Exploiting Load/Store Leakage of Sparse Vectors for Key Recovery in HQC

Gustavo Banegas[1], Benjamin Smith[1] and Jad Zahreddine[1,2]

[1] LIX, Inria, CNRS, École Polytechnique, Institut Polytechnique de Paris, France
gustavo@cryptme.in, smith@lix.polytechnique.fr
[2] eShard, France
jad.zahreddine@eshard.com

Abstract. Hamming Quasi-Cyclic (HQC) is a code-based key encapsulation mechanism selected by NIST for standardization, making its resistance to implementation attacks critically important. We present a side-channel attack that exploits load/store leakage in the manipulation of HQC's sparse secret vectors. Analysing Cortex-M4 assembly generated from the reference implementation, we identify a leakage surface in which the low and high 32-bit halves of each 64-bit word leak with different strengths, due to compiler-generated register spilling. We exploit this leakage to construct a simple zero-word distinguisher classifying machine words of the secret vector as zero or nonzero from electromagnetic measurements. The recovered zero positions are then translated into decoding hints, reducing HQC key recovery to a shortened syndrome-decoding problem. We analyse the resulting decoding complexity for all HQC parameter sets: at 32-bit granularity an expected 88.7% of the machine words of y are zero for HQC-1, cutting the decoding to $\approx 2^{46}$ bit operations. Experiments on a Cortex-M4 validate the predicted low/high-half asymmetry—approximately 500 traces for the stronger low-half channel and 5,000 for the weaker high-half channel—and recover the zero words of an HQC-1 key at 32-bit granularity. Finally, we discuss practical countermeasures that eliminate the sparsity exploited by the attack.

Keywords: HQC · Side-Channel Hints · Load-Store · Electromagnetic Side-Channel Analysis · Block-ISD.

1 Introduction

Hamming Quasi-Cyclic (HQC [GAMA+25]) has been selected by NIST for standardization as a post-quantum key encapsulation mechanism (KEM) [NIS], as a code-based complement for the lattice-based standard ML-KEM [NIS23]. As HQC moves toward standardization and deployment, its resistance to side-channel attacks—not only its algorithmic security—must be scrutinised.

Simple and differential power analysis (SPA/DPA) are well-established threats. The central questions are *which* operations leak exploitable information and *how* that leakage can aid key recovery. Prior work has mostly targeted *arithmetic* operations such as polynomial

Author list in alphabetical order; see https://ams.org/profession/leaders/CultureStatement04.pdf.

This work was supported by:

- the HYPERFORM consortium, funded by France through Bpifrance, and by the France 2030 program under grant agreement ANR-22-PETQ-0008 PQ-TLS.
- the SPARTAQUS consortium, funded by Germany through Agentur für Innovation in der Cybersicherheit GmbH.

Date of this document: 2026-07-21.

multiplication [BBB+24, YRZ+23] or other secret-dependent intermediates [RPJ+24]; leakage from merely *loading* or *storing* secret data has drawn little attention. For lattice-based schemes this is unsurprising: secret coefficients come from a non-trivial distribution on small integers, so the Hamming weight of a loaded machine word reveals little information on these coefficients.

HQC is different: its secret vector y is binary, and extremely sparse ($\omega = 66$ ones among $n = 17\,669$ bits for HQC-1), so the overwhelming majority of the machine words encoding y are identically zero. Marshall, Page, and Webb [MPW22] showed that on ARM Cortex-M4, consecutive load and store instructions leak the Hamming weight—and the Hamming distance—of the data on the memory bus, even across register-disjoint instructions. A single trace can therefore separate a zero word from a nonzero one at each step of the encoding loop. We turn this into a key-recovery attack by casting the problem as *decoding with hints*: every distinguishable nonzero word localizes at least one support position of y, shrinking the search space.

Decoding with hints. A growing line of work incorporates partial information (*hints*) about a secret error vector into Information-Set Decoding (ISD) to lower the cost of the underlying syndrome decoding problem (SDP). The general framework in [HPR+21] covers several hint types (known error or error-free locations, partial message knowledge, and sub-block weights), transforming decoding instances into strictly smaller ones and quantifying how many hints push the ISD work factor below the claimed security level of Classic McEliece [McE78], BIKE [ABB+24], and HQC. This is extended in [DEK26] to form reliability vectors from *perfect* and *approximate* hints, recasting the SDP as soft-decision decoding that interpolates smoothly between ISD complexity and polynomial time. Both works treat *hint acquisition as a black box*. We close that gap: our zero-word distinguisher produces, from electromagnetic traces, exactly the known-zero-position hints of [HPR+21] and the reliability information of [DEK26], so the resulting instance feeds both methods; in this work we implement and benchmark the ISD route and outline the statistical-decoding route.

Related work. Velek, Rabas, and Bucek [VRB26] give a single-trace SPA on the *Additional* HQC implementation, exploiting a lookup-table base-case multiplication to recover secret bits four at a time. Goy, Maillard, Gaborit, and Loiseau [GMGL24] mount a SASCA via belief propagation on the Reed–Solomon codec to recover the *shared key* from a single electromagnetic trace. Our work differs from these both in target and in generality: unlike [GMGL24], we recover the *long-term private key* y—reused across all decapsulations—rather than a per-session key, and we exploit load/store leakage of the sparse vector to feed a decoder rather than read secret bits directly, so our hints plug into [HPR+21] and [DEK26].

Contributions. We present a practical load/store side-channel attack on HQC that exploits the leakage of the secret sparse vector during polynomial multiplication, and carry it through to full key recovery. Concretely:

- **A new leakage surface.** We identify a previously unexplored load/store leakage in the `schoolbook_mul` routine of the reference HQC implementation (called by the higher-level Karatsuba multiplication), and show that the compiler-generated Cortex-M4 assembly (`-Os`) leaks each 64-bit word *asymmetrically*: the low 32-bit half is both loaded and spilled to the stack, whereas the high half is only loaded, so the two halves leak at different rates.
- **A zero-word distinguisher.** We build a simple Welch's t-distinguisher that classifies each 32-bit half-word of y as zero or nonzero. On an STM32F446RE it

separates the low half in ≈ 500 traces and the weaker high half in $\approx 5\,000$, and all words are classified in parallel from a single batch of traces.

- **Hints from leakage.** We reduce key recovery to a *shortened* syndrome-decoding instance whose known-zero hints feed directly into the decoding-with-hints frameworks of [HPR+21] and [DEK26]. At 32-bit (resp. 64-bit) granularity, an expected 88.7% (resp. 78.7%) of the words of y are zero for HQC-1.

- **Complexity of key recovery.** Exploiting the identity block of the shortened instance, we give a block-structured ISD that recovers the key, and we analyse its cost for all parameter sets. The half-word (32-bit) hints cut HQC-1 decoding to $\approx 2^{46}$ bit operations—about 40 s on a 48-core machine—versus several days for word (64-bit) hints; the compiler induced lower bit granularity is thus what turns an impractical decoding into a routine one.

- **Countermeasures.** We describe two mitigations—per-call additive masking, and storing y in the FAFFT transform domain—that remove the sparsity the attack relies on, and we discuss their cost.

2 Preliminary

2.1 Notation

As usual, $\mathbb{F}_2$ denotes the finite field with two elements. Given an integer $n \geq 1$, we define the ring

$$\mathcal{R} := \mathbb{F}_2[X]/(X^n - 1)\,.$$

We identify polynomials in $\mathcal{R}$ with their coefficient vectors (with respect to the monomial basis) in $\mathbb{F}_2^n$. In this representation, multiplication by some $a \in \mathcal{R}$ acts on vectors in $\mathbb{F}_2^n$ as multiplication by a circulant matrix $\mathrm{rot}(a)$. The Hamming weight of a vector $v \in \mathbb{F}_2^n$ is denoted by $\mathsf{hw}(v)$.

2.2 HQC in a nutshell

At a high level, HQC key generation (Algorithm 1) samples a public key $(h, s) \in \mathcal{R}^2$ and derives secret sparse vectors $x, y \in \mathbb{F}_2^n$ from a short seed such that

$$s = x + h \cdot y \in \mathcal{R} \tag{1}$$

(the product $\cdot$ here is multiplication in $\mathcal{R}$, not a vector dot product). The secret key SK can be the vectors x, y or the seed. From Equation 1, the public key PK is composed of a seed r_h that generates h and the result of the equation, namely, variable s. The PRG function is written as a simplification of the initialization of a hash function with the correct domain separator specified in the documentation of HQC [GAMA+25].

Encapsulation (Algorithm 4) uses fresh randomness to form a ciphertext $\mathsf{ct} = (u, v)$, while decapsulation (Algorithm 6) reconstructs the secret-dependent vector y from seed and computes expressions involving $u \cdot y$ (Algorithm 5). This step is where our leakage model applies: the multiplication routine loads the words of y in a regular pattern.

Here n is the ambient vector length (the smallest primitive prime greater than $n_1 n_2$), $k = 128$, 192, or 256 bits is the shared-key size, ω is the fixed Hamming weight of the secret vectors (x, y), and $\omega_r = \omega_e$ is the fixed weight of the ephemeral vectors (r_1, r_2, e); all values are given in Table 1.

In the following G, H, and I are SHA3-512 with domain separation.

