q$, an arithmetic sharing allows for more efficient implementations. In this case, $\circ$ is instantiated as an addition in $\mathbb{Z}_q$.

Many Post-Quantum Cryptography (PQC) schemes, including HQC, combine Boolean and arithmetic operations, thus requiring masking conversions that allow converting shares from one masking domain to the other. After the first proposed first- and arbitrary-order conversions by Goubin [Gou01] and Coron et *al.* [CGV14], multiple works have improved these constructions by reducing their complexity and generalizing the approach to arbitrary moduli q [CGTV15, BC22].

Due to the complexity of securely masking entire circuits and the high susceptibility to errors, the concept of *gadget-based* masking has been introduced, which uses a few atomic and secure building blocks, so-called *gadgets*, to construct larger circuits. Important examples for Boolean gadgets include the HPC1 and HPC2 gadgets introduced by Cassiers et *al.* [CGLS21] and the low-latency HPC3 gadget by Knichel et *al.* [KM22].

2.2 Composability Notions

Since the composition of multiple secure gadgets does not necessarily result in a secure circuit, *composability notions* define a set of requirements for the construction of gadgets such that their composition remains secure. The first such notions were Non-Interference (NI) and Strong Non-Interference (SNI), proposed by Barthe et *al.* [BBD+15, BBD+16], for which linear operations require a refresh with randomness followed by a register stage. This issue was addressed by Cassiers et *al.* [CS20] with the notion of PINI, which allows the trivial composition of gadgets and, crucially, the sharewise application of linear functions without the need for additional refresh gadgets. The aforementioned Hardware Private Circuit (HPC) gadgets as well as the conversions from [BC22] are PINI.

2.3 HQC

HQC is a code-based KEM whose security is based on the hardness of the Quasi-Cyclic Syndrome Decoding (QCSD) problem, using the Fujisaki-Okamoto (FO) transform to achieve a CCA-secure KEM. All additions and multiplications are executed as carryless operations in the ring $\mathcal{R} = \mathbb{F}_2[X]/(X^n - 1)$. Algorithm 1 shows the decapsulation of HQC, which is the most complex of the three algorithms, while key generation and encapsulation

only require a subset of its operations. Vectors in $\mathcal{R}$ are represented by bold letters. For brevity, we have inlined all algorithms from the Public-Key Encryption (PKE) scheme and shortened notation. We refer to the specification [GMA+25, GMA+26] for details.

HQC uses the SHAKE_256 Extendable Output Function (XOF) together with three different sampling algorithms to sample random dense or sparse vectors. SampleVect generates a dense vector from $\mathbb{F}_2^n$ as the outputs of the XOF, while SampleFixedWeightVect$ and SampleFixedWeightVect generate sparse vectors by randomly sampling the indices of the ω or ω_R bits set to one. SampleFixedWeightVect$ reduces 24 bits modulo n and rejects the sample if the unreduced value was beyond a certain threshold. Valid samples are compared with all other samples and discarded if a collision occurs. SampleFixedWeightVect avoids rejection sampling by accepting a small bias in the resulting distribution. A colliding sample is replaced with the index of the other sample.

The decapsulation first calculates the sparse vector $\mathbf{y}$ using SampleFixedWeightVect$, multiplies it with $\mathbf{u}$ and then adds $\mathbf{v}$. The result is decoded into the message $\mathbf{m}'$ by first decoding each duplicated Reed-Muller codeword individually before decoding all resulting bytes with Reed-Solomon decoding, which marks the end of the decryption phase. Using $\mathbf{m}'$, the shared key K' and θ' are calculated with a hash function, and the ciphertext is re-encrypted. Afterwards, the original and the re-encrypted ciphertext are compared, and either K' or the rejection key $\bar{K}$ is returned.

3 Masked Sparse-Dense Multiplication

In this section, we introduce our novel masking approach for the sparse-dense vector multiplication. First, we discuss its usage in HQC before describing our approach, its hardware architecture, and its security, and finally sketch higher-order extensions.

3.1 Sparse-Dense Multiplication in HQC

During all three major HQC operations, one or more operations follow the form $\mathbf{v}_{\mathsf{dense}} \cdot \mathbf{v}_{\mathsf{sparse},1} + \mathbf{v}_{\mathsf{sparse},2} \mod (x^n - 1)$ where $\mathbf{v}_{\mathsf{dense}} \in \mathbb{F}_2^n$ refers to a dense public vector while $\mathbf{v}_{\mathsf{sparse},1}, \mathbf{v}_{\mathsf{sparse},2} \in \mathbb{F}_2^n$ refer to sparse secret vectors. Key generation computes $\mathbf{s} = \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$, encapsulation computes $\mathbf{u} = \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$ as well as $\mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$, and decapsulation computes $\mathbf{u} \cdot \mathbf{y}$, followed by both products of the encapsulation. We note that $\mathbf{u}$ comes directly from the ciphertext and can therefore be chosen by an adversary (cf. Section 4).

Internally, a sparse vector is a sequence of its ω support entries, each as a word of $\lceil \log_2 n \rceil$ bits. We denote the support entries of $\mathbf{v}_{\mathsf{sparse},1}$ as $s_0, ..., s_{\omega-1}$ and those of $\mathbf{v}_{\mathsf{sparse},2}$ as $s'_0, ..., s'_{\omega-1}$. In a masked implementation, these entries are generated by the masked fixed-weight vector samplers (see Section 5.2) in a Boolean-shared form, i.e., $s_i = s_i^{(0)} \oplus s_i^{(1)}$.

Most unmasked HQC and BIKE hardware designs [ABP25, JLK+25b, DXN+23, THKX23, RCGG22, HWCW19] use a Rotate-and-Add routine, i.e.,

$$\mathbf{v}_{\mathsf{dense}} \cdot \mathbf{v}_{\mathsf{sparse},1} + \mathbf{v}_{\mathsf{sparse},2} = \bigoplus_{i=0}^{\omega-1} \left((\mathbf{v}_{\mathsf{dense}} \lll s_i) \oplus \mathbf{e}_{s'_i} \right)$$

where $\lll$ represents a cyclic left rotation in $\mathbb{F}_2^n$ and $\mathbf{e}_j$ refers to the j-th unit vector.

In practice, each rotation is performed word-serially since the dense vector is usually stored in a Random-Access Memory (RAM) block. Assuming a word size of w, a single rotation therefore takes about $m = \lceil n/w \rceil$ cycles. When attempting to mask this routine, the secret-dependent rotations are problematic since (1) the RAM read/write addresses depend on the rotation offsets and (2) the rotated public vector directly reveals s_i.

Table 1. Latency and RAM demand of multiplication using the masked barrel shifter approach [DR24] for all HQC parameter sets assuming a 64-bit RAM data width, both with (p) and without (s) pipelining across support indices, compared to the average latency of our approach for $C = 1$ and $C = 4$ (see Section 3.3).

Variant	n	ω	Masked barrel shifter			This work		
			Lat. (s)	Lat. (p)	RAM [kB]	Lat. ($C = 1$)	Lat. ($C = 4$)	RAM [kB]
HQC-1	17 669	66	274 230	22 160	15.5 / 68.7	19 054	4 954	6.6 / 13.3
HQC-3	35 851	100	897 600	64 515	31.4 / 148.1	57 246	14 358	13.5 / 26.9
HQC-5	57 637	131	1 888 496	131 546	50.5 / 237.9	119 518	30 154	21.6 / 43.2

3.2 Analysis of Previous Masked Designs

Demange and Rossi [DR24] proposed different ways of implementing a masked sparse-dense polynomial multiplication routine for software implementations of BIKE. They either convert the sparse vector to a dense representation and apply a Boolean-masked multiplication, or utilize a Boolean-masked barrel shifter to implement the Rotate-and-Add technique. Similarly, Banegas et *al.* [BSZ26] suggest converting the sparse vector to a Boolean-masked dense vector, so that each masked product requires two full-length unmasked multiplications.

While the barrel shifter technique by Demange and Rossi would technically also be applicable to hardware implementations, it is only practical when storing the dense vector in a large register, which would require $2 \cdot n$ flip-flops in addition to the area and randomness demand of the masked AND-gadgets in each shifter stage. When operating over a regular w-bit single-port RAM instead, each of the $\log_2 n$ rotation stages requires $\lceil n/w \rceil$ clock cycles, and pipelining between subsequent support entries, which would reduce the total latency to $(\lceil \log_2 n \rceil + \omega - 1) \cdot \lceil n/w \rceil$, requires a separate result RAM for each rotation stage. Therefore, we conclude that this approach either introduces a significant latency or a significant area overhead, as shown in Table 1.