Algorithm 1: HQC PKE key generation.

function HQC-PKE.KeyGen(seed)
  Output: Encryption key ekPKE, decryption key dkPKE
  $(\mathsf{seedPKE.dk}, \mathsf{seedPKE.ek}) \leftarrow I(\mathsf{seedPKE})$
  $\mathsf{ctxPKE.dk} \leftarrow$ XOF.Init(seedPKE.dk)
  $(\mathsf{ctxPKE.dk}, \mathbf{y}) \leftarrow$ SampleFixedWeightVect(ctxPKE.dk, $\mathcal{R}_\omega$)
  $(\mathsf{ctxPKE.dk}, \mathbf{x}) \leftarrow$ SampleFixedWeightVect(ctxPKE.dk, $\mathcal{R}_\omega$)
  $\mathsf{dkPKE} \leftarrow \mathsf{seedPKE.dk}$
  $\mathsf{ctxPKE.ek} \leftarrow$ XOF.Init(seedPKE.ek)
  $(\mathsf{ctxPKE.ek}, \mathbf{h}) \leftarrow$ SampleVect(ctxPKE.ek,$\mathcal{R}$)
  $\mathbf{s} \leftarrow \mathbf{x} + \mathbf{h} \cdot \mathbf{y}$
  $\mathsf{ekPKE} \leftarrow (\mathsf{seedPKE.ek}, \mathbf{s})$
  return (ekPKE, dkPKE)

Algorithm 2: HQC KEM key generation

function HQC-KEM.KeyGen()
  $\mathsf{seedKEM} \xleftarrow{\$} [0, 256)^{32}$ // 32 random bytes
  $\mathsf{ctxKEM} \leftarrow$ XOF.Init(seedKEM)
  $(\mathsf{ctxKEM}, \mathsf{seedPKE}) \leftarrow$ XOF.GetBytes(ctxKEM,*32*)
  $(\mathsf{ctxKEM}, \sigma) \leftarrow$ XOF.GetBytes(ctxKEM,*32*)
  $(\mathsf{ekPKE}, \mathsf{dkPKE}) \leftarrow$ HQC-PKE.KeyGen(seedPKE)
  $\mathsf{ekKEM} \leftarrow \mathsf{ekPKE}$
  $\mathsf{dkKEM} \leftarrow (\mathsf{ekKEM}, \mathsf{dkPKE}, \sigma, \mathsf{seedKEM})$
  return (ekKEM, dkKEM)

Algorithm 3: HQC PKE Encryption

function HQC-PKE.Encrypt(ekPKE,$\mathbf{m}$,θ)
  Input: Encryption key ekPKE, message $\mathbf{m}$, randomness θ
  Output: Ciphertext $(\mathbf{u}, \mathbf{v})$
  $(\mathsf{seedPKE.ek}, \mathbf{s}) \leftarrow \mathsf{ekPKE}$
  $\mathsf{ctxPKE.ek} \leftarrow$ XOF.Init(seedPKE.ek)
  $(\mathsf{ctxPKE.ek}, \mathbf{h}) \leftarrow$ SampleVect(ctxPKE.ek,$\mathcal{R}$)
  $\mathsf{ctx}_\theta \leftarrow$ XOF.Init(θ)
  $(\mathsf{ctx}_\theta, \mathbf{r}_2) \leftarrow$ SampleFixedWeightVect(ctx_θ, $\mathcal{R}_{\omega_r}$)
  $(\mathsf{ctx}_\theta, \mathbf{e}) \leftarrow$ SampleFixedWeightVect(ctx_θ, $\mathcal{R}_{\omega_e}$)
  $(\mathsf{ctx}_\theta, \mathbf{r}_1) \leftarrow$ SampleFixedWeightVect(ctx_θ, $\mathcal{R}_{\omega_r}$)
  $\mathbf{u} \leftarrow \mathbf{r}_1 + \mathbf{h} \cdot \mathbf{r}_2$
  $\mathbf{c} \leftarrow$ Encode($\mathbf{m}$) + Truncate($\mathbf{s} \cdot \mathbf{r}_2 + \mathbf{e}$, ℓ)
  return $(\mathbf{u}, \mathbf{v})$

Algorithm 4: HQC KEM Encapsulation

function HQC-KEM.Encaps(ekKEM)
  Input: Encapsulation key ekKEM
  Output: 32-byte shared secret key K, ciphertext cKEM
  $\mathbf{m} \leftarrow [0, 256)^k$
  $\mathsf{salt} \leftarrow [0, 256)^{16}$ // 16-byte random salt
  $(K, \theta) \leftarrow G(H(\mathsf{ekKEM} \parallel \mathbf{m} \parallel \mathsf{salt}))$
  $\mathsf{cPKE} \leftarrow$ HQC-PKE.Encrypt(ekKEM, $\mathbf{m}$, θ)
  $\mathsf{cKEM} \leftarrow (\mathsf{cPKE}, \mathsf{salt})$
  return (K, cKEM)

Algorithm 5: HQC PKE Decryption

function HQC-PKE.Decrypt(dkPKE, $(\mathbf{u}, \mathbf{v})$)
  Input: Decryption key ekPKE, ciphertext $(\mathbf{u}, \mathbf{v})$
  Output: Message $\mathbf{m}$
  $\mathsf{seedPKE.dk} \leftarrow \mathsf{dkPKE}$
  $\mathsf{ctxPKE.dk} \leftarrow$ XOF.Init(seedPKE.dk)
  $(\mathsf{ctxPKE.dk}, \mathbf{y}) \leftarrow$ SampleFixedWeightVect(ctxPKE.dk, $\mathcal{R}_\omega$)
  $\mathbf{m} \leftarrow$ Decode($\mathbf{v}$ − Truncate($\mathbf{u} \cdot \mathbf{y}$, ℓ))
  return m

Algorithm 6: HQC KEM Decapsulation

function HQC-KEM.Decaps(dkKEM, cKEM)
  Input: Decapsulation key dkKEM, ciphertext cKEM
  Output: Shared secret key K'
  $(\mathsf{ekKEM}, \mathsf{dkPKE}, \sigma) \leftarrow \mathsf{dkKEM}$
  $(\mathsf{cPKE}, \mathsf{salt}) \leftarrow \mathsf{cKEM}$
  $\mathbf{m}' \leftarrow$ HQC-PKE.Decrypt(dkPKE, cPKE)
  $(K', \theta') \leftarrow G(H(\mathsf{ekKEM} \parallel \mathbf{m}' \parallel \mathsf{salt}))$
  $\mathsf{cPKE}' \leftarrow$ HQC-PKE.Encrypt(ekKEM, $\mathbf{m}'$, θ'); $\mathsf{cKEM}' \leftarrow (\mathsf{cPKE}', \mathsf{salt})$
  $\bar{K} \leftarrow J(H(\mathsf{ekKEM}) \parallel \sigma \parallel \mathsf{cKEM})$
  if $\mathbf{m}' = \perp$ *or* $\mathsf{cKEM}' \neq \mathsf{cKEM}$ **then**
    $K' \leftarrow \bar{K}$
  return K'

Table 1 lists the parameters of each HQC instance. The values n_1 and n_2 are related to the code $\mathcal{C}$; the vector length parameter n is the smallest primitive prime greater than $n_1 n_2$. The parameter ω specifies the Hamming weight of the secret vectors: in HQC-1, for example, $n = 17\,669$ and $\omega = 66$.

Table 1: Parameter sets for HQC [GAMA+25]. Here n_1 and n_2 are the external and internal code lengths; n is the ambient vector length (note that n is the smallest primitive prime $> n_1 n_2$); k is the code dimension (bits); ω is the Hamming weight of secret vectors (x, y); and $\omega_r = \omega_e$ are the Hamming weights of the ephemeral vectors (r_1, r_2, e).

Instance	Security	n_1	n_2	n	k	ω	$\omega_r = \omega_e$	DFR
HQC-1	NIST-1	46	384	17 669	128	66	75	$< 2^{-128}$
HQC-3	NIST-3	56	640	35 851	192	100	114	$< 2^{-192}$
HQC-5	NIST-5	90	640	57 637	256	131	149	$< 2^{-256}$

Fixed-weight sampling. During decapsulation, the secret key seed is expanded using a pseudorandom generator to reproduce the vectors x and y via a deterministic procedure

$$y \leftarrow \texttt{SampleFixedWeightVect}(\cdot).$$

The vector y is a long binary vector of length n and very small Hamming weight $\omega \ll n$. This vector is then stored in memory and subsequently used in arithmetic operations.

2.3 Power side-channel analysis

Side-channel analysis (SCA) exploits physical quantities emitted by a device—power consumption, electromagnetic radiation, timing, or acoustic emissions—rather than weaknesses in the underlying algorithm. In a *power* side-channel attack, the adversary measures the current drawn by the target at each clock cycle. The resulting waveform, called a *power trace*, leaks information about the data being processed because the dynamic power dissipated by CMOS logic is proportional to the number of gate transitions, which depends on the data handled by the processor at each step [KJJ99].

In SPA, a single trace (or very few traces) suffices to distinguish between execution paths or operand values because the leakage is apparent from direct inspection [KJJ99]. SPA is effective when different code paths produce clearly different power profiles — the canonical example being the square-and-multiply exponentiation in RSA, where multiplications and squarings yield visually distinct patterns.

Leakage models. The relationship between an intermediate value v and the measured power consumption L is described by a *leakage model*. Three leakage models are relevant to our analysis.

First, the Hamming weight (HW) model assumes that power consumption is proportional to the number of bits set to one in the processed value:

$$L = a \cdot \mathsf{hw}(v) + b + \varepsilon, \qquad \varepsilon \sim \mathcal{N}(0, \sigma^2), \tag{2}$$

where a and b are device-dependent constants, and ε represents additive Gaussian measurement noise with mean 0 and variance σ^2. The HW model accurately captures the leakage behavior of many 8- and 32-bit microcontrollers, where the instantaneous power consumption is closely related to the number of active data lines [MOP07].

For pipelined processors and hardware in which power consumption is dominated by register or bus transitions, a more accurate model assumes that power consumption is proportional to the number of bits that change between successive values:

$$L = a \cdot \mathsf{hw}(v \oplus v_{\text{prev}}) + b + \varepsilon \tag{3}$$

where v_{prev} is the previous value of v in the bus.

Zero-value model. Our attack relies on a special case of the Hamming weight model, which we refer to as the *zero-value leakage model*. We assume that the power trace produced when processing an all-zero operand is statistically distinguishable from the trace corresponding to any nonzero operand, without requiring the exact Hamming weight of the operand to be observable. Defining the indicator variable

$$z = \mathbf{1}[\mathsf{hw}(v) > 0],$$

the leakage is modeled as

$$L = \mu_0 + \Delta \cdot z + \varepsilon, \tag{4}$$

where μ_0 is the mean leakage for the zero operand and Δ is the average increase in leakage for any nonzero operand.

2.4 Leakage model for HQC

During decapsulation, y is regenerated from the secret seed and consumed by the polynomial-multiplication routine, which loads its machine words in a fixed order (Section 3), yielding a sequence of power leakages aligned with the individual word loads.