3.3 Proposal of a Hardware-Friendly Masked Multiplication Technique

Instead of relying on a Boolean sharing of the support indices, we convert them to an arithmetic sharing modulo n, such that $s_i = s_{i,A}^{(0)} + s_{i,A}^{(1)} \mod n$. The core idea is that rotation by s_i is equivalent to subsequent rotation by both arithmetic shares individually, i.e., $\mathbf{v}_{\mathsf{dense}} \lll s_i = \left(\mathbf{v}_{\mathsf{dense}} \lll s_{i,A}^{(0)}\right) \lll s_{i,A}^{(1)}$. Since each rotation offset is now statistically independent of the secret support entry, RAM addresses can be considered public. However, this alone would not be secure, considering issue (2), since an attacker can simply compare the rotated dense vector against the non-rotated version to recover s_i. Therefore, the dense vector needs to be Boolean-masked after the first rotation is performed. Intuitively, an adversary then either needs to probe all arithmetic shares of s_i directly or all Boolean shares of the rotated vector in order to recover the rotation offset, as we prove in Section 3.5.

Furthermore, the addition of the second sparse secret vector $\mathbf{v}_{\mathsf{sparse},2}$ to the multiplication result can be efficiently integrated into this routine. As the weight of both sparse vectors is identical in HQC, each iteration can be assigned a pair of support indices (s_i, s'_i) from both vectors. We also convert the value s'_i to an arithmetic sharing modulo n, choosing the same random share as for s_i. As a result, $s_i = s_{i,A}^{(0)} + s_A^{(1)} \mod n$ and $s'_i = s'^{(0)}_{i,A} + s_A^{(1)} \mod n$ with $s_A^{(1)}$ being the common arithmetic share. Addition is now implemented by XORing a single 1-bit at position $s'^{(0)}_{i,A}$ after already having performed the rotation of the public vector by $s_{i,A}^{(0)}$. Since the position of the one bit is a single share which is independent of the secret support entry, this insertion can happen unmasked by flipping the respective bit in one share of the Boolean-masked intermediate rotation output. After this intermediate

Algorithm 2: SecSparseDenseMulAdd (first order, parallelism C)

Input: $\mathbf{v}_{\mathsf{dense}} \in \mathbb{F}_2^n$; Boolean sharings $(s_i^{(0)}, s_i^{(1)})$ and $(s'^{(0)}_i, s'^{(1)}_i)$ of the support indices of $\mathbf{v}_{\mathsf{sparse},1}$ and $\mathbf{v}_{\mathsf{sparse},2}$ for $i \in [0, \omega - 1]$

Output: Boolean sharing $(\mathbf{acc}^{(0)}, \mathbf{acc}^{(1)})$ of $\mathbf{v}_{\mathsf{dense}} \cdot \mathbf{v}_{\mathsf{sparse},1} + \mathbf{v}_{\mathsf{sparse},2} \bmod (x^n - 1)$

$(\mathbf{acc}^{(0)}, \mathbf{acc}^{(1)}) \leftarrow (\mathbf{0}, \mathbf{0})$

for_time $k \in [0, (\omega/C) - 1]$ **do**

  $s_A^{(1)} \xleftarrow{\$} \mathbb{Z}_n$ `// Sample common arithmetic share`

  for_time $r \in [0, C-1]$ with $i = kC + r$ **do**

    $s_{i,A}^{(0)} \leftarrow \mathtt{Unmask}(\mathtt{SubModN}((s_i^{(0)}, s_i^{(1)}), s_A^{(1)}))$ `// Perform B2A conversions`

    $s'^{(0)}_{i,A} \leftarrow \mathtt{Unmask}(\mathtt{SubModN}((s'^{(0)}_i, s'^{(1)}_i), s_A^{(1)}))$

  end

  for_area $r \in [0, C-1]$ with $i = kC + r$ **do**

    $\boldsymbol{\mu}_r \leftarrow$ fresh random mask

    $\mathbf{t}_r \leftarrow \mathtt{Reg}[(\mathbf{v}_{\mathsf{dense}} \lll s_{i,A}^{(0)}) \oplus \mathtt{Reg}[\mathbf{e}_{s'^{(0)}_{i,A}} \oplus \boldsymbol{\mu}_r]]$

  end

  `// Combine parallel lanes and rotate + accumulate result shares`

  $\mathbf{acc}^{(0)} \leftarrow \mathbf{acc}^{(0)} \oplus \left((\bigoplus_r \mathbf{t}_r) \lll s_A^{(1)}\right)$, $\mathbf{acc}^{(1)} \leftarrow \mathbf{acc}^{(1)} \oplus \left((\bigoplus_r \boldsymbol{\mu}_r) \lll s_A^{(1)}\right)$

end

return $(\mathbf{acc}^{(0)}, \mathbf{acc}^{(1)})$

vector is subsequently rotated by $s_A^{(1)}$ positions, the inserted bit will end up at position $s'^{(0)}_{i,A} + s_A^{(1)} \mod n = s'_i$. Integrated handling of the addition saves the otherwise required masked sparse-to-dense conversion and a subsequent sweep over the full vector.

Lastly, to further reduce the latency, we process C different support indices at once using a common random share during the conversion. Then, we rotate the public dense vector by share 0 of each support entry independently, which requires C read ports on the dense vector RAM as well as C instances of the rotation and insertion logic. Due to the fact that share 1 is identical across all entries in this chunk, we can combine the outputs of the first rotation share-wise using an XOR operation before performing the second rotation, since $\left(t \lll s_A^{(1)}\right) \oplus \left(t' \lll s_A^{(1)}\right) = (t \oplus t') \lll s_A^{(1)}$. As a result, we only need $\lceil \omega/C \rceil$ iterations. For ease of presentation, we assume that C divides ω in the rest of the paper, while our implementation fills the last chunk with dummy entries otherwise.

We summarize our multiplication procedure in Algorithm 2. To differentiate between sequential execution and instantiation of multiple parallel hardware modules, we use `for_time` and `for_area`, respectively. Registers which are explicitly required for security reasons are denoted as `Reg[·]`. While each individual arithmetic share is uniformly distributed, gaining knowledge of the first share of multiple support vector entries in the same chunk would leak the difference between the unshared values, due to the common second share. However, we show in Section 3.5 that no single probe can propagate to two such values, and detail why this limits our approach to first-order masking in Section 3.6.

3.4 Hardware Architecture

Figure 1 provides an overview of our hardware architecture. Internally, the design consists of a masking conversion stage, which transforms the Boolean-shared sparse support entries to arithmetic sharings, which are then used by the C-many read branches to perform the first dense vector rotation. In order to create a Boolean sharing of the rotated dense vector, the mask generator generates 64-bit mask words for each read branch. The

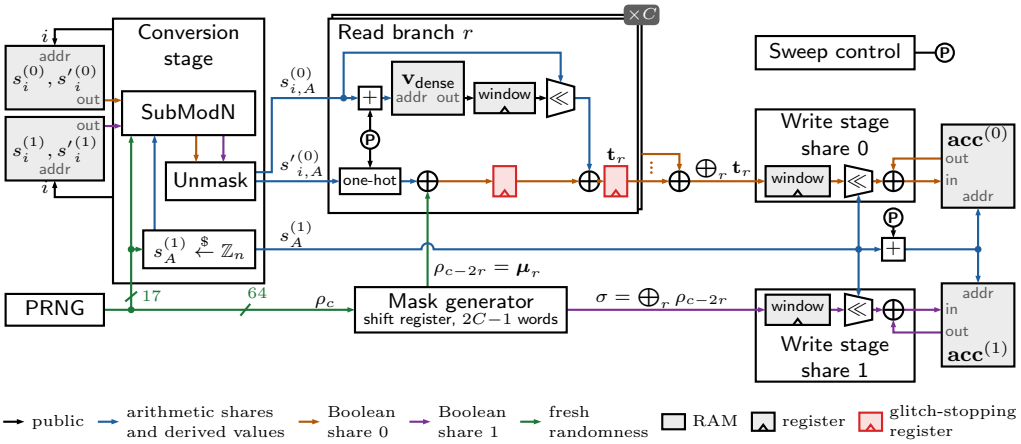


Figure 1. Hardware architecture of the masked multiplication design. One read branch is instantiated for each sparse support entry which is processed in parallel, i.e., C many instances in total. The sweep control module distributes the current word position P to all markers labeled **P**.

accumulated output of all read branches forms the input of the share 0 write stage, while the accumulated mask words are used as input to the share 1 write stage. As rotations must happen word-serially, the sweep control module keeps track of the progress of the word-level sweep over the full dense vector and generates the read and write addresses.

Rotation sweep handling. In every cycle, the sweep control module advances the public position pointer P, representing the index of the current word between the two rotations. The read branch adds a word offset based on $s_{i,A}^{(0)}$ to derive the read address for the dense vector RAM. Read words are used to populate a small internal buffer containing a continuous 3-word window, from which the 64-bit output word is extracted based on the bit-level rotation offset, i.e., $s_{i,A}^{(0)} \mod 64$. Without this buffer, each dense vector word would need to be read twice. The read branch further uses the position pointer to identify whether the single 1-bit from the second sparse vector needs to be added to the currently processed output word. If yes, a 64-bit one-hot vector encoding this bit is generated, combined with the random Boolean mask, and XORed with the extracted part of the window. As we will describe in Section 3.5, the two registers highlighted in red are required to stop a glitch-extended probe from gaining information about either both individual Boolean shares or the secret-dependent difference between multiple support entries. The share-wise write stages operate similarly by generating the result RAM address based on P and the second rotation offset $s_A^{(1)}$, using the same window buffer technique, and update the accumulators with a read-modify-write operation.