Fix a machine-word size $W \in \{32, 64\}$, and let $m = \lceil n/W \rceil$. Partition the secret vector into W-bit words: $y = (y^{(0)}, \ldots, y^{(m-1)})$ with $y^{(j)} \in \mathbb{F}_2^W$, padding the last word with zeros. As we only need to tell whether a word is zero, set $z_j = 0$ if $y^{(j)} = 0$ and $z_j = 1$ otherwise. Writing $L_{t,j}$ for the leakage of the j-th word in trace t, the zero-value model gives

$$L_{t,j} = \mu_0 + \Delta\, z_j + \eta_{t,j}, \qquad \eta_{t,j} \sim \mathcal{N}(0, \sigma^2), \tag{5}$$

with μ_0 the mean leakage of a zero word and Δ the average increase for a nonzero word. Because the private key is reused, y is processed identically on every call, so an adversary can average T aligned traces, $\bar{L}_j = \frac{1}{T}\sum_t L_{t,j}$, with $\mathbb{E}[\bar{L}_j] = \mu_0 + \Delta z_j$ and $\mathrm{Var}(\bar{L}_j) = \sigma^2/T$: more traces sharpen the zero/nonzero decision. A classifier on $\bar{L}_j$ returns $\hat{z}_j \in \{0, 1\}$. The set of indices $J_0 = \{\, 0 \le j < m : \hat{z}_j = 0 \,\}$ of words judged zero determines the known-zero coordinate set

$$Z := \bigcup_{j \in J_0} \big\{\, jW, \ldots, \min\big((j{+}1)W - 1,\, n - 1\big) \big\},$$

with complement $S := \{0, \ldots, n-1\} \setminus Z$ and $n' = |S|$ the remaining unknown coordinates.

2.4.1 Decoding with leakage hints

The known-zero set Z restricts the secret to its unknown coordinates $y' := (y_i : i \in S) \in \mathbb{F}_2^{n'}$. Multiplication by h is the circulant matrix $\mathrm{rot}(h) \in \mathbb{F}_2^{n\times n}$, so relation (1) reads $s^\top = x^\top + \mathrm{rot}(h)\, y^\top$. Since $y_i = 0$ for every $i \in Z$, only the columns of $\mathrm{rot}(h)$ indexed by S contribute; writing $\mathrm{rot}(h)_S \in \mathbb{F}_2^{n\times n'}$ for that submatrix,

$$s^\top = x^\top + \mathrm{rot}(h)_S\, (y')^\top. \tag{6}$$

The number of equations is unchanged, but the number of unknowns drops from $2n$ to $n + n'$: the leakage modifies neither the code dimension nor the number of observations, removing only the coordinates known to be zero. This yields a *decoding problem with hints*,

$$H'(x \| y')^\top = s^\top, \qquad H' = \big[\, I_n \;\; \mathrm{rot}(h)_S \,\big] \in \mathbb{F}_2^{n\times(n+n')},$$

attackable by information-set decoding adapted to the known-zero positions, with the smaller unknown y' in place of y.

Remark 1. The reduction depends directly on the leakage quality: when a large fraction of the coordinates of y are identified as zero, $n' \ll n$ and the decoding instance becomes substantially smaller.

Effect of the word size. Since $\mathsf{hw}(y) = \omega \ll n$, most machine words are all-zero, and a smaller W raises the probability that an entire word is zero. Reducing W thus enlarges Z, shrinks n', and gives stronger hints and smaller decoding instances.

3 Searching for leakage

To assess and exploit the leakage, we must locate load and store operations on secret sparse vectors in the reference C code of HQC[1], since there is no side-channel resistant implementation publicly available.

As shown in Section 2, there are two types of secret sparse vectors in HQC: ones linked to a specific ciphertext $(\mathbf{r}_1, \mathbf{r}_2, \mathbf{e})$, i.e. regenerated at each `HQC-KEM.Encaps`[2] routine, and ones linked to a key pair $(\mathbf{x}, \mathbf{y})$, i.e. regenerated at each `HQC-KEM.KeyGen` routine. In the reference C implementation of HQC, these secret sparse vectors are manipulated in the following functions:

- vector addition in $\mathbb{F}_2^n$, corresponding to the C function `vect_add`;
- vector multiplication in $\mathbb{F}_2^n$, corresponding to `vect_mul` followed by `schoolbook_mul`;
- `SampleFixedWeightVect`, corresponding to either `vect_sample_fixed_weight1` or `vect_sample_fixed_weight2`, followed by `vect_write_support_to_vector`.

Table 2 lists the functions loading and storing these secret sparse vectors. There are multiple leakage targets for a single secret sparse vector in each HQC-KEM phase, for example: in `HQC-KEM.Decaps` we have both a load and a store leakage of $\mathbf{y}$.

Table 2: Secret sparse vector loads and stores in the HQC reference implementation.

	Vector	Loaded by C function	Stored by C function
`HQC-PKE.KeyGen`	$\mathbf{x}$	`vect_add`	`vect_write_support_to_vector`
	$\mathbf{y}$	`schoolbook_mul`	`vect_write_support_to_vector`
`HQC-PKE.Encrypt`	$\mathbf{r}_1$	`vect_add`	`vect_write_support_to_vector`
	$\mathbf{r}_2$	`schoolbook_mul` $(2\times)$	`vect_write_support_to_vector`
	$\mathbf{e}$	`vect_add`	`vect_write_support_to_vector`
`HQC-PKE.Decrypt`	$\mathbf{y}$	`schoolbook_mul`	`vect_write_support_to_vector`

We present several leakage sources of secret data in the C implementation. However, our main target is `HQC-PKE.Decrypt`, i.e., the decapsulation process. During decapsulation, the implementation computes $\mathbf{u} \cdot \mathbf{y}$ using `schoolbook_mul`, so the load leakage we exploit occurs during every decapsulation. Note that $\mathbf{y}$ is both stored, by `vect_write_support_to_vector` when it is regenerated from the `seed`, and loaded, by `schoolbook_mul`, within a single decapsulation, providing two leakage options per call. Moreover, this represents a more realistic attack model.

3.1 Analysis of target C functions

We analysed the reference C implementation of HQC to identify memory instruction (load and store) leakages. We are targeting Cortex-M4 microcontrollers, so we compiled the code for an ARMv7E-M target using GCC version 14.2.1, with the `Os` flag for size optimization (a realistic embedded device scenario). To qualitatively assess the load and store instructions

[1]The code is available in https://gitlab.com/pqc-hqc/hqc, tag v5.0.0 (commit f46e5422).

[2]Notation aligns with the 2025-08-22 specification of HQC: https://pqc-hqc.org/doc/hqc_specifications_2025_08_22.pdf.

leaking secret data, we analyze the assembly code (obtained with `objdump`) corresponding to each of the C functions listed in Table 2.

Analysis of `vect_add`. Listing 1 shows `vect_add`'s source code from the reference C implementation of HQC. We noticed that secret sparse vectors are always passed as the operand `v1` in the calls listed in Table 2. The *load* of the secret sparse vector from memory is highlighted in yellow. For comparison, we also highlight in red the result of the addition targeted by [HWZY26] in the case of $\mathbf{v} \oplus (\mathbf{uy})$. (Their work applies a chosen-ciphertext DPA targeting the output of the `vect_add` function processing two dense vectors.)

```
1 void vect_add(uint64_t *o, const uint64_t *v1, const uint64_t *v2, uint32_t size) {
2     for (uint32_t i = 0; i < size; ++i) {
3         o[i] = v1[i] ^ v2[i];
4     }
5 }
```

Listing 1: Constant-time vector addition in $\mathbb{F}_2^n$

Listing 2 gives an extract of the assembly code generated by the compiler. Notice the load double instruction `ldrd`: it loads a 64-bit word into a low and a high 32-bit register in 3 cycles.[3] This gives us our first source of leakage. With the `O3` optimization, the compiler uses the same `ldrd` instruction for fetching `v1[i]` from memory, hence `Os` and `O3` provide the same leakage source.

The low and high 32-bit values of `uint64_t` are loaded separately (as the datapath on a Cortex-M4 is only 32-bits), but it is hard to distinguish them without knowing the microarchitecture and having a very accurate probe. Hence, we consider this operation as having a 64-bit word leakage granularity: our distinguisher outputs a 64-bit value that is either zero or nonzero.

```
1 ldrd  r10, r11, [r1], #8  ; load low and high 32-bit values of v1[i] into r10 and r11
2 ldrd  r8, r9, [r2, #8]!
3 eor.w r4, r10, r8
4 eor.w r5, r11, r9
5 strd  r4, r5, [r0, #8]!   ; store low and high 32-bit values of o[i]
```

Listing 2: Extract of the assembly of `vect_add`, GCC 14.2.1 `-Os`

Analysis of `schoolbook_mul`. Vector multiplication in $\mathbb{F}_2^n$ is implemented with Karatsuba's algorithm: `vect_mul` calls the recursive function `karatsuba_mul`, which calls `schoolbook_mul` at each tail call. As before, secret sparse vectors are always passed as the first operand to `vect_mul`, `karatsuba_mul`, and `schoolbook_mul`. However, only slices of secret sparse vectors are passed at each tail call to `schoolbook_mul`. We can map each `schoolbook_mul` call to the indices of the secret sparse vector because calls are deterministic and not randomized.

Listing 3 gives `schoolbook_mul`'s source code. The load of the `a` operand, corresponding to slices of the secret sparse vector, is highlighted in yellow. The mask computation and operations highlighted in red constitute an additional attack surface: a 64-bit mask set to `0xFF...FF` indicates that `bit`-th bit of the `i`-th processed 64-bit word of the current secret sparse vector slice `a` is set to one. As this is rare, because of the sparsity, one can implement an SPA and recover the secret sparse vectors `y` and `r2`, according to Table 2. This attack

[3] According to the ARM documentation: https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions

was first presented by [VRB26], on a slightly different implementation: the Additional Implementation of HQC submitted to the 4th round of the NIST PQC competition. We successfully ported their SPA to the reference C implementation of HQC, but this work focuses only on the load of `a[i]`.

```
1  static void schoolbook_mul(uint64_t *r, const uint64_t *a, const uint64_t *b, size_t n) {
2      memset(r, 0, 2 * n * sizeof(uint64_t));
3      for (size_t i = 0; i < n; i++) {
4          uint64_t ai = a[i];
5          for (int bit = 0; bit < 64; bit++) {
6              uint64_t mask = -((ai >> bit) & 1ULL);
7              size_t base = i;
8              int sh = bit;
9              int inv = 64 - sh;
10             if (sh == 0) {
11                 for (size_t j = 0; j < n; j++) {
12                     r[base + j] ^= b[j] & mask;
13                 }
14             } else {
15                 for (size_t j = 0; j < n; j++) {
16                     r[base + j] ^= (b[j] << sh) & mask;
17                     r[base + j + 1] ^= (b[j] >> inv) & mask;
18                 }
19             }
20         }
21     }
22 }
```

Listing 3: Constant-time schoolbook multiplication in $\mathbb{F}_2^n$.

Listing 4 provides an extract of the assembly code for `schoolbook_mul`. The instructions corresponding to Line 4 of Listing 3—which manipulates secret data—are highlighted in yellow. Observe that the compiler preferred split loads (Lines 2 and 5) for the low and high 32-bit values of `a[i]` over an `ldrd` instruction. This is because it has to account for loop operations, like counter and address updates. Because of the split at the instruction level, this target will have a 32-bit word leakage granularity. Moreover, because of register spilling, there is a store (Line 4) of the low 32-bit value of `a[i]`; hence, we foresee higher leakage of the low part than of the high part. That is: a nonzero word having a `1` in the low 32-bits (i.e. positions 1 to 32) will leak more because the load is followed by a store, while a word with a `1` in the high 32-bits, (positions 33 to 64) will leak less because there is only a load.

```
1  ldr     r2, [sp, #36]   ; load the pointer address of a[i] from memory
2  ldr.w   r3, [r2, #8]!   ; load the low 32-bit values of a[i] into r3, increment i
3  str     r2, [sp, #36]   ; store the updated pointer address value for next iteration
4  str     r3, [sp, #44]   ; store the low 32-bit values of a[i] back (register spilling)
5  ldr.w   lr, [r2, #4]    ; load the high 32-bit values of a[i] into lr (r14)
```

Listing 4: Extract of the assembly of `schoolbook_mul`, GCC 14.2.1 `-Os`

Listing 5 gives the assembly output when `schoolbook_mul` is compiled with `O3` optimization. Interestingly, we get a new store of the high 32-bit values because of register spilling, increasing the surface of the leakage. *This is a concrete example where speed optimizations can increase side-channel leakage of manipulated secrets.* Indeed, the compiler optimization almost doubles the leakage of the secret value with the additional `str`.

```
1  ldr     r2, [sp, #56]   ; load the pointer address of a[i] from memory
2  ldr.w   r1, [r2, #8]!   ; load the low 32-bit values of a[i] into r3, increment i
3  str     r2, [sp, #56]   ; store the updated pointer address value for next iteration
4  ldr     r2, [r2, #4]    ; load the high 32-bit values of a[i] into lr (r14)
```

```
5  str     r1, [sp, #44]  ; store the low 32-bit values of a[i] back (register spilling)
6  str     r2, [sp, #48]  ; store the low 32-bit values of a[i] back (register spilling)
```

Listing 5: Extract of the assembly of `schoolbook_mul`, GCC 14.2.1 `-O3`

Analysis of `vect_write_support_to_vector`. Listing 6 shows the C source code for `vect_write_support_to_vector`, which turns a support (a list of bit positions) into a sparse vector, with each 64-bit word updated highlighted in yellow. In the reference C implementation of HQC, `vect_write_support_to_vector` is always called with a zero vector `v`, and `val` is initialized to 0 (Line 11), so at the end of the loop `val` holds the `i`-th element of the secret sparse vector and the logical "or" is effectively an assignment storing it to memory—our target. The mask computation highlighted in red provides a strong zero/nonzero indicator: an inner-loop mask of `0xFF...FF` marks the `i`-th element as nonzero. An SPA analogous to [VRB26], which we validated on `schoolbook_mul`, is therefore possible here too—but it would recover only zero 64-bit words, with no simple way to locate the set bits, and is otherwise identical to [VRB26], so we dismiss this target.

```
1  void vect_write_support_to_vector(uint64_t *v, uint32_t *support, uint16_t weight) {
2      uint32_t index_tab[PARAM_OMEGA_R] = {0};
3      uint64_t bit_tab[PARAM_OMEGA_R]   = {0};
4      for (size_t i = 0; i < weight; i++) {
5          index_tab[i] = support[i] >> 6;
6          int32_t pos  = support[i] & 0x3f;
7          bit_tab[i]   = ((uint64_t)1) << pos;
8      }
9      uint64_t val = 0;
10     for (uint32_t i = 0; i < VEC_N_SIZE_64; i++) {
11         val = 0;
12         for (uint32_t j = 0; j < weight; j++) {
13             uint32_t tmp = i - index_tab[j];
14             int val1 = 1 ^ ((tmp | -tmp) >> 31);
15             uint64_t mask = -val1;
16             val |= (bit_tab[j] & mask);
17
18         }
19         v[i] |= val;
20     }
21 }
```

Listing 6: Constant-time support-to-vector conversion.

Listing 7 gives the assembly for `v[i] |= val` (Line 19 of Listing 6) with the secret-storing instruction highlighted in yellow. As for `vect_add`, storing a 64-bit value on a 32-bit architecture compiles to a store double `strd` (identical under `O3`), so, as in Section 3.1, the leakage granularity is 64-bit.

```
1  ldrd   r2, r3, [r0, #8]! ; load low and high 32-bit values of v[i] (which are 0)
2  orr.w  r9, r2, r6        ; logical OR between low 32-bit values of v[i] and val
3  orr.w  r10, r3, r7       ; logical OR between high 32-bit values of v[i] and val
4  strd   r9, r10, [r0]     ; store the updated low and high 32-bit values of v[i]
```

Listing 7: Extract of assembly of `vect_write_support_to_vector` from GCC 14.2.1 `-Os`

3.2 Choice of the target C function

As we mentioned above, `HQC-KEM.Decaps` is the most realistic attack target. Our analysis above shows that conveniently, the C `schoolbook_mul` function used by `HQC-KEM.Decaps` is also the target most vulnerable to our methods.

Table 3 lists the assembly instructions that we can target in each of the C functions analyzed above with `Os` optimization, each with their leakage granularity. We chose to attack `schoolbook_mul` because of the smaller leakage granularity: that is, we can distinguish zero versus nonzero words of size 32-bits instead of the 64-bit words in the other two functions. Moreover, since `schoolbook_mul` has both load and store operations, this choice enables a leakage characterization of loads versus stores. Another interesting aspect in the context of `Os` optimization is the asymmetric leakage between the low 32-bit values of the 64-bit secret sparse vector word, which are loaded and stored, versus the high 32-bit values, which are only loaded. Future work can quantify the leakage of operations `ldrd` and `strd` coming from the remaining two functions.

Table 3: Target assembly instructions in each target C function

C function	Target assembly instructions	Leakage granularity
`vect_add`	`ldrd`	64-bit word
`schoolbook_mul`	`ldr.w, str, ldr.w`	32-bit word
`vect_write_support_to_vector`	`strd`	64-bit word

4 A theoretical attack

Recall from Section 2 that during KEM decapsulation the secret vector y is deterministically regenerated from the secret `seed`, stored in memory, and consumed by the schoolbook multiplication routine, which iterates over the 64-bit words $a[i] = y^{(j)}$ of y and loads each word before the inner mask–accumulate step (Section 3, Listing 3). Our goal is to recover y (and hence the full secret key) from the power leakage of these word loads: the distinguisher below classifies each $y^{(i)}$ as zero or nonzero, producing known-zero-position hints for the decoding-with-hints framework.

4.1 Leakage model

The channel we exploit is the load of the secret word $a[i]$ in `schoolbook_mul`, not the per-bit mask computation. On the target ARMv7E-M core (Cortex-M4) a 64-bit word is transferred as two 32-bit register operations, and under `-Os` the low half is additionally spilled to the stack, so the low half incurs a load *and* a store while the high half incurs only a load. We therefore model each half separately.

Write $y^{(j)} \in \mathbb{F}_2^{64}$ for the j-th word of y and let $y^{(j)}_{\mathrm{lo}}, y^{(j)}_{\mathrm{hi}} \in \mathbb{F}_2^{32}$ be its low and high 32-bit halves. Refining the word-level model (5) to the two halves, we introduce, for each half $h \in \{\mathrm{lo}, \mathrm{hi}\}$, the indicator

$$z_j^{(h)} = \begin{cases} 0, & y_h^{(j)} = 0, \\ 1, & \text{otherwise}, \end{cases} \tag{7}$$

and model the leakage sampled while word j is processed in trace t as

$$L_{t,j}^{(h)} = \mu_0^{(h)} + \Delta^{(h)}\, z_j^{(h)} + \eta_{t,j}^{(h)}, \qquad \eta_{t,j}^{(h)} \sim \mathcal{N}(0, \sigma^2), \tag{8}$$

with $\mu_0^{(h)}$ the mean leakage of a zero half and $\Delta^{(h)}$ the average increase for a nonzero one. Since the low half is both loaded and stored while the high half is only loaded, and also because stores leak more than loads, $\Delta^{(\mathrm{lo})} > \Delta^{(\mathrm{hi})}$: the low half is the stronger channel. This asymmetry, predicted by the compiled code, is confirmed experimentally in Section 5: the low half can be distinguished with an order of magnitude fewer traces. Under `-O3` the high half is spilled as well (Listing 5), raising $\Delta^{(\mathrm{hi})}$ toward $\Delta^{(\mathrm{lo})}$.

Trace averaging. Since the same private key y is reused, an adversary acquires T aligned traces and averages $\bar{L}_j^{(h)} = \frac{1}{T}\sum_t L_{t,j}^{(h)}$, with $\mathbb{E}[\bar{L}_j^{(h)}] = \mu_0^{(h)} + \Delta^{(h)} z_j^{(h)}$ and $\mathrm{Var}(\bar{L}_j^{(h)}) = \sigma^2/T$. The variance decreases at rate $1/T$, so both halves are eventually classifiable; the weaker high half simply requires proportionally more traces.

Word-level indicator. A 64-bit word is zero if and only if both halves are zero, so $z_j = z_j^{(\mathrm{lo})} \vee z_j^{(\mathrm{hi})}$. Classifying at 32-bit granularity thus subsumes the 64-bit classification while producing strictly more zero-coordinate information.

Sparse occupancy. Because y is extremely sparse, most machine words are identically zero. Partitioning $y \in \{0,1\}^n$ of weight ω into $m = \lceil n/W \rceil$ words of W bits, a standard sparse-occupancy approximation gives

$$\Pr[\text{word} = 0] \approx \left(1 - \frac{W}{n}\right)^{\omega} \approx \exp\left(-\frac{\omega W}{n}\right). \tag{9}$$

For HQC-1 ($n = 17\,669$, $\omega = 66$) this is ≈ 0.787 for $W = 64$ and ≈ 0.887 for $W = 32$: the half-word channel thus resolves positions more finely and eliminates a larger fraction of coordinates.