Conversion stage. First-order B2A masking conversions commonly consist of just a single subtractor and a subsequent opening or unmasking routine [CGV14, BBE+18, BC22]. We optimize our conversion module based on two observations: (1) We can already convert the next chunk while processing the rotation sweep of the previous chunk, so that the conversion does not affect our latency as long as it finishes within one sweep. (2) Since all entries use a common arithmetic share, a fully pipelined design can only be securely supplied with new inputs every two cycles. Otherwise, transitional leakage on the output of the unmasking step would leak $\mathsf{HD}\left((s - s_A^{(1)}), (s' - s_A^{(1)})\right)$ for two secret support indices s and s'. As a result, we design a partially pipelined SubModN module based on a single, pipelined ripple-carry structure using 16 Boolean-masked full adders built from HPC1 gadgets [CGLS21], which computes both $s - s_A^{(1)}$ as well as the subsequent conditional

addition of the modulus n. Using each full adder twice per input, this module can only receive fresh inputs every second clock cycle by design, avoiding the area and randomness overhead of a fully pipelined modular subtractor which would only be half utilized. We also share the blinding randomness of all HPC1 gadgets, as proven secure for first-order masked designs by Hadžić and Bloem [HB25]. To avoid transitional leakage due to intermediate subtraction outputs reaching the unmasking stage, this path is gated, so that the output register of the unmasking step alternates between 0 and a valid B2A conversion output.

Optimized Boolean mask handling. When implemented naïvely, each of the C read branches needs its own fresh 64-bit mask so that they do not cancel out when being combined. Instead we propose to use a single mask stream of 64-bit words ρ_c in every cycle c, which is used as input to a shift register containing $2 \cdot C - 1$ many words. We place taps at every second position of that shift register so that read branch r uses the independent mask word $\mu_r = \rho_{c-2r}$. Due to the stride of two, subsequent clock cycles use disjoint masks and a transition-extended probe cannot remove any masks.

Latency and Randomness Requirements. Each chunk is processed in $m + 11$ cycles, where $m = \lceil n/64 \rceil$, and only the B2A conversions of the initial chunk add to the latency. As a result, a multiplication requires on average $\lceil \omega/C \rceil \cdot (m+11) + 4 \cdot C + 40$ clock cycles, as the rejection sampling of $s_A^{(1)}$ takes a variable number of cycles. These additional 11 cycles per chunk correspond to at most 4 % overhead compared to the latency of an unmasked word-serial Rotate-and-Add multiplier with the same RAM data width and parallelism. Furthermore, the multiplication consumes only 81 fresh random bits per clock cycle, 64 for mask words and 17 for the SubModN module which are reused for rejection sampling since these two operations happen strictly sequentially. Due to the optimized Boolean mask handling, this randomness demand does not increase for higher parallelism degrees.

3.5 Security Analysis

In this section we prove that our masked multiplication is glitch- and transition-robustly first-order probing secure and 1-PINI, allowing for its integration into our HQC design.

Security model and assumptions. We consider the robust probing model [FGP+18] with glitch- and transition-extended probes. A probe on a wire in clock cycle c yields the values in cycles c and $c - 1$ of all registers, RAM read data ports, and primary inputs in its combinational fan-in. Probing a RAM port yields the address and the data word, but no other words stored in the same memory. In our architecture, each RAM holds at most values from one share domain. We make two assumptions:

(1) All fresh masking randomness bits are uniform and independent in every clock cycle. Unless stated otherwise in Section 3.4, all random bits have a single consumer.

(2) With $s_A^{(1)}$ and n treated as public operands, the SubModN module is robustly 1-PINI and for every fixed input sharing, one output share is uniformly distributed over the masking randomness. Since masking conversions have been studied extensively, we omit a full gadget-level proof, but verified both properties through exhaustive gate-level simulation on a reduced-width ($w_{\mathsf{sub}} = 4$) instance of the hardware module.

Design properties. Our proof relies on the following five architectural properties, which hold by construction. We note that a *mask word* refers to a combination of words taken from the mask stream while a *masked word* refers to data XORed with a mask word.

(P1) No extended probe observes both a mask word as well as a masked word, i.e., both Boolean shares of a secret value.

(P2) In any two consecutive cycles, the registers that serve as inputs to the unmasking step do not contain values from different B2A conversion operations.

(P3) No extended probe observes unmasked values computed from arithmetic shares of two different B2A conversions belonging to the same chunk of support indices, e.g., $s^{(0)}_{0,A}$ and $s^{(0)}_{1,A}$. Similarly, no extended probe observes a value computed from $s^{(1)}_{A}$ together with an unmasked value computed from any $s^{(0)}_{i,A}$ or $s'^{(0)}_{i,A}$.

(P4) The read branch with index r uses mask word ρ_{c-2r} in clock cycle c. Furthermore, the two registers in each read branch which are highlighted in red within Figure 1, as well as the mask shift register are updated in every clock cycle.

(P5) An inserted bit of the second sparse vector $\mathbf{v}_{\mathsf{sparse},2}$ only meets the rotated public dense vector word after being masked and registered.

Both Boolean shares use separate data paths and memories, and the output register of the read branch stops glitches from propagating back to values before the mask application (P1). The gating in front of the unmasking step ensures (P2). The read branches only receive their respective pair of $s^{(0)}_{i,A}$ and $s'^{(0)}_{i,A}$, which no extended probe can see together due to the masking of the one-hot vector, while the write stage only receives $s^{(1)}_{A}$ (P3). (P4) holds based on the placement of the shift register taps, and the register after XORing the one-hot vector with the mask word ensures (P5).

Lemma 1 (Blinding using a common share)**.** *For every chunk, let $s^{(1)}_{A}$ be sampled uniformly and independently of all secrets from $\mathbb{Z}_n$. Two properties hold: (i) For every possible value of a support index $s \in \mathbb{Z}_n$, the arithmetic share $s^{(0)}_{A} = s - s^{(1)}_{A} \mod n$ is uniformly distributed and independent of s. (ii) Across two different chunks, the arithmetic shares $s^{(0)}_{i,A}$ and $s'^{(0)}_{i,A}$ for all i within the relevant chunks are independent.*

Proof. Property (i) follows trivially since $s^{(0)}_{A} \mapsto s - s^{(1)}_{A} \mod n$ is a bijection on $\mathbb{Z}_n$. In addition, (ii) holds since $s^{(1)}_{A}$ is freshly and independently sampled for each chunk. □

Lemma 2 (Independence of reused mask words)**.** *Let $(\rho_c)_{c\in\mathbb{Z}}$ be independent and uniformly distributed words, representing the 64-bit PRNG output in every clock cycle as displayed in Figure 1, and let $\sigma(c) = \bigoplus_{r=0}^{C-1} \rho_{c-2r}$ be the combined mask word for every chunk. It holds: (i) For any set of distinct cycles $c_1, ..., c_k$, the combined mask words $\sigma(c_1), ..., \sigma(c_k)$ are mutually independent and uniformly distributed. (ii) For every cycle c, the $2C$-many PRNG words ρ_{c-2r} and $\rho_{(c+1)-2r}$ for $r \in [0, C-1]$, i.e., all mask words used by the individual read branches in cycles c and $c+1$, together with all $\sigma(c')$ for $c' < c$, are mutually independent and uniformly distributed.*

Proof. We use the following observation: If each combined word consisting of the XOR of multiple input words ρ_e contains at least one word ρ_u that occurs in no previous one, they are all mutually independent and uniformly distributed. In our case, the lowest index of a word that contributes to $\sigma(c')$ is $c' - 2(C-1)$. (i) Assuming that $c_1 > c_2 > \cdots > c_k$, each $\sigma(c_{j+1})$ contains the mask word $\rho_{c_{j+1}-2(C-1)}$, which has a lower index than all words in $\sigma(c_1), \ldots, \sigma(c_j)$. (ii) The $2C$ words ρ_{c-2r} and $\rho_{(c+1)-2r}$ for $r \in [0, C-1]$ are trivially distinct and therefore independent and uniformly distributed. By the same argument as for (i), each $\sigma(c')$ for $c' < c$ contains the mask word $\rho_{c'-2(C-1)}$ which has a lower index than all $2C$ branch mask words as well as all combined mask words in subsequent cycles. □

Intuitively, property (ii) states that two consecutive values of the read branch output registers never share a mask word and that the combined words are masked independently of all previous words in the window buffers.

Theorem 1 (Robust first-order security)**.** *Under the previously described assumptions, our proposed masked multiplication routine is glitch- and transition-robust 1-probing secure as well as 1-PINI. This holds for every parallelism configuration $C \geq 1$, every HQC parameter set, both with and without the hiding countermeasures as detailed in Section 4.3.*

Proof (sketch). We distinguish five categories of probe locations:

Control logic and common arithmetic share. A probe in the sweep control module, the rejection sampler for $s_A^{(1)}$, or at the address ports of the results $\mathbf{acc}^{(0)}$ or $\mathbf{acc}^{(1)}$ observes only public values, fresh randomness, and $s_A^{(1)}$ of at most two independent chunks. All non-public values can be simulated by returning uniformly random values.