4.2 Zero-word distinguisher

For each (half-)word, the attacker compares its averaged leakage against a template built from processing a zero word, using a statistical test: a significant deviation classifies the (half-)word as nonzero, otherwise it is declared zero, yielding $\hat{z}_j^{(h)} \in \{0,1\}$ and hence $\hat{z}_j$. The asymmetry $\Delta^{(\mathrm{lo})} > \Delta^{(\mathrm{hi})}$ gives two regimes:

- **Word granularity ($W = 64$).** Classifying whole words only requires detecting a nonzero low *or* high half; the strong low-half channel dominates, so 64-bit words are separated with the fewest traces.
- **Half-word granularity ($W = 32$).** Distinguishing the halves separately requires the weaker high half, costing an order of magnitude more traces but increasing the expected known zero-coordinate fraction ρ from ≈ 0.787 to ≈ 0.887.

Remark 2. The distinguisher resolves *zero versus nonzero* at 32-bit granularity; it does not reveal the positions of the set bits inside a nonzero half-word. Locating the support within nonzero words is left to the "decoding-with-hints" step.

4.3 Reduction to a shortened syndrome decoding instance

Writing the public equation in vector form, let $\mathrm{rot}(h) \in \mathbb{F}_2^{n\times n}$ be the circulant matrix corresponding to h, so

$$s^\top = x^\top + \mathrm{rot}(h)\, y^\top. \tag{10}$$

With $e = (x \,\|\, y) \in \mathbb{F}_2^{2n}$ and $H = [I_n \mid \mathrm{rot}(h)] \in \mathbb{F}_2^{n\times 2n}$, this is the structured syndrome decoding instance

$$H\, e^\top = s^\top, \qquad \mathsf{hw}(x) = \mathsf{hw}(y) = \omega. \tag{11}$$

Let J_0 be the word indices classified as zero, inducing the zero-coordinate set

$$Z \supseteq \bigcup_{j \in J_0} \{jW, \ldots, \min((j+1)W - 1,\, n-1)\}, \tag{12}$$

where $W \in \{32, 64\}$ is the classification granularity. Let $S = \{0, \ldots, n-1\} \setminus Z$, $n' = |S|$, and $y' \in \mathbb{F}_2^{n'}$ the restriction of y to S. Substituting $y_i = 0$ for all $i \in Z$ into (10) and letting $\mathrm{rot}(h)_S$ keep only the columns indexed by S,

$$s^\top = x^\top + \mathrm{rot}(h)_S \, (y')^\top. \tag{13}$$

Defining the shortened parity-check matrix

$$H' = \begin{bmatrix} I_n & \mathrm{rot}(h)_S \end{bmatrix} \in \mathbb{F}_2^{n \times (n+n')}, \tag{14}$$

this becomes the *shortened* instance

$$H' \, (x \,\|\, y')^\top = s^\top, \qquad \mathsf{hw}(x) = \mathsf{hw}(y') = \omega. \tag{15}$$

The number of unknowns shrinks from $2n$ to $n + n'$. Writing $n' \approx (1-\rho)n$ when a fraction ρ of the coordinates of y is classified as zero, the *total* unknown length contracts by

$$\frac{2n}{n+n'} \approx \frac{2}{2-\rho}, \tag{16}$$

i.e. $\approx 1.80\times$ for HQC-1 at $W = 32$ ($\rho \approx 0.887$). Equivalently, the y-block alone shrinks from n to $n' \approx 0.113n$, a factor $1/(1-\rho) \approx 8.9\times$ on the sparse unknown—the quantity that drives the statistical-decoding gain of Section 4.6. The constraint $\omega(x) = \omega(y') = \omega$ is preserved, so every ISD technique exploiting the two-block weight structure carries over.

4.4 Robustness to classification errors

The reduction assumes every coordinate placed in Z is truly zero. The two classification errors are *not* symmetric:

- A *false nonzero* (a zero word declared nonzero) merely fails to remove a coordinate: n' is slightly larger and decoding marginally slower, but the true solution is untouched.
- A *false zero* (a nonzero word declared zero) places a genuine support coordinate of y into Z. Then $\mathsf{hw}(y') < \omega$ over S, the true solution leaves the search space of (15), and the decoder fails silently, since the weight check on x is never met.

Only false zeros are fatal, and they are one-sided, so the attacker biases the distinguisher toward "nonzero": a (half-)word is declared zero only when its averaged leakage is statistically *indistinguishable* from the zero template, rather than at the symmetric TVLA threshold. This suppresses false zeros at the cost of harmless false nonzeros, i.e. a slightly larger n'.

Quantitatively, the attack succeeds only if *every* nonzero word is caught. At $W = 32$, HQC-1 has $m = \lceil 17\,669/32 \rceil = 553$ half-words, of which $m(1-\rho) \approx 62$ are nonzero in expectation. If p is the per-word false-zero probability, the expected number of lost support coordinates is $\approx 62\,p$, so single-shot success requires $p \lesssim 1/62 \approx 1.6 \times 10^{-2}$, comfortably $p \le 10^{-3}$. Since $\mathrm{Var}(\bar{L}_j^{(h)}) = \sigma^2/T$, this is met by acquiring enough traces on the weaker high half; the reported T of Section 5 is chosen accordingly.

Remark 3. A residual handful of false zeros need not be fatal if the decoder is run with a small weight slack: allowing $\mathsf{hw}(y') \in \{\omega - \delta, \ldots, \omega\}$ over a support enlarged by the suspected-zero coordinates re-admits the lost positions at a modest increase in n', turning a hard failure into a graceful cost increase whenever the high-half channel is trace-limited.

4.5 ISD-style key recovery after leakage

Starting from (15), the block structure of H' permits a simple recovery. For any candidate $y' \in \mathbb{F}_2^{n'}$ of weight ω, the left block I_n uniquely determines

$$x = s + \mathrm{rot}(h)_S\, y', \tag{17}$$

so y' is valid if and only if $\omega(x) = \omega$. Key recovery reduces to searching for a weight-ω vector y' over n' positions subject to a single weight check on x, e.g. via meet-in-the-middle or block-wise enumeration [HPR+21, DEK26]. The leakage has transformed an instance of length $2n$ into one of length $n + n' \approx 1.113n$ (HQC-1, 32-bit), making the combinatorial search substantially more feasible.

Remark 4. To make the effect concrete, we solved (15) with plain Prange ISD on scaled quasi-cyclic instances (n prime, x, y of weight ω); the expected iteration count $\binom{n+n'}{2\omega}/\binom{n}{2\omega}$ shrinks rapidly as leakage removes coordinates. On a toy instance ($n = 211$, $\omega = 6$) revealing the zero words at $W = 32$ gave a 17× speed-up, and $W = 16$ a 35× one, matching the combinatorial prediction. Extrapolating to HQC-1 ($n = 17\,669$, $\omega = 66$), the expected Prange work falls from $\approx 2^{134}$ (no leakage) to $\approx 2^{39}$ at $W = 64$ and $\approx 2^{22}$ at $W = 32$; plain Prange is used only to expose the scaling, and dedicated ISD lowers these figures further.

4.6 Statistical decoding after leakage

The shortened instance also *admits* statistical decoding as an alternative to ISD, on the same instance (15): rather than enumerate, one recovers the support of y' from a bias in random linear projections. Only the ISD route is implemented and benchmarked here (Table 4); the sketch below shows the instance is amenable to soft-decision techniques [DEK26], but a complete decoder is left to future work.

Since h is uniform and n prime, h is invertible in $\mathcal{R}$ with overwhelming probability [GAMA+25]. Multiplying (13) by h^{-1} yields

$$t^\top = (h^{-1} \cdot x)^\top + (y')^\top, \tag{18}$$

where $t = h^{-1} \cdot s$ is publicly computable, $h^{-1} \cdot x$ is random-looking noise, and y' is the sparse signal. For any $\ell \in \mathbb{F}_2^{n'}$ and index i, $\langle \ell, t \rangle = \langle \ell, y' \rangle + \langle \ell, h^{-1}x \rangle$; the noise term has bias proportional to ω/n', and conditioning on $\ell_i = 1$ gives a detectable excess probability $(1 - 2\omega/n')/2$ for $\langle \ell, t \rangle = y'_i$ [DEK26]. Averaging over many random ℓ with $\ell_i = 1$ amplifies this into a test separating support coordinates of y' from zero ones.

The leakage reduces the ambient dimension from n to n': for HQC-1 at $W = 32$ the support density ω/n' rises by $1/(1-\rho) \approx 8.9$, concentrating the sparse signal and shrinking the search space. With enough traces the noise averages out, recovering the support of y'; x then follows uniquely from (17).

Remark 5. The advantage of the hints is a reduction of the ambient dimension, not an amplification of a single projection's bias: the elementary per-coordinate bias $(1 - 2\omega/n')$ in fact *decreases* as n' shrinks. Since the noise $h^{-1}x$ is per-coordinate unbiased, a concrete statistical decoder must exploit genuine low-weight parity checks, which we leave to future work.

5 Experimental Results

Our target is an STM32F446RE, a platform not evaluated in [MPW22]. The assembly of `schoolbook_mul` also differs from the kernels they construct: their closest kernel, LD-ST, matches the assembly of `vect_write_support_to_vector`, but in our case each operation

acts on two 32-bit registers holding one 64-bit C variable, whereas theirs manipulate a single 32-bit register. We therefore cannot transfer their conclusions directly, though they indicate that the leakage we target is plausible.

5.1 Experimental Setup

We target an STM32F446RE Nucleo board (Cortex-M4, clocked at 30 MHz). The firmware contains the targeted function `schoolbook_mul`, compiled with the size optimization `Os` using PlatformIO Core 6.1.18 (ARM GNU toolchain 14.2.1). We insert two software triggers wrapping the assembly code of Listing 4 to delimit the region of interest without additional signal processing.

To obtain realistic operands, we compile the reference C implementation of HQC, tagged as release v5.0.0 (commit f46e5422)[4], adding `printfs` to log the operands of the `schoolbook_mul` calls handling the secret sparse vectors `y` and `r2` (Table 2). Running multiple HQC instances on an x86 machine yields a set of sample vectors, which we replay on the target while capturing its electromagnetic (EM) emanations. We use a Langer RF-B 0,3-3 probe placed over a region of the Cortex-M4 identified by prior EM cartography, two Langer PA 303 amplifiers (3 GHz, 30 dB), and a Teledyne LeCroy WP404HD oscilloscope sampling at 1 GS/s.

5.2 Leakage assessment

A fine-grained assessment would distinguish the leakage of each individual bit, since even within one 32-bit register bit 1 and bit 3 leak differently. Because we capture EM emanations with a relatively large probe (3 mm), we adopt the simplification that all bits in the same 32-bit register share one leakage profile. The two 32-bit registers are used in *separate* instructions, so from the `uint64_t` perspective the cumulative leakage of bits 1-32 (the low register) is separable from that of bits 33-64 (the high register): the two contributions occur at distinct instants, as each half is loaded or stored in a separate instruction. Moreover, as shown in Section 3.1, the low half (a load *and* a store) leaks with significantly higher amplitude than the high half (a load only). These two criteria—*temporal occurrence* and *amplitude*—let the distinguisher attribute a detected nonzero pattern to the low or the high half, effectively giving a 32-bit leakage granularity.

Our goal is to separate a zero 64-bit word processed by `schoolbook_mul`'s main loop from the low and high halves of a nonzero one. We therefore build three sparse vectors:

- a zero vector;
- a vector with a single one at position 1 (processed in the low register);
- a vector with a single one at position 33 (processed in the high register).

We multiply each by a random ciphertext `u`, capture the EM leakage, and run two Test Vector Leakage Assessments (TVLAs): the zero vector against the low-register nonzero vector, and against the high-register nonzero vector. We expect the first to reach significance with fewer traces, since there both `ldr.w` and `str` leak, whereas only `ldr.w` leaks in the second (Section 3.1).

Figure 1 confirms this: the first TVLA crosses the threshold at ≈ 500 traces and the second at $\approx 5\,000$, so the low register leaks markedly more than the high one, matching the `Os` assembly of Listing 4. The TVLA threshold includes a correction over the L samples of the segment following [WO19].

At an equal trace count, the two TVLAs together characterise `ldr.w` versus `str` for a HW-1 nonzero word relative to a zero one. Figure 2 shows that on the STM32F446RE a

[4]The latest version (commit 161cd4fd) does not change any of the three functions analyzed in Section 3.

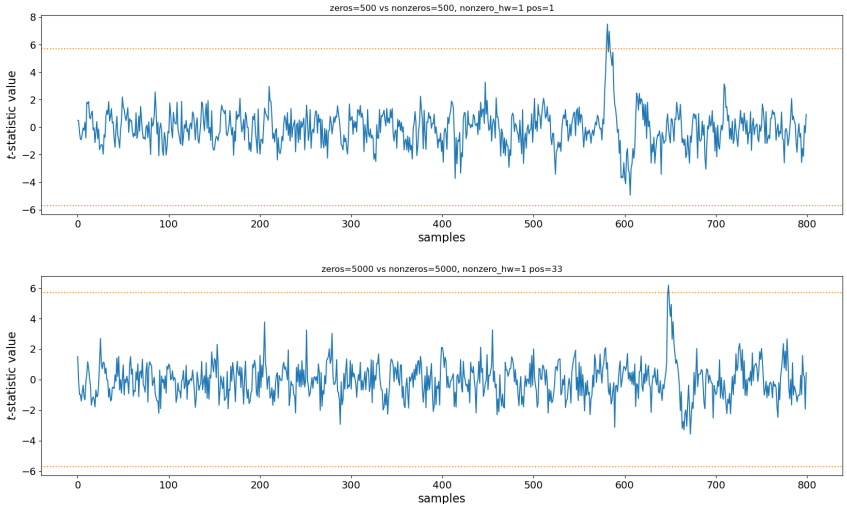

Figure 1: TVLA of zero vs. nonzero for bit position 1 with 500 traces per group (top) and bit position 33 with 5 000 traces per group (bottom). The title of each graph indicates the number of traces for a zero `uint64_t`, the number of traces for a nonzero, its hw, and the position of ones (1 to 64) in it.

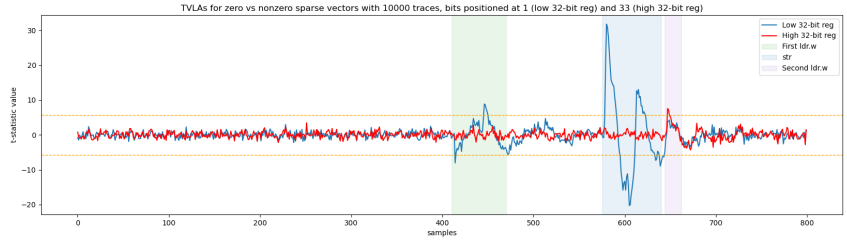

Figure 2: Superposed TVLAs over 10 000 traces for zero vs. nonzero sparse vectors, with the nonzero part processed in the low register (blue) and in the high register (red). Highlighted sections map the leakage regions to the leaking instructions.

memory write leaks more than a memory read; we map the leakage regions to the responsible instructions by highlighting, confirming the difference in both temporal occurrence and amplitude at equal traces.

5.3 Practical attack

Our attack is a *profiled* template attack. To build zero templates, we run `schoolbook_mul` on a zero sparse vector on the target device, then use the Welch t-test as the distinguisher. Because the leakage is strong and easily averaged over the reference implementation, plain zero templates suffice; Gaussian templates or more elaborate methods are unnecessary. Such methods may reduce the trace count and improve leakage exploitation on more noisy targets. We leave these left to future work as the distinguisher here already establishes our point: sparse vectors leak heavily through memory operations.

For a target HQC-1 execution, we capture many traces and, at each loop iteration, t-test the target `uint64_t` against the zero template of the corresponding iteration (avoiding noise from the memory address value leakage): a positive result marks it nonzero, otherwise zero. Since the `schoolbook_mul` iterations map deterministically to `uint64_t` positions in

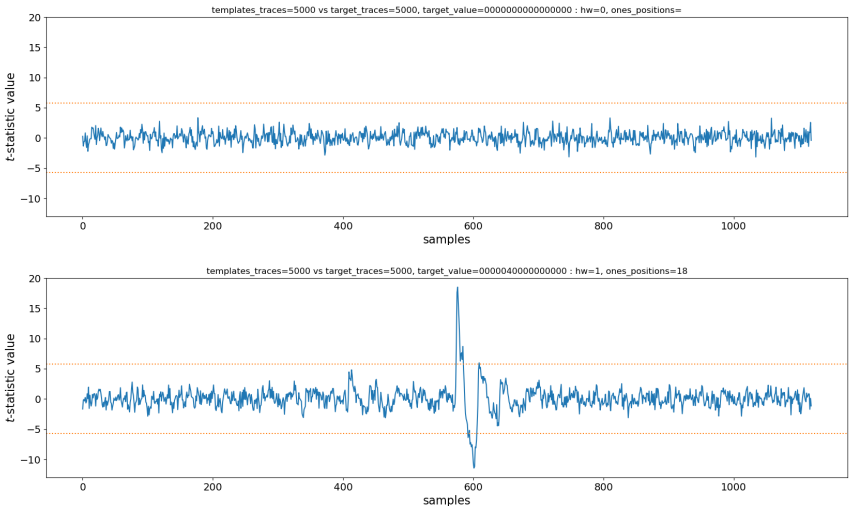

Figure 3: Attack on `y` being processed in `schoolbook_mul`, iteration 2 (top) and iteration 3 (bottom). The title of each graph indicates the number of zero template traces, the number of target traces, the target value in little-endian hexadecimal format, its hamming weight, and the position of ones (1 to 64) in it, if any.

the secret sparse vector (Section 3), all words are classified in parallel, and the bottleneck is only the trace count needed for a positive t-test: at worst $\approx 5\,000$, for a nonzero confined to the high register. This recovers the zeros of a secret sparse vector at 32-bit granularity.

The trace count is the same across all three HQC parameter sets, as they call the same `schoolbook_mul`; run in parallel, the attack time is also identical, the only differences being the per-trace acquisition time (which grows with the `vect_mul` length) and the number of parallel classifications (which grows with the `schoolbook_mul` iteration count).

Figure 3 shows `uint64_t` indices 2 and 3 (of $\{0, \ldots, 276\}$) for a `y` from an HQC-1 instance: the word at index 2 is entirely zero, while at index 3 the low half is nonzero and the high half is zero. Over the whole vector we classify all $m = 553$ 32-bit half-words of `y`, of which 62 are nonzero; we thus classify $\approx 88.8\%$ of the coordinates to be zero, in line with the expected 88.7% of Section 4.1. The nonzero half-words are then resolved by the decoder of Section 5.4. The threshold is the conservative, false-zero-suppressing choice of Section 4.4, since a single misclassified nonzero word would drop a true support coordinate and break recovery.

5.4 Remaining attack complexity

The end-to-end cost of the attack decomposes into three parts:

1. acquiring the EM traces,
2. zero-word classification, and
3. decoding the shortened instance (15).

We treat each in turn, summarising the figures for each parameter set in Table 4.

Trace acquisition. The distinguisher classifies every (half-)word of y in parallel from the *same* batch of aligned traces: a single `schoolbook_mul` execution exposes the loads of all $m = \lceil n/W \rceil$ words, so the query cost is the number T of traces, not $T \cdot m$. As

reported in Section 5, $T \approx 5\,000$ traces suffice for a reliable Welch-t decision in the worst case (a nonzero coefficient confined to the weaker high half); the strong low half is already separated at ≈ 500 traces. Because the same `schoolbook_mul` routine is used at all three security levels, T is essentially independent of the parameter set: only the acquisition time *per trace* grows with the length of `vect_mul`.

Classification. Each of the m (half-)words requires one Welch-t test between its T target traces and the zero-template traces over the L samples of the aligned segment, i.e. $O(m\,T\,L)$ arithmetic operations in total. For HQC-1 this is a few times 10^9 operations — negligible next to the acquisition and decoding, and trivially parallel over the m words.