SubModN module. Simulatability follows from assumption 2 (SubModN is 1-PINI).

Unmasking logic and read branches. This category contains probes in the unmasking step, registers holding $s_{i,A}^{(0)}$ or $s'^{(0)}_{i,A}$, the address and data ports of the dense vector RAM, the window buffers and output word extraction logic, the one-hot generation logic, and the masked one-hot vector insertion registers. By (P2), a probe on the unmasking step observes the outputs of at most one conversion operation, i.e., a single uniformly distributed arithmetic share. By (P3), all other probe locations in this category observe unmasked values computed from at most one arithmetic share in a chunk and never in combination with $s_A^{(1)}$. Following Lemma 1, these arithmetic shares are uniform and independent, so the simulator samples them and computes all derived values. By assumption 2, the Boolean output shares of SubModN are simulated by a uniform share and its XOR with the sampled arithmetic share. The remaining values in this category are fresh mask words or the masked one-hot vector insertion registers. From (P4) and (P5), it follows that the latter is a uniformly random Boolean share, which is simulated by a uniform bit string.

Boolean share 0 data path. This category contains the branch output registers, combination / accumulation logic, the share 0 write stage, and $\mathbf{acc}^{(0)}$. By (P1), only values belonging to Boolean share domain 0 are observed. By (P4), a transition-extended probe on the branch output register yields values from two consecutive clock cycles, which are masked with ρ_{c-2r} and $\rho_{(c+1)-2r}$ with $r \in [0, C-1]$. According to Lemma 2, these masks, as well as the combined masks which are visible in a transition within the write stage, are independent and uniform, meaning that no masks cancel out. Therefore, the simulator samples all mask words and all combined words from previous cycles uniformly and computes all other values from them, $\mathbf{acc}^{(0)}$, and $s_A^{(1)}$.

Boolean share 1 data path. Analogously, a probe on the mask generator, σ, the share 1 write stage, or $\mathbf{acc}^{(1)}$ only observes values of Boolean share domain 1 (P1), which the simulator computes from uniformly sampled masks, $s_A^{(1)}$, and $\mathbf{acc}^{(1)}$.

PINI argument. Only the simulators for probes in the SubModN module or the two Boolean share data paths require access to input shares, and only to the share domain of the given region. Lastly, the hiding countermeasures of Section 4.3 only change the order of the rotation sweep in the public sweep control unit. Since (P1)–(P5) hold irrespective of this, the proof stays valid. The same holds for increased parallelism, since all additional read branches act on independent entries and are only combined after masking with independent and uniform masks (Lemma 2). □

3.6 Generalization to Higher Orders

We now sketch how our technique can be extended to $d > 1$ without claiming to have a formal proof. At order d, each support index is split into $d+1$ arithmetic shares. The unmasked dense vector has to be split into $d+1$ Boolean shares after its first rotation,

before it needs to be refreshed after each subsequent rotation to prevent an attacker from recovering the sum of multiple arithmetic shares by comparing shares across rotations.

However, two aspects which made our implementation efficient do not trivially translate to higher orders. First, we require at least d rotations before the insertion of the $\mathbf{v}_{\mathsf{sparse},2}$ entry and at least d rotations afterward, to prevent an adversary from recovering the secret by probing the insertion point in the partially rotated vector as well as all following (or preceding) rotation offsets. Since the total number of rotations is $d+1$, we need $2 \cdot d \leq (d+1)$, which only holds for $d = 1$. Therefore, addition of $\mathbf{v}_{\mathsf{sparse},2}$ requires either an artificially inflated arithmetic share count or a separate pass. Second, it is no longer possible to choose all but one shares identical across all entries. Otherwise, an adversary could probe the non-common shares of two entries and recover the difference between both secret values. As a result, each sparse entry needs its own arithmetic sharing, so that branch results can only be combined right before the last write stage.

This increases the memory requirements as each of the d rotations after the first one produces a partially rotated $(d+1)$-share vector that needs to be stored. With one rotation on the read side and one on the write side of a RAM, the number of memory blocks would be approximately $C \cdot (d+1) \cdot \lceil (d+1)/2 \rceil$.

4 Horizontal Attacks on the Masked Multiplication

As recently demonstrated by Hesse et *al.* [HFG+26], theoretical probing security can be insufficient to guarantee practical side-channel security, especially for PQC primitives. In this section, we show that our multiplication without hiding is vulnerable to a practical horizontal attack and integrate hiding countermeasures.

4.1 Susceptibility to Horizontal Attacks

In our design, the RAM read and write addresses are determined by the arithmetic shares $s_{i,A}^{(0)}$ and $s_A^{(1)}$, which are used in all m cycles of a sweep, i.e., 277 cycles in the case of HQC-1. An adversary can therefore combine information from all sub-traces of a sweep to estimate both shares from a single trace. This is not specific to our design since any multiplier based on a Rotate-and-Add technique, including the masked barrel shifter concept by Demange and Rossi [DR24], uses the same secret rotation offset over a large number of cycles. For unmasked implementations, Sim et *al.* [SKC+19] have demonstrated such a single-trace attack in software, while Beckwith et *al.* [BZKG24] managed to recover the key from such multiplication leakage in a hardware implementation of BIKE. TVLA does not detect such forms of leakage since it requires the combination of information from multiple sample points within a single trace. Consequently, our multiplier without hiding shows no first-order leakage after 10 million traces (cf. Section 6.4), although the following attack manages to recover all support indices from just 200 traces.

4.2 Analysis of a Practical Horizontal Attack

We demonstrate a practical attack against our first-order masked multiplication on an FPGA, using the setup of Section 6.1. For this attack, we consider a profiling adversary, who has access to an identical target device with known secrets and full control over all inputs, including masking randomness, and uses 300 profiling traces. The adversary can further choose the dense vector, which is the case for $\mathbf{u}$ during the computation of $\mathbf{u} \cdot \mathbf{y}$ in the decapsulation, since it directly comes from the ciphertext. While the Hamming distance between consecutive words loaded into the read branch window buffer only weakly depends on the secret for a random dense vector, we can enhance this leakage by setting every 64-bit word of $\mathbf{u}$ to either all zeros or all ones in a specific pattern to identify the

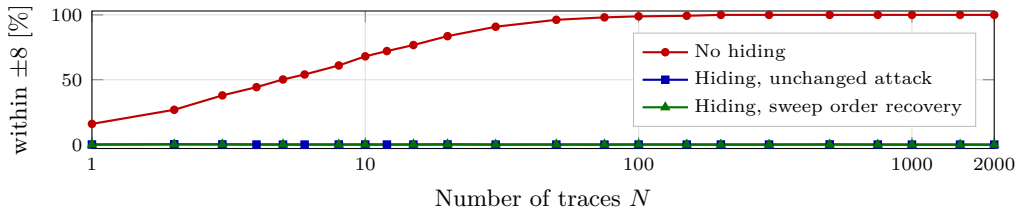


Figure 2. Percentage of the 66 support indices of HQC-1 within ±8 positions of the correct value after N traces, averaged over 50 executions of the attack.

current position within the vector. While this allows for recovery of $s_{i,A}^{(0)}$, we utilize leakage during the write address computation to recover the other share $s_A^{(1)}$.

We therefore construct templates using linear regression [SLP05] for the Hamming distances of two subsequent read words and for the write address, which we use to compute the likelihood for every value of $s_{i,A}^{(0)}$ and $s_A^{(1)}$, yielding the likelihood of each $s_i = s_{i,A}^{(0)} + s_A^{(1)} \mod n$. Since the values s_i stay the same across multiple invocations while only the arithmetic sharing changes, we can combine information from multiple traces.

As highlighted in Figure 2, 98.9 % of the 66 support indices were recovered within ±8 positions after 100 traces, and all of them starting at 200 attack traces. While we do not perform full key recovery, we want to highlight that this is a variant of the decoding with side-channel hints problem, which has been analyzed in the past [RHHM17, BDK+25].

4.3 Efficient Integration of Hiding Countermeasures

The previous attack relies on the fact that the rotation sweep over the dense vector uses a fixed public order. However, full shuffling is not practical since the window buffer relies on sequential access to avoid reading each input word multiple times. Instead, we propose to randomize the order in a coarser way by dividing the sweep into two segments whose start position is randomized. More specifically, the first segment can start at any word $t_1 \in [1, m-1]$ while the second one starts at $t_2 = t_1 + \lfloor m/2 \rfloor + j \mod m$, where j is a random 3-bit jitter. The sweep of both segments is then randomly interleaved based on a random select bit in every cycle. The start position of the second segment is deliberately chosen in a way so that both segments are approximately equal in size. Otherwise, the interleaved sweep would again degenerate to a linear one as soon as one of the sweep segments is done. Since the state of the window buffer needs to be retained for both sweep segments, it needs to be duplicated both on the read side as well as on the write side.

This countermeasure adds five cycles per chunk, increasing the overall latency by at most 1.7 %, and requires one more random bit. We report its area overhead in Section 6.2.