Decoding. We solve the shortened instance (15) with the block-structured information-set decoder implicit in (17). Each iteration draws a set R of n' of the n equations, hypothesises $x_R = 0$, and solves the $n' \times n'$ linear system

$$(\mathrm{rot}(h)_S)_R\,(y')^\top = s_R^\top, \tag{19}$$

accepting the candidate if and only if $\mathsf{hw}(y') = \mathsf{hw}\big(s + \mathrm{rot}(h)_S\,y'\big) = \omega$. In other words, a draw succeeds exactly when the ω support positions of x all fall outside R and $(\mathrm{rot}(h)_S)_R$ is invertible. The information set R holds exactly $n' = |S|$ of the n equations, so its complement has size $n - n' = |Z|$; a fixed coordinate of x therefore lies outside R with probability $(n-n')/n = |Z|/n = \rho$, the same zero fraction measured by the distinguisher on y. The support of x avoids R with probability

$$\Pr[\,\mathrm{supp}(x) \cap R = \emptyset\,] = \frac{\binom{n-\omega}{n'}}{\binom{n}{n'}} = \prod_{i=0}^{\omega-1} \frac{n-n'-i}{n-i} \approx \left(1 - \tfrac{n'}{n}\right)^{\omega} = \rho^{\omega}, \qquad n' = (1-\rho)\,n. \tag{20}$$

Here $\rho = |Z|/n$ is a property of the leakage on y, whereas the event concerns the support of x; the identity $\rho = (n-n')/n$ that links them holds only because the decoder draws as many equations as there are unknowns, $|R| = n' = |S|$. The single approximation is the standard $\prod_i (n-n'-i)/(n-i) \approx (1-n'/n)^{\omega}$, valid for $\omega \ll n$; the final equality $(1-n'/n)^{\omega} = \rho^{\omega}$ is exact. The invertibility of $(\mathrm{rot}(h)_S)_R$ contributes a further factor $\approx P_{\mathrm{inv}} = \prod_{i\geq 1}(1-2^{-i}) \approx 0.29$ (measured ≈ 0.33 on our instances). The expected number of iterations and the total decoding work are therefore

$$\mathbb{E}[\mathrm{iters}] \approx \frac{\rho^{-\omega}}{P_{\mathrm{inv}}}, \qquad T_{\mathrm{dec}} \approx \frac{\rho^{-\omega}}{P_{\mathrm{inv}}} \cdot (n')^3 \ \text{ bit operations}, \tag{21}$$

where $(n')^3$ is the bit cost of the one Gaussian elimination in (19). We count *bit* operations in T_{dec}; on a w-bit machine the elimination is run on packed rows and costs $(n')^3/w$ *word* operations, so a plain 64-bit implementation already saves a factor $w = 64$ over the bit count (for HQC-1 at $W = 32$, $T_{\mathrm{dec}} = 2^{46.1}$ bit operations $= 2^{40.1}$ word operations). Two features make this efficient: the identity block fixes x from y', so an iteration only requires the ω ones of x to avoid R (rather than *all* 2ω errors to avoid an information set), which lowers the iteration count from $\rho^{-2\omega}$ to $\rho^{-\omega}$; and the elimination in (19) is $n' \times n'$ rather than $n \times n$. For HQC-1 at $W = 32$ this improves on the generic Prange bound of Section 4.5 from 2^{22} to 2^{13} iterations, each on a matrix a hundred times smaller.

Implementation and timing. Our reference implementation stores the rows of $(\mathrm{rot}(h)_S)_R$ as packed 64-bit words and performs one $n' \times n'$ elimination per iteration; on a single core[5] a `numpy`-based (i.e. unoptimised) solver takes ≈ 186 ms for HQC-1 at $W = 32$

[5] AMD EPYC 7702 64-Core Processor (128 Threads) @ 3.35 GHz with 1TB RAM

Table 4: Attack complexity per parameter set and leakage granularity W. ρ: fraction of coordinates classified zero; $n' = |S|$: remaining unknowns; $\mathbb{E}[\text{iters}]$ and T_{dec} (in *bit* operations) from (21). The wall-clock column is $\mathbb{E}[\text{iters}]\times$(measured per-iteration time)/48 for our `numpy` solver, *not* T_{dec} divided by a clock rate; per-iteration times scale as $(n')^3$. The trace count $T \approx 5\,000$ is common to all rows.

Instance	W	ρ	n'	$\log_2 \mathbb{E}[\text{iters}]$	$\log_2 T_{\text{dec}}$	wall-clock (48 cores)
HQC-1	64	0.787	3763	24.6	60.2	7.6 d
	32	0.887	1997	13.2	46.1	$\approx$ 40 s
HQC-3	64	0.837	5861	27.6	65.1	228 d
	32	0.915	3062	14.7	49.4	$\approx$ 6 min
HQC-5	64	0.865	7803	29.3	68.1	4.9 y
	32	0.930	4040	15.6	51.5	$\approx$ 26 min

($n' = 1997$), consistent with the $(n')^3/w \approx 1.2 \times 10^8$ word operations per iteration at the throughput of that solver. Parallelism is process-level: 48 workers run the randomised loop of (19) with independent seeds and the first to pass the weight check halts the rest, so with $\mathbb{E}[\text{iters}] \approx 2^{13.2}$ the expected wall-clock is $2^{13.2} \times 186\,\text{ms}/48 \approx 40\,\text{s}$. A dedicated $\mathbb{F}_2$ linear-algebra kernel (e.g. the method-of-four-russians solver of M4RI [AB26]) would lower the per-iteration cost.

Table 4 shows the complexity for a full attack and the wall-clock figures. These wall-clock values are *effective measurements*, not estimated by dividing T_{dec} by a clock rate. The decoding cost is dominated by (21) and depends critically on the leakage granularity: the half-word hints obtained from the *combined store+load* channel ($W = 32$, $\rho \approx 0.89$) reduce the decoding of HQC-1 to 2^{46} bit operations, which our parallel Python implementation solves in about 40 s on a 48-core machine, whereas the load-only whole-word hints ($W = 64$, $\rho \approx 0.79$) leave a 2^{60}-operation instance (several days on the same machine). The same $\approx 5\,000$ traces feed both regimes, so the store leakage is precisely what turns an impractical decoding into a routine one. At the higher security levels the $W = 32$ decoding stays feasible (minutes on a cluster), while the $W = 64$ variant becomes impractical.

6 Countermeasures

6.1 Additive masking

One lightweight countermeasure is to never store y in its plain sparse form. Instead, we can sample a fresh uniform-random $\mathbf{r} \in \mathbb{F}_2^n$ at each decapsulation, storing the pair $(\mathbf{r}, \mathbf{y}' := \mathbf{r} \oplus \mathbf{y})$. The product $u \cdot y$ is then recovered as

$$u \cdot y \;=\; u \cdot \mathbf{r} \;\oplus\; u \cdot \mathbf{y}', \tag{22}$$

by computing two multiplications and XOR-ing the results.

As long as $\mathbf{r}$ is freshly sampled per call, both operands are computationally indistinguishable from uniform, hence non-sparse: each word is zero with probability only 2^{-W}, far below the ≈ 0.887 rate exploited by our distinguisher. The load attack therefore does not apply to either multiplication, and the leakage is eliminated at its source. Note that $\mathbf{r}$ must be re-sampled every call: reusing it would expose a fixed, non-sparse $\mathbf{y}'$, letting an adversary average across invocations, cancel $\mathbf{r}$, and recover y.

The cost is performance: decapsulation now requires two full-length polynomial multiplications instead of one, roughly doubling the bottleneck operation, plus a one-time cost to generate $\mathbf{r}$ and to compute $\mathbf{y}'$ by flipping the ω support positions of y in $\mathbf{r}$.

6.1.1 Implementation and evaluation

We implemented the additive masking in the reference C implementation of HQC and measured its cost on all three parameter sets. The countermeasure is confined to the decapsulation product $u \cdot y$ computed in the `hqc_pke_decrypt` routine, and is selected at compile time so that the unmodified reference remains available as a baseline. The mask is drawn from the same DRBG the scheme already uses for the message and salt, never from the secret key seed, and is freshly sampled on every decapsulation.

Two variants. We realise the mask in two ways. *(i) Per-call masking* follows Section 6.1 directly: at each decapsulation a fresh $\mathbf{r} \in \mathbb{F}_2^n$ is sampled, $\mathbf{y}' = \mathbf{y} \oplus \mathbf{r}$ is formed, and $u \cdot y = u \cdot \mathbf{y}' \oplus u \cdot \mathbf{r}$ is evaluated with two multiplications and an XOR. The mask is discarded afterwards, so the secret key is unchanged. *(ii) Stored-share masking with refresh* moves the split to key generation. The secret key stores the two dense shares ($\mathbf{y}' = \mathbf{y} \oplus \mathbf{r}$, $\mathbf{r}$) in place of the seed, so the sparse vector y is never reconstructed in the coefficient domain during decapsulation. Each call first re-randomises the shares in place with a fresh $\boldsymbol{\delta} \in \mathbb{F}_2^n$,

$$\mathbf{y}' \leftarrow \mathbf{y}' \oplus \boldsymbol{\delta}, \qquad \mathbf{r} \leftarrow \mathbf{r} \oplus \boldsymbol{\delta}, \tag{23}$$

which preserves the invariant $\mathbf{y}' \oplus \mathbf{r} = y$ while changing both stored words, so that no fixed secret-derived value is loaded twice across calls, and then evaluates $u \cdot y = u \cdot \mathbf{y}' \oplus u \cdot \mathbf{r}$ as above. This variant keeps the coefficient-domain vector y entirely off the repeatedly attacked path—it appears only transiently at key generation—but it makes the decapsulation key *stateful*, since the refreshed shares are written back, and it enlarges the secret key, as two n-bit vectors replace the seed. A stateful key must be held in rollback-resistant storage: resetting it to a previous state would reinstate a repeated mask and re-enable trace averaging. Both variants are first-order: they remove the sparse single-share leakage the attack exploits, but not a hypothetical second-order combination of the two shares. We verified that both variants decapsulate correctly and return exactly the same shared secret as the unmodified reference across all parameter sets.

Setup. We benchmark the reference C implementation compiled with GCC 16.1.1 and with `-O3` on a 13th Gen Intel Core i7-1365U. For each parameter set and mode we report the median cycle count, obtained with `rdtscp` over 2000, 500, and 200 iterations for HQC-1, HQC-3, and HQC-5, respectively, at two granularities: the isolated decryption routine `hqc_pke_decrypt` (the operation the countermeasure protects) and the full decapsulation `crypto_kem_dec`. As the reference multiplication is a schoolbook/Karatsuba routine rather than an optimised kernel, the absolute counts are large; the *ratios* to the unmasked baseline are the implementation-independent quantities of interest.

Results. Table 5 reports the measurements. Masking the decryption product roughly doubles that routine—a factor of 1.95–1.98 across all parameter sets—matching the "two multiplications instead of one" prediction of Section 6.1. On the full decapsulation the overhead is only 1.32–1.33×, because decapsulation also re-encrypts (the Fujisaki–Okamoto transform) and decodes, so the extra multiplication is a smaller share of the total. The two variants cost essentially the same time—the XOR refresh and the DRBG draw are negligible next to the extra multiplication—so the choice between them is dictated by security and key size, not by speed. The stored-share variant enlarges the secret key by $2\lceil n/8 \rceil - 32$ bytes, a factor of 2.89–2.96, while the public key and ciphertext are unchanged[6].

[6]The code with this countermeasure is available in https://gitlab.inria.fr/eclair/hqcload.

Table 5: Cost of additive masking on the reference HQC implementation (13th Gen Intel Core i7-1365U, GCC 16.1.1 and with `-O3`, median cycles). "decrypt" is the isolated `hqc_pke_decrypt`; "decaps" is the full `crypto_kem_dec`; |sk| is the secret-key size in bytes. Factors are relative to the unmasked reference of each parameter set.

		decrypt		decaps		|sk|	
set	mode	cycles	×	cycles	×	bytes	×
HQC-1	reference	2 617 290	1.00	7 712 648	1.00	2321	1.00
	per-call	5 102 640	1.95	10 190 324	1.32	2321	1.00
	stored+refresh	5 091 156	1.95	10 184 322	1.32	6707	2.89
HQC-3	reference	7 744 522	1.00	23 087 538	1.00	4602	1.00
	per-call	15 270 688	1.97	30 638 202	1.33	4602	1.00
	stored+refresh	15 227 802	1.97	30 524 690	1.32	13534	2.94
HQC-5	reference	18 841 256	1.00	56 158 166	1.00	7333	1.00
	per-call	37 242 114	1.98	74 492 206	1.33	7333	1.00
	stored+refresh	37 146 158	1.97	74 357 750	1.32	21711	2.96

6.2 Storing y in the FAFFT transform domain

Polynomial multiplication is a bottleneck for HQC performance, and recently [RLC+25] and [CCPY26] have suggested accelerating this using the Frobenius Additive Fast Fourier Transform (FAFFT) algorithm in place of the Karatsuba multiplication proposed in the specification. If this is done, then one can precompute and store the transform $\hat{y} := \mathsf{FAFFT}(y)$ as the private-key representation, so that the sparse vector y is never loaded during decapsulation. This saves the forward FAFFT which would otherwise be applied to y in each decapsulation, to say nothing of the computation required to recover y from the secret `seed`, so it implies a significant time saving; however, $\hat{y}$ has twice the length of y (and far more than the length of `seed`), so this comes at the cost of a nontrivial space overhead.

The FAFFT of a (very) sparse vector is necessarily (very) dense by the standard uncertainty principle for discrete Fourier transforms (see e.g. [DS89]). However, given that ω is designed to be in $O(\sqrt{n})$, the uncertainty principle actually only tells us that $\mathsf{hw}(\hat{y})$ is in $\Omega(\sqrt{n})$. In practice, though, the weight of $\hat{y}$ is variable, and as a random sum of transform-domain basis vectors we expect it to be generally large and distributed essentially as for random vectors of length $2n$. In particular, The zero-word rate of $\hat{y}$ for random y (of weight ω) is expected to be far below the ≈ 0.887 our distinguisher relies on, and observing the loads of $\hat{y}$ therefore yields essentially no information. Even if some zero words are detected, the weight of most nonzero words should be relatively large—certainly beyond the limit where decoding hints are useful. Storing $\hat{y}$ instead of y (or `seed`) should therefore provide some natural resistance to our attack.

The FAFFT is not required, or even suggested, by the current HQC specification, so this countermeasure is only available to newer implementations that already use the FAFFT. Moreover, generating the private key directly in the transform domain (to protect `HQC-KEM.KeyGen`) is non-trivial: the standard fixed-weight sampler works in the coefficient domain, and converting the representation while preserving the public-key equation $s = x + h \cdot y$ requires care. We leave a complete treatment, and a detailed analysis of the impact on our attack, to future work.

7 Conclusion

We introduced a side-channel attack against HQC exploiting leakage from the loading and storing of secret sparse vectors during polynomial multiplication. Rather than recovering

secret bits directly, *the attack distinguishes zero from nonzero machine words and converts this information into decoding hints.* Reusing state-of-the-art decoding with hints [DEK26], these hints recover the secret key. For HQC-1, classifying at 32-bit granularity reveals up to 88.7% of the coordinates as zero, yielding a shortened syndrome-decoding instance that can be solved much more efficiently than the original.

More broadly, our results show that *memory accesses alone can constitute a powerful leakage source when manipulating highly sparse secrets*, highlighting the need to consider compiler-generated code and load/store operations in side-channel evaluations of PQC implementations.

Future work. More advanced distinguishers or template attacks may reduce the trace count and improve leakage exploitation on more noisy targets. Moreover, evaluating the attack across microarchitectures, compiler versions, and optimization levels would clarify the impact of compiler-generated code on leakage. Algorithmically, integrating the hints with soft-decision decoding, and checking whether similar leakage affects other PQC schemes relying on secret sparse vectors, are natural next steps. Finally, the transform-domain countermeasure calls for dedicated study: sampling fixed-weight vectors *directly* in the FAFFT domain—so the sparse coefficient-domain representation is never materialised or loaded—while remaining consistent with $s = x + h \cdot y$ is, to our knowledge, not well studied, and a complete implementation and security evaluation remain open.

Acknowledgements

We thank Pierre-Yvan Liardet, Rafael Carrera Rodriguez, Aurélien Vasselle, and Guillaume Bethouart for useful discussions and help with equipment.

References

[AB26] Martin Albrecht and Gregory Bard. *The M4RI Library – Version - 20260122*. The M4RI Team, 2026. https://github.com/malb/m4ri.

[ABB+24] Nicolas Aragon, Paolo L. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Santosh Ghosh, Shay Gueron, Tim Güneysu, Carlos Aguilar Melchor, Rafael Misocki, Edoardo Persichetti, Jan Richter-Brockmann, Nicolas Sendrier, Jean-Pierre Tillich, Valentin Vasseur, and Gilles Zémor. BIKE – Bit Flipping Key Encapsulation, 2024.

[BBB+24] Estuardo Alpirez Bock, Gustavo Banegas, Chris Brzuska, Lukasz Chmielewski, Kirthivaasan Puniamurthy, and Milan Sorf. Breaking DPA-protected Kyber via the pair-pointwise multiplication. In Christina Pöpper and Lejla Batina, editors, *Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part II*, Lecture Notes in Computer Science, pages 101–130. Springer, 2024.

[CCPY26] Ming-Shing Chen, Chun-Ming Chiu, Chun-Tao Peng, and Bo-Yin Yang. Accelerating HQC with Additive FFT. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):520–544, Apr. 2026.

[DEK26] Letizia D'Achille, Andre Esser, and Nicolai Kraus. Syndrome decoding with hints. *IACR Cryptol. ePrint Arch.*, 2026:341, 2026.

[DS89] David L. Donoho and Philip B. Stark. Uncertainty principles and signal recovery. *SIAM Journal of Applied Mathematics*, 49(3):906–931, 1989.

[GAMA+25] Philippe Gaborit, Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Edoardo Persichetti, Gilles Zémor, Jurjen Bos, et al. Hamming quasi-cyclic (HQC). *NIST Post-Quantum Standardization, 4th Round; National Institute of Standards and Technology: Gaithersburg, MD, USA*, 2025.

[GMGL24] Guillaume Goy, Julien Maillard, Philippe Gaborit, and Antoine Loiseau. Single trace HQC shared key recovery with SASCA. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):64–87, 2024.

[HPR+21] Anna-Lena Horlemann, Sven Puchinger, Julian Renner, Thomas Schamberger, and Antonia Wachter-Zeh. Information-set decoding with hints. In Antonia Wachter-Zeh, Hannes Bartz, and Gianluigi Liva, editors, *Code-Based Cryptography - 9th International Workshop, CBCrypto 2021, Munich, Germany, June 21-22, 2021 Revised Selected Papers*, Lecture Notes in Computer Science, pages 60–83. Springer, 2021.

[HWZY26] Zhuo Huang, Weijia Wang, Xiaogang Zhou, and Yu Yu. DPA-Style Attacks on HQC. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):928–952, Apr. 2026.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology – CRYPTO 1999*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[McE78] Robert J McEliece. A public-key cryptosystem based on algebraic. *Coding Thv*, 4244(1978):114–116, 1978.

[MOP07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[MPW22] Ben Marshall, Dan Page, and James Webb. MIRACLE: micro-architectural leakage evaluation A study of micro-architectural power leakage across many devices. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2022(1):175–220, 2022.

[NIS] National Institute of Standards and Technology. *Post quantum cryptography standardization*. https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Post-Quantum-Cryptography-Standardization.

[NIS23] National Institute of Standards and Technology. *Module-Lattice-Based Key-Encapsulation Mechanism Standard (ML-KEM)*, 2023. Federal Information Processing Standards Publication 203 (Initial Public Draft), https://doi.org/10.6028/NIST.FIPS.203.ipd.

[RLC+25] Antonio Ras, Antoine Loiseau, Mikaël Carmona, Simon Pontié, Guénaël Renault, Benjamin Smith, and Emanuele Valea. Optimizing HQC using Frobenius Additive FFT on a RISC-V-based System-on-Chip. In *2025 28th Euromicro Conference on Digital System Design (DSD)*, pages 608–615, 2025.

[RPJ+24] Prasanna Ravi, Thales B. Paiva, Dirmanto Jap, Jan-Pieter D'Anvers, and Shivam Bhasin. Defeating Low-Cost Countermeasures against Side-Channel Attacks in Lattice-based Encryption A case study on Crystals-KYBER. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(2):795–818, 2024.

[VRB26] Pavel Velek, Tomás Rabas, and Jirí Bucek. Simple power analysis of polynomial multiplication in HQC. In Roberto Di Pietro, Karen Renaud, and Paolo Mori, editors, *Proceedings of the 12th International Conference on Information Systems Security and Privacy, ICISSP 2026, Marbella, Spain, March 4-6, 2026, Volume 2*, pages 299–306. SCITEPRESS, 2026.

[WO19] Carolyn Whitnall and Elisabeth Oswald. A critical analysis of ISO 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In Steven D. Galbraith and Shiho Moriai, editors, *Advances in Cryptology - ASIACRYPT 2019 - 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8-12, 2019, Proceedings, Part III*, volume 11923 of *Lecture Notes in Computer Science*, pages 256–284. Springer, 2019.

[YRZ+23] Bolin Yang, Prasanna Ravi, Fan Zhang, Ao Shen, and Shivam Bhasin. STAMP-single trace attack on M-LWE pointwise multiplication in Kyber. *IACR Cryptol. ePrint Arch.*, 2023:1184, 2023.