4.4 Evaluation of Hiding Countermeasures

As visualized in Figure 2, when applying the unmodified attack to the implementation with hiding enabled, the success rate drops to approximately 0.3 %, which is close to chance level. A different version of the attack which tries to recover the interleaved sweep order using potential leakage from the read address computation also did not succeed in recovering the individual shares after up to 2 000 attack traces. While we cannot definitively conclude that the resulting design is secure against all potential horizontal attack vectors, our implemented countermeasures succeed in preventing the analyzed attacks and are expected to make key recovery significantly harder in practice.

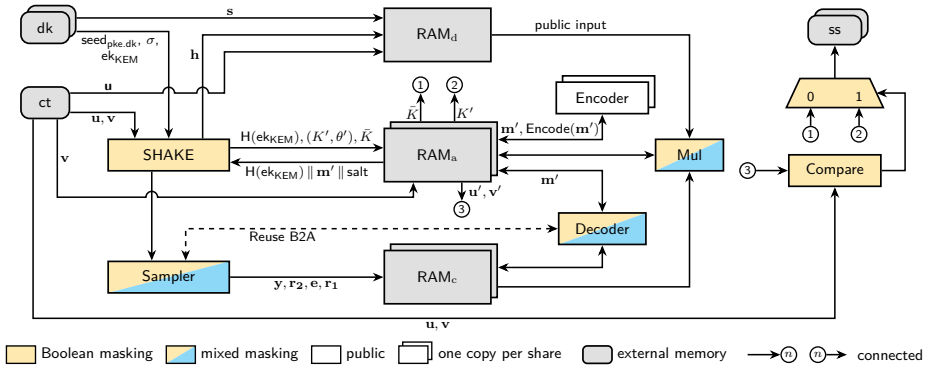


Figure 3. General architecture of the HQC decapsulation

5 HQC Design

We now describe the remaining hardware architecture of our HQC design, focusing on the decapsulation. Key generation and encapsulation use the same general architecture, but omit some modules such as the decoder.

5.1 General Architecture

An overview of the decapsulation architecture is shown in Figure 3. We assume that ciphertext, secret key, and shared secret are stored in external RAM blocks with a data width of 64 bit, which allows the hash module to read or write an entire lane in one clock cycle. In addition, we use three types of internal RAMs. The public single-port RAM_d with a data width of 64 bit stores all public data inputs of the multiplication and is replicated according to the multiplication parallelism (cf. Section 3). Furthermore, we use the secret, dual-port RAM_a with a data width of 64 bit as an accumulator for the multiplication and to store values such as $\mathbf{m}'$ or K' for later use. Finally, the secret, dual-port RAM_c has a data width of 16 bit and stores the non-zero coefficients of the sparse polynomials as well as intermediate values of the Reed-Muller decoding.

All hashing operations are performed by a single generic SHA / SHAKE. The two fixed-weight vector sampling modules, combined into a single module in Figure 3, utilize an A2B and a B2A conversion, which can optionally be reused by the decoder (cf. Section 5.2 and Section 5.3). The comparison module compares $(\mathbf{u}, \mathbf{v})$ and $(\mathbf{u}', \mathbf{v}')$ in chunks of 64 bits and outputs 1 iff both inputs were identical. All comparison results are accumulated using a single Boolean masked AND-gadget, and the final multiplexer uses this correctness bit to choose between $\bar{K}$ and K'.

5.2 Fixed-Weight Vector Sampling

While both samplers (cf. Figure 4) have a different structure, they both require a B2A and an A2B conversion since they perform field arithmetic on Boolean masked data before the duplicate checks are done in Boolean domain again. As the two samplers are never active at the same time, they share both conversion modules.

5.2.1 SampleFixedWeightVect$

The 24-bit XOF output x is first transformed to an arithmetic sharing $\hat{x}$ using the B2A conversion modulo 2^{32}. While the key generation can use a smaller conversion modulo 2^{24}, the decapsulation pads x with 8 zero-bits and discards the upper 8 bits of $\hat{x}$. Following the conversion, both shares $\hat{x}_0$ and $\hat{x}_1$ are individually reduced modulo n using a Barrett

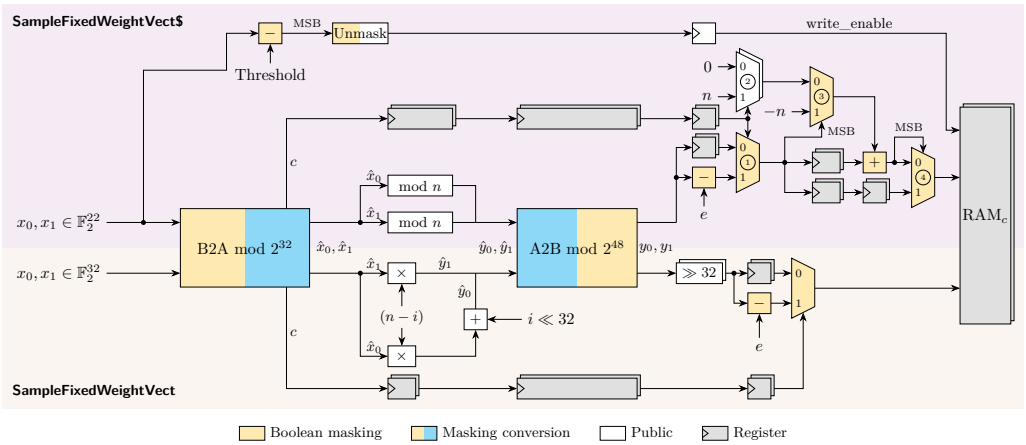


Figure 4. Architecture of the two fixed-weight vector samplers, which share both conversions.

reduction, which guarantees that the unshared value $\hat{x}$ is smaller than $2n$, which we can resolve with a conditional subtraction of n after the A2B conversion.

However, performing a reduction modulo n results in the shares leaving their designated modular field. As a consequence, the carry that may occur when unsharing $\hat{x}$ and that is normally implicitly handled by the reduction modulo 2^{24} becomes an issue since the value of 2^{24} is inherently present in the two shares and is further processed by the reduction. We can track the error value e through the reduction and subtract it in the end if a carry between shares would occur, which is an information that the B2A conversion trivially provides as the carry of its last addition. The reduction calculates $y = x - ((x \cdot n_\mu)/2^{24}) \cdot n$ for the unreduced input x using the constant n_μ, which becomes $y' = (x + 2^{24}) - (((x + 2^{24}) \cdot n_\mu)/2^{24}) \cdot n$ if a carry occurs. Transforming this equation gives us that $y' = y + (2^{24} - n \cdot n_\mu)$, i.e., $e = 2^{24} - n \cdot n_\mu$ in case of a carry.

After the A2B conversion, we first correct the error due to a carry by conditionally subtracting e (Mux 1) and then conditionally subtract n to finalize the reduction. However, by reducing both shares individually, we may also have subtracted n once too much, which is indicated by its carry-bit and can only occur without a carry during the B2A conversion. As a consequence, we choose between 0 and n using the carry from the conversion, which we can do sharewise as both inputs are public (Mux 2), and between this result and $-n$ using the carry from the subtraction (Mux 3), add it to the output of the subtraction, and select the final result using the resulting carry (Mux 4).

In parallel, we subtract the sampling threshold $\lfloor \frac{2^{24}}{n} \rfloor \cdot n$ from the XOF output and unmask the carry-bit of this subtraction, which indicates whether the sample is valid. Unmasking this bit poses no threat for security as it conveys no information about the current Keccak state or other sampled values. The unmasked valid bit is used as a write-enable to RAM$_c$, and to increment the coefficient counter.

To reduce the latency of the sampling process, we implement the sampler as a fully pipelined module, which also requires us to perform the check for duplicates *after* all coefficients have been sampled. In addition, we always reduce and store all available outputs from the current hash state, which costs a few clock cycles, but allows us to use the redundant coefficients as replacements for potential collisions.

Once a sufficient number of coefficients has been sampled into RAM$_c$, we detect duplicates using a Boolean masked comparator and replace them with one of the redundant sampled coefficients. In the rare case that more redundant coefficients are needed, we perform another set of reductions.

5.2.2 SampleFixedWeightVect

This sampler first converts the 32-bit XOF output to arithmetic domain and multiplies both shares by $n - i$, which can be done sharewise as this is a public value. Then, the value i, shifted 32 positions to the left, is added to the first share, which is more efficient than adding i in Boolean domain. Afterwards, the result, which can have a size of up to 48 bits, is converted back to Boolean domain and shifted 32 positions to the right.

Similarly to the other sampler, a carry in the shares after the B2A conversion would lead to a wrong sampling result. In this case, the result y is calculated as $y' = i + ((x + 2^{32}) \cdot (n - i)) >> 32 \Leftrightarrow y' = y + (n - i)$. Therefore, we conditionally subtract the error $e = (n - i)$ in the end, using the carry as a select bit. During the elimination of duplicates, a coefficient that collides is replaced with the index of the other colliding coefficient.

5.3 Encoding and Decoding

As Reed-Solomon and Reed-Muller codes are linear over $\mathbb{F}_2$, we can encode each share individually. The Reed-Solomon encoding divides $\mathbf{m}$ by the fixed generator polynomial using an Linear Feedback Shift Register (LFSR) with $2 \cdot \delta$ multipliers over $\mathbb{F}_{2^8}$, and each output byte is immediately encoded by the RM-code using a chain of XORs and replicated.

While the decoding is also linear and could thus be applied share-wise, this linearity can only be used if both shares are valid codewords. However, the input to the decoder is the result of a multiplication with numerous re-randomizations, so neither share is guaranteed to be a valid codeword and the decoder has to be fully masked.

5.3.1 Reed-Muller Decoding

Reed-Muller decoding first requires sums all replicated bits before applying the Hadamard transform. As both operations are linear over an arithmetic sharing, we convert every codeword bit to arithmetic domain using B2A conversions, instantiating between 1 and 8 conversions or reusing the B2A conversion from the samplers. After conversion, all bits are summed and the Hadamard transform is applied sharewise. We always execute an entire layer of the Hadamard transform in one clock cycle, which significantly reduces the required latency and removes the need for complex address generation logic.

After the Hadamard transform, we have to find the index of the highest absolute value from the previous results. We first transform all 128 values back to Boolean domain, optionally reusing the conversion from the samplers, before performing the comparisons by always comparing two values and only keeping the larger absolute value together with its index, which enables us to do all comparisons in a pipelined way. For each comparison, we first compute the absolute value of both candidates by XORing the sign bit to each of the other bits, followed by an addition with the sign bit. The two positive values are then subtracted, and the maximum absolute value is selected using the carry-bit of the subtraction. Finally, the index of the largest absolute value is stored in $\mathrm{RAM_c}$.

5.3.2 Reed-Solomon Decoding

After decoding all Reed-Muller codewords, the Reed-Solomon decoding starts by calculating all syndromes of the received polynomial using Horner's scheme, which is done sharewise using unmasked multipliers that we also use to find the error locations by evaluating the error-locator polynomial. For Berlekamp's algorithm, the computation of the polynomial Ω, and the error values, we instantiate a single masked multiplier and inverter and use them in a highly serial manner to reduce the area and randomness overhead. Finally, we immediately add each error value to its corresponding codeword byte.

5.4 Generalization to Higher Orders

Most modules described in the previous sections can be trivially generalized to higher orders, including the hash module, decoder, comparator, and all conversions. The main challenge lies in the generalization of the sampling procedures to higher order. An issue that affects both samplers is the carry in arithmetic shares, which is a simple Boolean value with two shares, but can take a range of possible values with more shares, making both the identification and the correction of the error e more complex. In addition, with $d+1$ shares, the sum of all individually reduced shares in SampleFixedWeightVect$ would only be guaranteed to be smaller than $(d+1) \cdot n$, requiring a more complex final reduction.

6 Evaluation

In this section, we evaluate our masked multiplication on its own (Section 6.2), followed by the full HQC designs (Section 6.3) and our side-channel evaluation (Section 6.4).

6.1 Experimental Setup

RTL generation and synthesis. Since our configurable hardware designs are integrated into the HADES framework [BLH+25], we utilize its DSE capabilities to identify the best-performing designs, for example regarding the multiplication parallelism, according to specific performance metrics, i.e., area (A), critical path (C), latency (L), and randomness (R), as well as derived product metrics like the Area-Critical Path-Latency Product (ACLP). As the latency of the multiplication is probabilistic due to the rejection sampling of $s_A^{(1)}$, we report average-case latencies for all modules. Application-Specific Integrated Circuit (ASIC) area metrics are generated using Synopsys Design Compiler S-2021.06-SP4 for the full HQC designs and using yosys 0.69 for the standalone multiplication modules, both targeting the NanGate45 open cell library. The resulting area numbers exclude RAM blocks as the memory demand is instead reported separately. To generate Field-Programmable Gate Array (FPGA) metrics, we use Vivado 2024.2 and report post-route resource utilization and the highest f_{max} at which timing-driven place-and-route closes.

Measurement setup. We perform all TVLA measurements on a ChipWhisperer CW305 target board containing an Artix-7 100T FPGA, clocked at 5 MHz by an external Rigol DG4102 waveform generator, with VCCINT driven directly by a lab power supply. Power consumption of the device is measured over the integrated 100 mΩ shunt resistor using a LeCroy WavePro 254HD-MS oscilloscope with 12-bit quantization, which is phase-synchronized with the target clock. We use a sampling rate of 250 MS/s, i.e., 50 samples per cycle, and capture an FSM trace of the target design via the integrated logic analyzer. Masking randomness is produced via an unrolled Trivium instance with runtime-configurable output gating for setup validation measurements, during which the Pseudorandom Number Generator (PRNG) keeps running to ensure an identical noise level across measurements.

6.2 Performance Evaluation of the Masked Multiplication

The left part of Table 2 reports area and latency for different parallelism values of our multiplication, instantiated for HQC-1. As expected, the latency decreases with increasing parallelism, going from 19.05 kcycles at $C = 1$ to 1.55 kcycles for $C = 16$, while area only grows by a factor of less than 6 since only additional read branches need to be added in terms of logic. However, for larger C, the required copies of $\mathbf{v}_{\mathsf{dense}}$ lead to impractical memory demands. For HQC-3 and HQC-5, the area at $C = 1$ stays at about 25 kGE while the latency increases to 57.2 and 119.5 kcycles, respectively. Hiding increases the latency by

Table 2. Our masked multiplication for HQC-1 without and with hiding. Left: ASIC results (Yosys, NanGate45). Right: FPGA results (Vivado 2024.2, XC7A100T-2) and prior designs.

C	Area [kGE]		RAM	Latency [kcycles]	
	no hid.	hiding	[kbit]	no hid.	hiding
1	24.5	29.8	53.2	19.05	19.39
2	32.0	39.1	70.9	9.55	9.72
4	46.8	57.1	106.4	4.95	5.04
8	77.3	94.1	177.3	2.67	2.71
16	137.3	166.8	319.1	1.55	1.57
32	257.1	313.0	602.8	1.03	1.05*
75	576.2	702.3	1 365.1	0.65	0.67*

Design	LUT	FF	f_{max}	Lat.	Time
			[MHz]	[kcyc.]	[µs]
[DXN+23][a]	620	245	270	18.48	68.5
[JLK+25a][b]	657	159	–	29.33	–
Ours	2 280	1 232	179	19.05	106.7
Ours, hiding	2 817	1 672	169	19.39	114.4

[a] Unmasked, XC7A200T-3.
[b] Hiding only, XC7A200T (unknown speed grade).

Table 3. ASIC results (Synopsys DC, NanGate45) for the all HQC-1 algorithms. Average latency over 100 executions. Memory is excluded from area and reported in the SRAM column.

Algorithm	Opt.	Area	SRAM	Rand.	Lat.	Freq.	Delay
		[kGE]	[kB]	[bit]	[kcycles]	[MHz]	[µs]
Key Generation	ACP	209.1	4.8	1532	64.0	483	132.5
	ACLP	274.3	15.9	1982	11.2	442	25.3
	ACLRP	400.2	15.9	444	13.5	518	26.1
Encapsulation	ACP	260.6	4.9	1956	123.0	377	326.0
	ACLP	356.2	27.0	2306	18.2	461	39.5
	ACLRP	594.0	27.0	456	21.7	375	58.0
Decapsulation	ACP	450.6	5.0	3278	226.8	361	628.2
	ACLP	320.8	18.3	5098	62.3	362	171.9
	ACLRP	793.1	18.3	1210	112.6	315	357.0

1.7 to 1.8 % for $C \leq 16$ and the area by 21 to 22 %. The updated specification [GMA+26] increases the weights by one, which has no significant impact on the performance.

The right part of Table 2 compares our multiplication cores with $C = 1$ against prior designs. Using an identical 64-bit RAM width, our first-order masked design requires only 3 % more cycles than the unprotected design by Deshpande et *al.* [DXN+23], while using 3.7 times more LUTs and 5 times more flip-flops. Furthermore, our masked multiplication is 34 % faster than the hiding-only multiplier by Jeon et *al.* [JLK+25a].

6.3 Performance Evaluation of the Masked HQC Designs

Performance metrics for HQC-1 designs targeting ASICs are reported in Table 3. We want to highlight that all designs run at more than 300 MHz while some even exceed 500 MHz. For all three algorithms, the ACP-optimized design uses a multiplication parallelism of $C = 1$, which results in low area at the expense of high latency. Additionally optimizing for latency increases this parallelism and therefore SRAM utilization, while only having a moderate impact on area and randomness demand. As a result of the faster multiplication, overall latency improves by a factor of 6 for key generation and encapsulation, while decapsulation only improves by a factor of 3 since sampling and decoding outweigh the time spent during multiplication. Lastly, ACLRP-optimized designs use the same multiplication configuration while using only 25 % as much randomness, at the cost of significantly increased area requirements, mostly caused by the usage of inherently larger HPC2 gadgets instead of HPC3 and the additional pipelining registers due to their longer latency. Notably, the ideal multiplication parallelism differs between key generation (6), encapsulation (11), and decapsulation (7), since encapsulation performs twice the amount of multiplications compared to key generation while spending a comparable amount of time sampling values.

Comparison with unprotected designs. We compare the performance for our designs on an Artix-7 FPGA with the unprotected designs of Deshpande et *al.* in Table 4. Since the synthesized designs were generated in a separate DSE run, some configurations can

Table 4. FPGA results (Vivado 2024.2, Artix-7 100T) of our masked HQC-1 designs and of the unprotected designs by Deshpande et *al.* [DXN+23]. Average latency as reported by HADES.

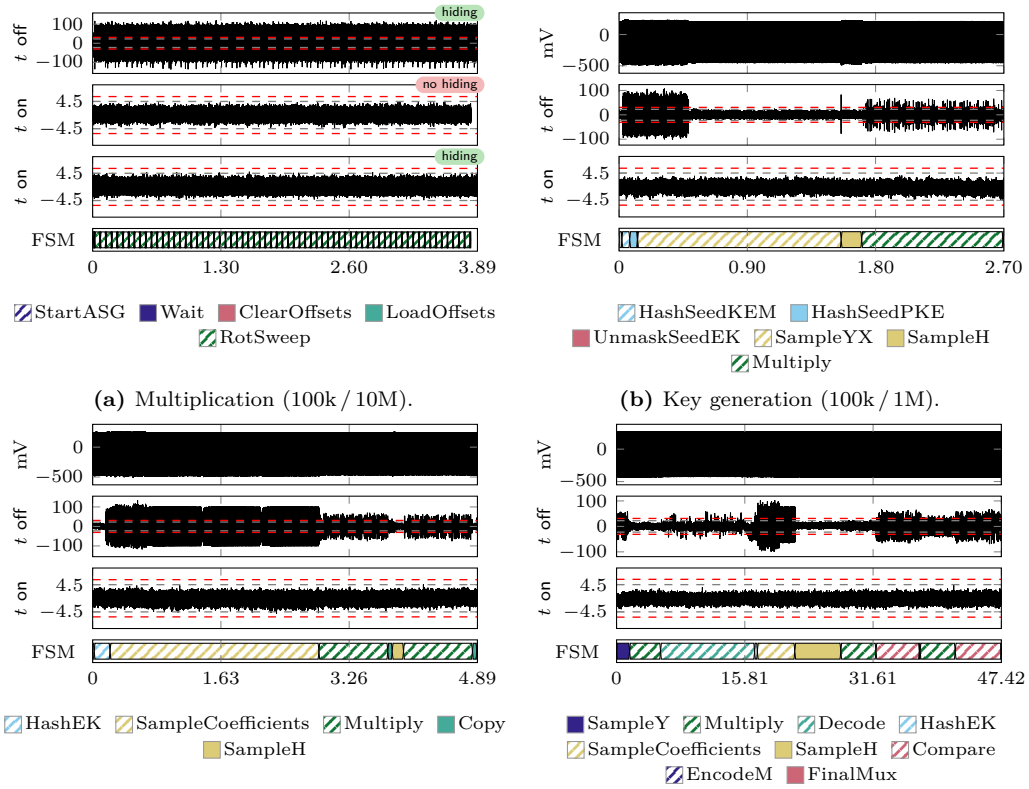
Algorithm	Design	Prot.	C	LUT	FF	BRAM [36 Kb]	f_{max} [MHz]	Lat. [kcycles]	Time [µs]	Rand. [bit]
Key Generation	[DXN+23]	none	–	2 396	901	10	179	15.8	88.0	–
	ACP	HPC3	1	15 578	12 106	9	154	64.05	416.3	1 346
	ACLP	HPC3	6	29 000	18 863	14	133	10.49	78.7	2 382
Encapsulation	[DXN+23]	none	–	5 828	2 531	15	179	22.4	125.3	–
	ACP	HPC2	1	22 578	23 558	8	139	125.4	902.8	1 149
	ACLP	HPC2	7	36 543	30 152	15	125	21.29	170.3	1 426
Decapsulation	[DXN+23]	none	–	10 959	7 003	22	179	37.2	207.8	–
	ACP	HPC2	1	41 163	47 659	10	137	220.8	1 611.9	2 530
	ACLP§	HPC3	5	54 926	45 133	14	105	60.88	578.4	6 768

slightly differ from the ASIC table. We want to highlight that our ACLP-optimized key generation is faster than the unprotected version (78.7 µs compared to 88.0 µs). While the corresponding encapsulation design also offers a lower latency than the unprotected reference, the overall delay is longer due to the lower maximum clock frequency. However, the designs by Deshpande et *al.* were synthesized for a higher speed grade (-3 compared to -2), which shifts the comparison in their favor. Our masked decapsulation routine shows the highest overhead over the unmasked design, requiring 1.6 times the clock cycles and 2.8 times overall execution time while requiring 3.4 to 4.0 times the LUTs and more than 6 times as many flip-flops. However, all of our presented designs use less memory than the design by Deshpande et *al.*. Furthermore, the area for generating masking randomness is excluded in this comparison, which would increase the gap further.

6.4 Security Evaluation

Evaluated designs. First, we evaluate the $C = 1$ multiplication core with and without hiding (see Table 2) before we evaluate the ACLP-optimized configurations of the key generation, encapsulation, and decapsulation routine for HQC-1. These configurations utilize HPC2 gadgets, a Keccak core computing the χ-step on 8 slices in parallel, and $C = 4$ and $C = 5$, respectively. Since the ACLP-optimized decapsulation design does not fit onto the XC7A100T alongside the measurement infrastructure, we evaluate a purely area-optimized configuration instead, which uses a fully serialized Keccak core and $C = 1$. Due to its latency of almost 237 kcycles, we reduce the sampling rate to 50 MS/s, resulting in traces with roughly 2.4 million samples. Due to the clock-synchronized sampling, 10 samples per clock cycle remain sufficient for leakage assessment [OC15].

Evaluation methodology. We perform fixed-vs-random t-tests [SM15] either passing a fresh sharing of a fixed input value or of a random input value. We fix or randomize both sparse vectors for multiplication, $\mathsf{seed}_{\mathsf{KEM}}$ for key generation, $\mathbf{m}$ for encapsulation, and $\mathsf{seed}_{\mathsf{PKE,dk}}$ as well as σ for decapsulation, while public inputs are always fixed. Except for negligible probability, the random decapsulation therefore takes the implicit rejection path. To avoid false positives, our evaluation builds of encapsulation and key generation keep the derived $\mathsf{seed}_{\mathsf{PKE,ek}}$ constant by using a hardcoded value instead of unmasking the corresponding SHAKE output bits. Furthermore, rejections and duplicates in SampleFixedWeightVect$ cause a timing difference for some random inputs of key generation and decapsulation, which is not secret-dependent since the respective values are discarded. To avoid false positives, we exclude any random inputs which result in a different timing behavior than the fixed input, i.e., about 30 % of the key generation and 16 % of the decapsulation inputs. Lastly, the commonly used threshold of ±4.5 is inappropriate for very long traces as discussed by Ding et *al.* [DZD+17], since the type I error grows with the



(a) Multiplication (100k / 10M).

(b) Key generation (100k / 1M).

(c) Encapsulation (100k / 1M).

(d) Decapsulation (100k / 1M).

Figure 5. Side-channel measurement results. Each sub-plot contains: a sample trace (except multiplication), first-order t-test results with randomness disabled (labeled off; logarithmic scale beyond ±10) and enabled (labeled on), as well as an FSM trace. Time in milliseconds. TVLA threshold ±4.5 and length-adjusted threshold (see Section 6.4) in gray and red, respectively. Sub-captions contain the number of traces with randomness disabled / enabled.

number of samples, resulting in a false-positive rate of over 99 % for more than 1 million sample points. Following their methodology, we compute length-adjusted thresholds of ±6.11, ±6.15, and ±6.25 for key generation, encapsulation, and decapsulation.

Results. Results of all measurement campaigns are shown in Figure 5, including setup verification measurements with masking randomness disabled using 100 000 traces each. As expected, all evaluated designs show significant signs of leakage with maximum first-order t values of over ±100. With masking randomness enabled, the maximum absolute t value reaches 4.69 and 5.02 for the multiplication designs with hiding disabled / enabled after 10 million traces. Similarly, for key generation, encapsulation, and decapsulation, the maximum t values stay well below the length-adjusted threshold with 4.59, 5.10, and 4.41. The evaluation therefore implies that our masked designs are actually secure in practice, however, as discussed in Section 4, without capturing horizontal effects.

7 Conclusion

In this work, we presented the first fully masked hardware implementation of HQC, efficiently masking the sparse-dense multiplication and the fixed-weight vector sampling.

Compared to previous masked multiplication approaches in software, our novel first-order masking technique allows for significantly higher efficiency in hardware, almost

matching the latency of previous unprotected multipliers. We additionally demonstrated a practical horizontal attack against our design, locating all support indices within ± 8 positions after 200 traces, and integrated efficient and effective countermeasures. For the fixed-weight samplers, we proposed an architecture based on mixed-domain masking and constructed complete designs of key generation, encapsulation, and decapsulation.

Side-channel security of the multiplication module as well as of the full HQC implementation was verified using TVLA on an FPGA, showing no signs of leakage after 10 million and 1 million traces, respectively. Our performance evaluation furthermore highlights the wide range of trade-offs that can be achieved using our configurable hardware modules within the HADES framework [BLH+25].

References

[ABP25] Francesco Antognazza, Alessandro Barenghi, and Gerardo Pelosi. An efficient and unified RTL accelerator design for hqc-128, hqc-192, and HQC-256. *IEEE Trans. Computers*, 74(7):2306–2320, 2025.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. Verified proofs of higher-order masking. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 457–485. Springer, 2015.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. Strong non-interference and type-directed higher-order masking. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 116–129. ACM, 2016.

[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, *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, volume 10821 of *Lecture Notes in Computer Science*, pages 354–384. Springer, 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.

[BDK+25] Sebastian Bitzer, Jeroen Delvaux, Elena Kirshanova, Sebastian Maaßen, Alexander May, and Antonia Wachter-Zeh. How to lose some weight: a practical template syndrome decoding attack. *Des. Codes Cryptogr.*, 93(7):2503–2519, 2025.

[BLH+25] Fabian Buschkowski, Georg Land, Niklas Höher, Jan Richter-Brockmann, Pascal Sasdrich, and Tim Güneysu. HADES: automated hardware design exploration for cryptographic primitives. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(4):1–45, 2025.

[BSZ26] Gustavo Banegas, Benjamin Smith, and Jad Zahreddine. Exploiting load/store leakage of sparse vectors for key recovery in HQC. *CoRR*, abs/2607.19109, 2026.

[BZKG24] Luke Beckwith, Huizhen Zhou, Jens-Peter Kaps, and Kris Gaj. Power side-channel key recovery attack on a hardware implementation of BIKE. In *Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2024, Kobe, Japan, December 16-18, 2024*, pages 1–6. IEEE, 2024.

[CGLS21] Gaëtan Cassiers, Benjamin Grégoire, Itamar Levi, and François-Xavier Standaert. Hardware private circuits: From trivial composition to full verification. *IEEE Trans. Computers*, 70(10):1677–1690, 2021.

[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, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, Lecture Notes in Computer Science, pages 130–149. Springer, 2015.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, Lecture Notes in Computer Science, pages 188–205. Springer, 2014.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999.

[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.

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

[DXN+23] Sanjay Deshpande, Chuanqi Xu, Mamuri Nawan, Kashif Nawaz, and Jakub Szefer. Fast and efficient hardware implementation of HQC. In Claude Carlet, Kalikinkar Mandal, and Vincent Rijmen, editors, *Selected Areas in Cryptography - SAC 2023 - 30th International Conference, Fredericton, Canada, August 14-18, 2023, Revised Selected Papers*, volume 14201 of *Lecture Notes in Computer Science*, pages 297–321. Springer, 2023.

[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.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018.

[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 Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(3):223–263, 2022.

[GLG22] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. A new key recovery side-channel attack on HQC with chosen ciphertext. In Jung Hee Cheon and Thomas Johansson, editors, *Post-Quantum Cryptography - 13th International Workshop, PQCrypto 2022, Virtual Event, September 28-30, 2022, Proceedings*,

volume 13512 of *Lecture Notes in Computer Science*, pages 353–371. Springer, 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-Brokmann, Nicolas Sendrier, Jean-Pierre Tillich, and Valentin Vasseur. Hamming quasi-cyclic (hqc), 2025.

[GMA+26] 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), 2026.

[Gou01] Louis Goubin. A sound method for switching between boolean and arithmetic masking. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop, Paris, France, May 14-16, 2001, Proceedings*, volume 2162 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2001.

[GSA+25] Guillaume Goy, Maxime Spyropoulos, Nicolas Aragon, Philippe Gaborit, Renaud Pacalet, Fabrice Perion, Laurent Sauvage, and David Vigilant. Side-channel sensitivity analysis on HQC: Towards a fully masked implementation. Cryptology ePrint Archive, Paper 2025/1344, 2025.

[HB25] Vedad Hadzic and Roderick Bloem. Efficient and composable masked AES s-box designs using optimized inverters. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(1):656–683, 2025.

[HFG+26] 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 Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):1–27, Apr. 2026.

[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.

[HWCW19] Jingwei Hu, Wen Wang, Ray C. C. Cheung, and Huaxiong Wang. Optimized polynomial multiplier over commutative rings on fpgas: A case study on BIKE. In *International Conference on Field-Programmable Technology, FPT 2019, Tianjin, China, December 9-13, 2019*, pages 231–234. IEEE, 2019.

[HWZY26] Zhuo Huang, Weijia Wang, Xiaogang Zhou, and Yu Yu. DPA-style attacks on HQC. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2026(2):928–952, 2026.

[JLK+25a] Jaeho Jeon, Suseong Lee, Myeongjun Kim, Eunyoung Seo, Myunghyun Cho, Seonggyeom Kim, Bo Gyeong Kang, and Young-Sik Kim. A sparse polynomial multiplier for HQC integrating parallelism and power-based side-channel countermeasures. *IACR Cryptol. ePrint Arch.*, 2025:2003, 2025.

[JLK+25b] Jaeho Jeon, Suseong Lee, Myeongjun Kim, Eunyoung Seo, Myunghyun Cho, Seonggyeom Kim, Bo Gyeong Kang, and Young-Sik Kim. A sparse polynomial multiplier for HQC integrating parallelism and power-based side-channel countermeasures. *IACR Cryptol. ePrint Arch.*, 2025:2003, 2025.

[KM22] David Knichel and Amir Moradi. Low-latency hardware private circuits. 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 1799–1812. ACM, 2022.

[OC15] Colin O'Flynn and Zhizhang Chen. Synchronous sampling and clock recovery of internal oscillators for side channel analysis and fault injection. *J. Cryptogr. Eng.*, 5(1):53–69, 2015.

[PRJ+25] Thales B. Paiva, Prasanna Ravi, Dirmanto Jap, Shivam Bhasin, Sayan Das, and Anupam Chattopadhyay. Et tu, brute? side-channel assisted chosen ciphertext attacks using valid ciphertexts on HQC KEM. In *Post-Quantum Cryptography, PQCrypto 2025*, Lecture Notes in Computer Science, pages 294–321. Springer, 2025.

[PT19] Thales Bandiera Paiva and Routo Terada. A timing attack on the HQC encryption scheme. In Kenneth G. Paterson and Douglas Stebila, editors, *Selected Areas in Cryptography - SAC 2019 - 26th International Conference, Waterloo, ON, Canada, August 12-16, 2019, Revised Selected Papers*, volume 11959 of *Lecture Notes in Computer Science*, pages 551–573. Springer, 2019.

[RBG+25] Tomás Rabas, Jirí Bucek, Vincent Grosso, Karolína Zenknerová, and Róbert Lórencz. Profiling side-channel attack on HQC polynomial multiplication using machine learning methods. In Matthieu Rivain and Pascal Sasdrich, editors, *Constructive Approaches for Security Analysis and Design of Embedded Systems - First International Conference, CASCADE 2025, Saint-Etienne, France, April 2-4, 2025, Proceedings*, volume 15952 of *Lecture Notes in Computer Science*, pages 580–602. Springer, 2025.

[RCGG22] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu. Racing BIKE: improved polynomial multiplication and inversion in hardware. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):557–588, 2022.

[RHHM17] Melissa Rossi, Mike Hamburg, Michael Hutter, and Mark E. Marson. A side-channel assisted cryptanalytic attack against qcbits. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 3–23. Springer, 2017.

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

[SKC+19] Bo-Yeon Sim, Jihoon Kwon, Kyu Young Choi, Jihoon Cho, Aesun Park, and Dong-Guk Han. Novel side-channel attacks on quasi-cyclic code-based cryptography. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2019(4):180–212, 2019.

[SLP05] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In Josyula R. Rao and Berk Sunar,

editors, *Cryptographic Hardware and Embedded Systems - CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 - September 1, 2005, Proceedings*, volume 3659 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2005.

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015.

[SRSW20] Thomas Schamberger, Julian Renner, Georg Sigl, and Antonia Wachter-Zeh. A power side-channel attack on the CCA2-secure HQC KEM. In *Smart Card Research and Advanced Applications, CARDIS 2020*, volume 12609 of *Lecture Notes in Computer Science*, pages 119–134. Springer, 2020.

[SVP+25] Maxime Spyropoulos, David Vigilant, Fabrice Perion, Renaud Pacalet, and Laurent Sauvage. Masked vector sampling for HQC. In *Proceedings of the 22nd International Conference on Security and Cryptography, SECRYPT 2025*, pages 750–758. SciTePress, 2025.

[THKX23] Yazheng Tu, Pengzhou He, Çetin Kaya Koç, and Jiafeng Xie. LEAP: lightweight and efficient accelerator for sparse polynomial multiplication of HQC. *IEEE Trans. Very Large Scale Integr. Syst.*, 31(6):892–896, 2023.

[WBB+22] Guillaume Wafo-Tapa, Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Etienne Marcatel. A practicable timing attack against HQC and its countermeasure. *Adv. Math. Commun.*, 16(3):621–642, 2022.